Larger-scale model training on multi-GPU systems

Giuseppe Fiameni – gfiameni@nvidia.com

ELLIS Summer School on Large-Scale AI for Research and Industry Modena, 18-22 September 2023

About Me

Giuseppe Fiameni – gfiameni@nvidia.com

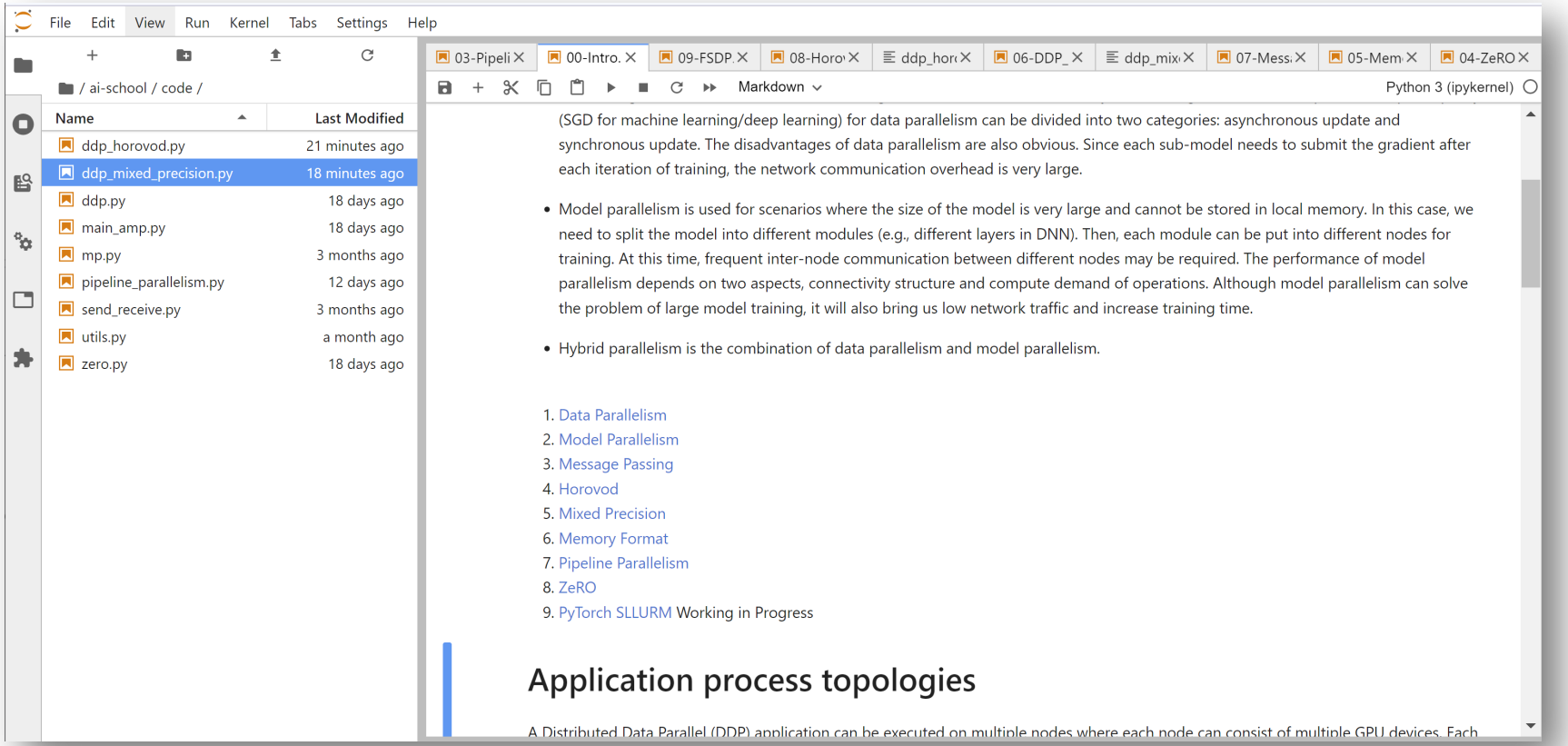
- AI & HPC @ NVIDIA
  - *Support Higher Education and Research through collaboration projects*
- Lead engineer of the NVIDIA AI Technology Center, Italy
  - *Agreement among CINI, CINECA and NVIDIA to accelerate scientific discovery*
- 10+ years experience delivering HPC applications and large data processing solutions at CINECA (www.hpc.cineca.it)

Why this tutorial?

- Deep Learning is transitioning from being a ***computer science*** *towards a* ***computational science***.
- Advanced computing and large-scale infrastructure are fundamental to conduct science i.e., train gigantic models, process massive data, achieve better performance, reduce time to solutions.

Notebooks & CODE

https://gitlab.hpc.cineca.it/gfiamen1/hpc-dl-ellis-2023



CINECA Leonardo system

https://leonardo-supercomputer.cineca.eu/

- Nodes: 3456 booster nodes
- Processors: Intel Xeon 8358 32 cores, 2.6 GHz
- Accelerators: **4 x NVIDIA custom Ampere GPU 64GB HBM2**
- RAM: (8x64) GB DDR4 3200 MHz
- Network: 2 x NVIDIA HDR cards 2x100Gb/s
- 106 PB (raw) Large capacity storage, 620 GB/s

4th fastest supercomputer worldwide

AGENDA

Why Large datasets and Large Neural Networks?

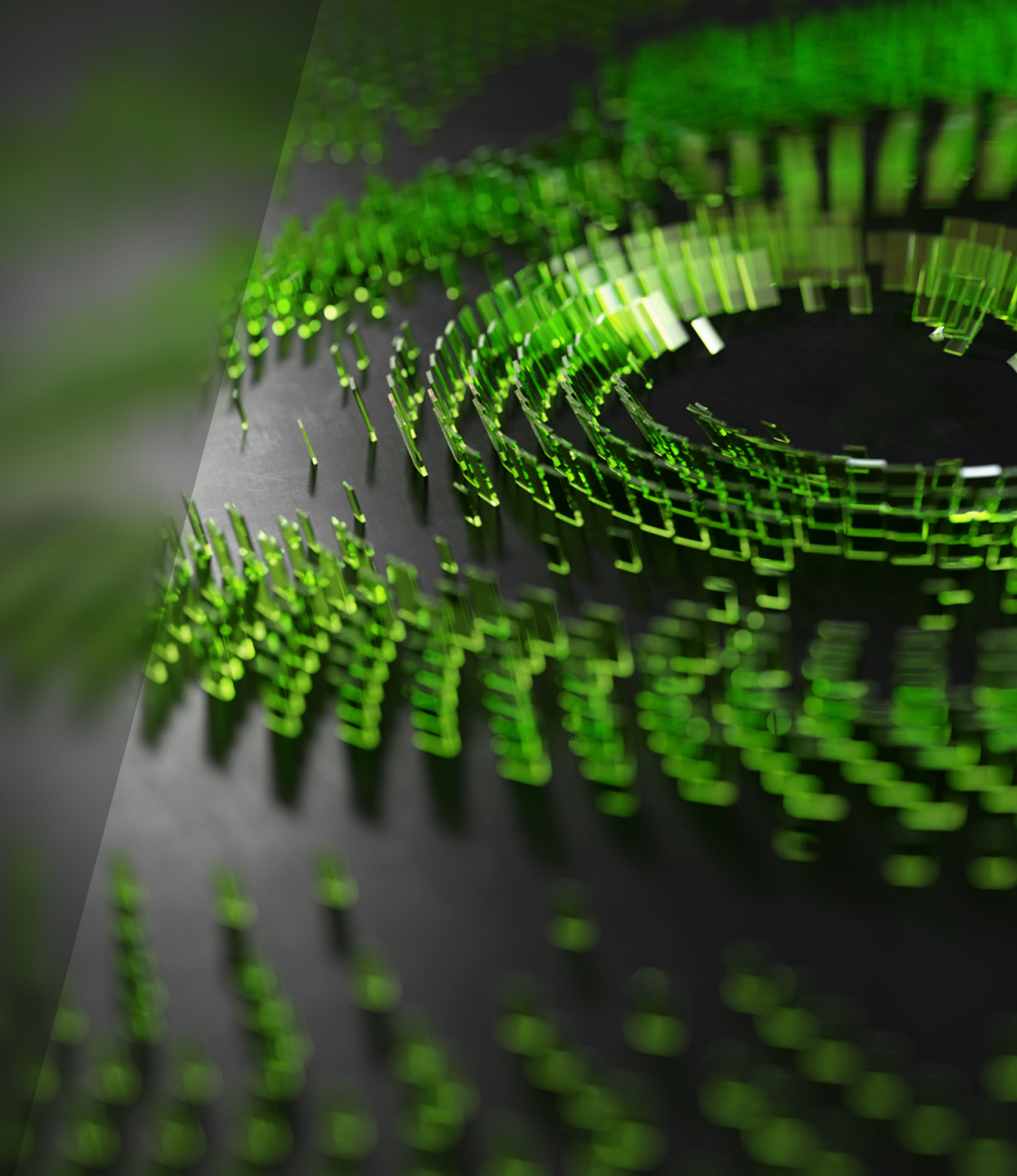
Scaling law

Training large neural networks

Single GPU optimization

Multi GPU optimization

Hands-on using CINECA resources



Andrej Karpathy
@karpathy

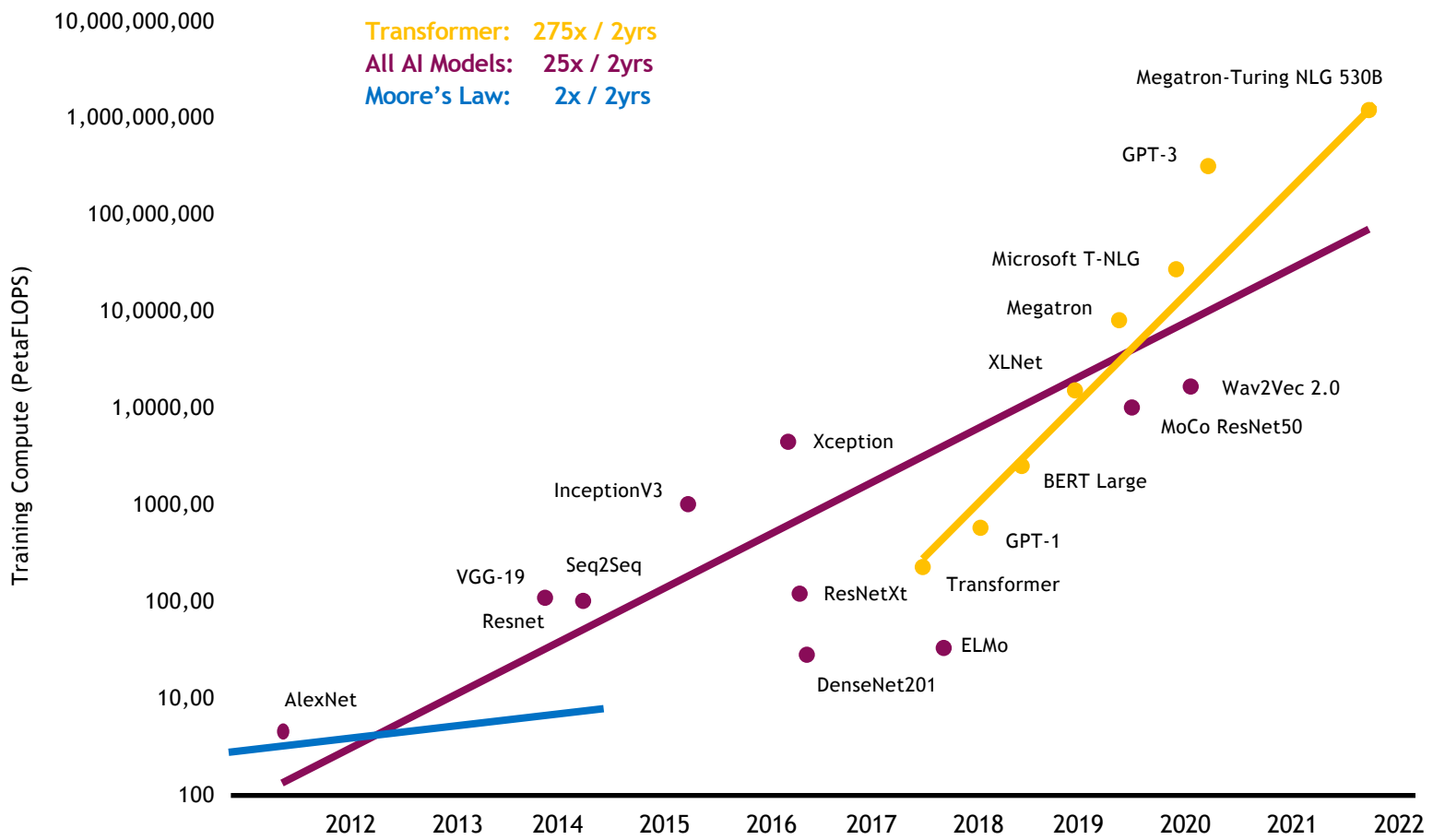
The most dramatic optimization to nanoGPT so far (~25% speedup) is to simply increase vocab size from 50257 to 50304 (nearest multiple of 64). This calculates added useless dimensions but goes down a different kernel path with much higher occupancy. Careful with your Powers of 2.

Andrej Karpathy @karpathy · Feb 4
Replying to @abhi_venigalla and @MosaicML

I love how sometimes changing one integer/flag can have the same impact as a 1 month optimization project. You just know there is some OMP_NEVER_HEARD_OF=3 that gets addition 3% MFU. Or my personal favorite - that undocumented bios flag that only 4 people on Earth know exists :D

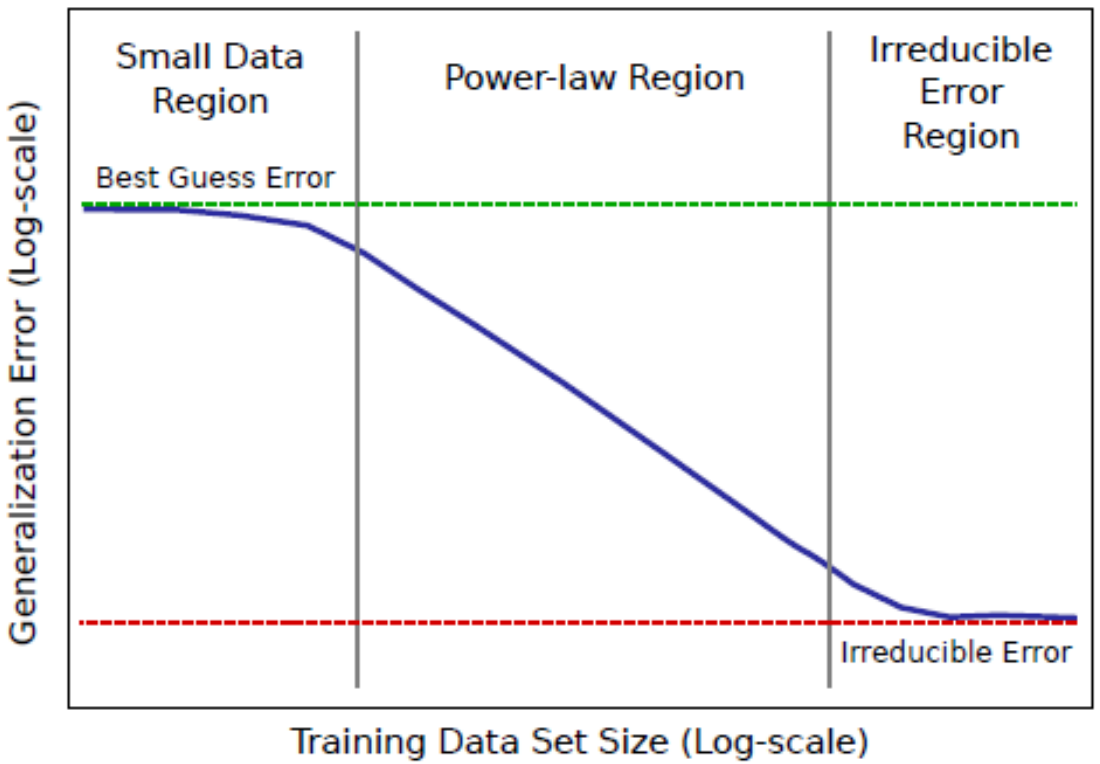
Dramatic increase in Model Sizes

The Trend Continues



NEW PROMISE

Logarithmic relationship between the dataset size and accuracy



Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., ... & Zhou, Y. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv preprint arXiv:1712.00409.

Scaling Laws apply to NLP

As you increase the dataset size, you must also increase the model size

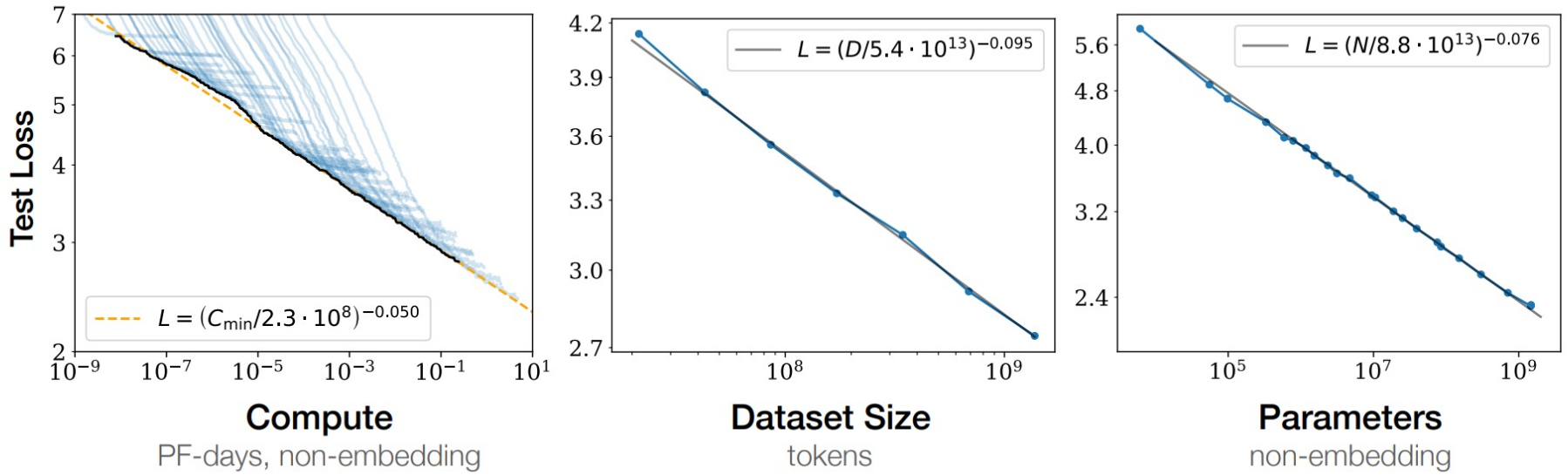


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.

Scaling Laws apply to computer vision too

Increase in performance is proportional to the model size and dataset size

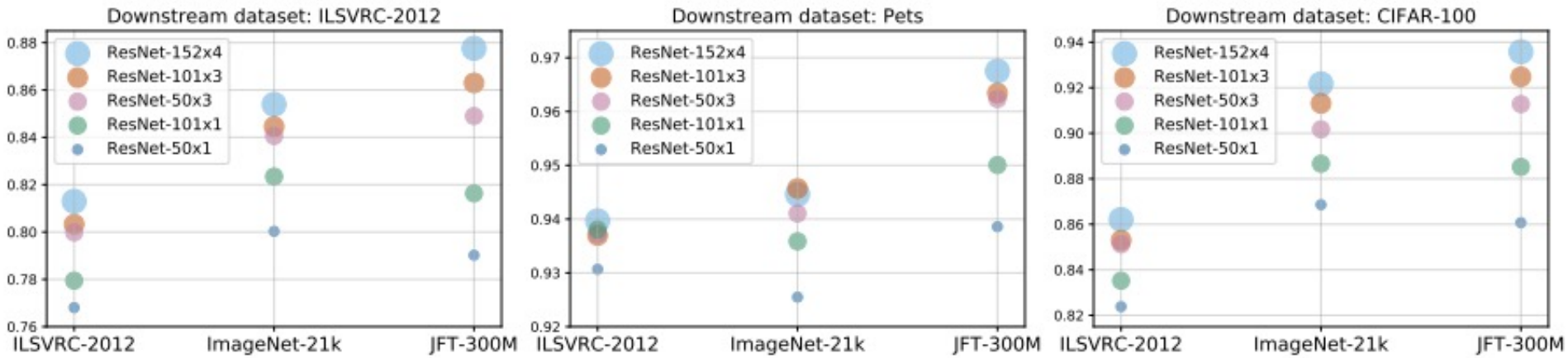
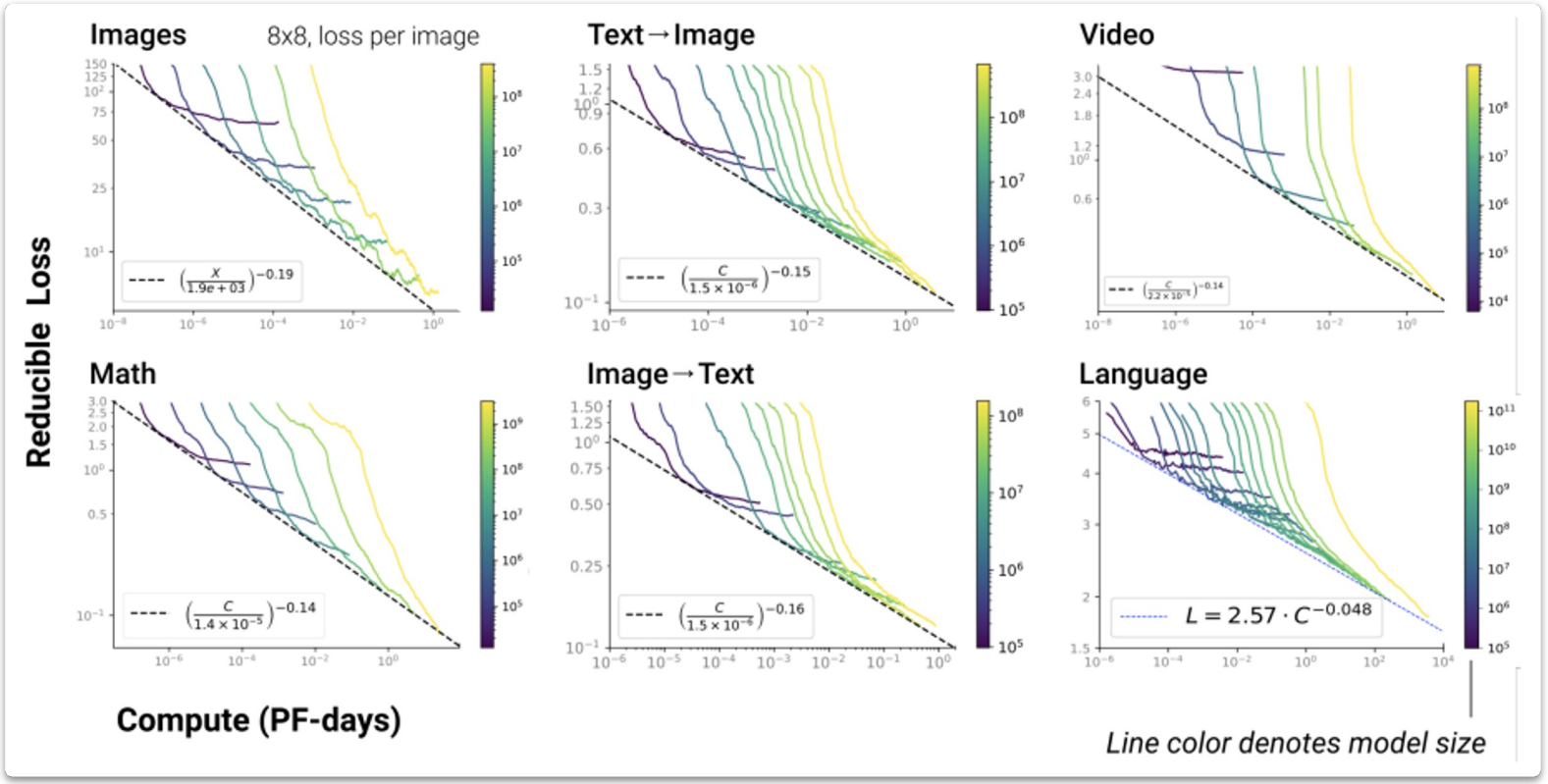


Fig. 5: Effect of upstream data (shown on the x-axis) and model size on downstream performance. Note that exclusively using more data or larger models may hurt performance; instead, both need to be increased in tandem.

Empirical evidence

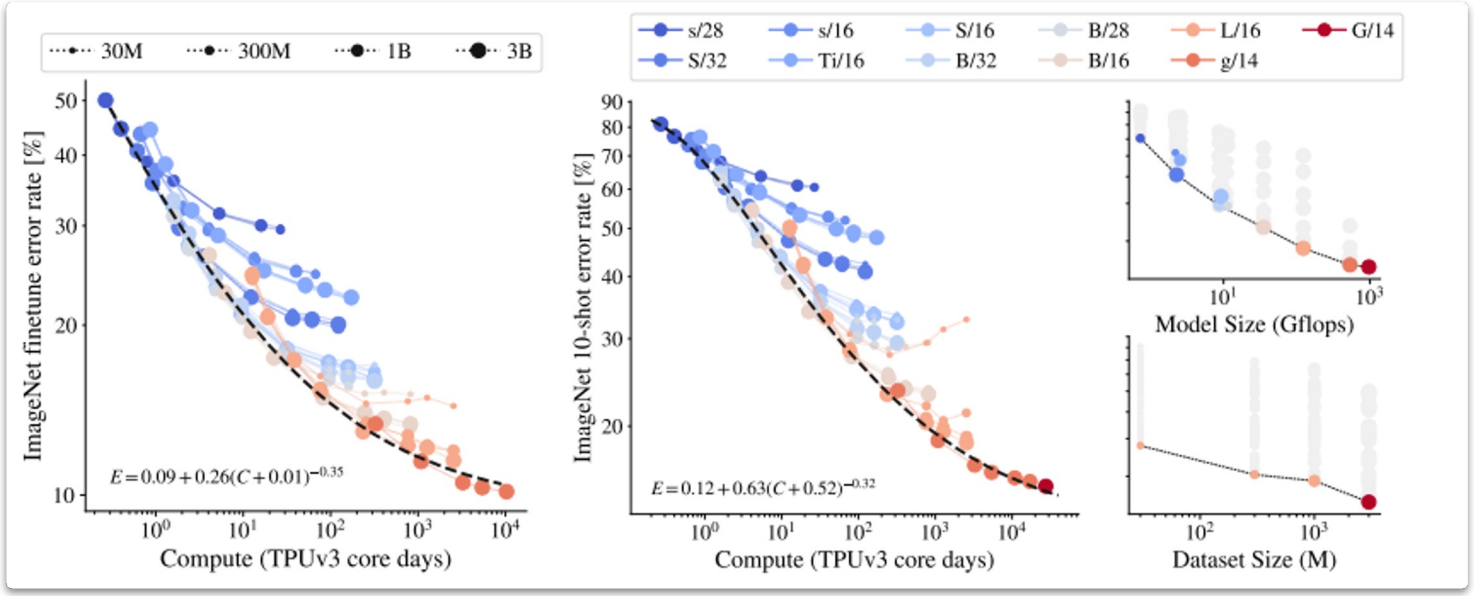
The Scaling Laws for generative models



Henighan, Tom, et al. "**Scaling laws for autoregressive generative modeling**." *arXiv preprint arXiv:2010.14701* (2020).

Empirical evidence

The Scaling Laws in Computer Vision & Speech



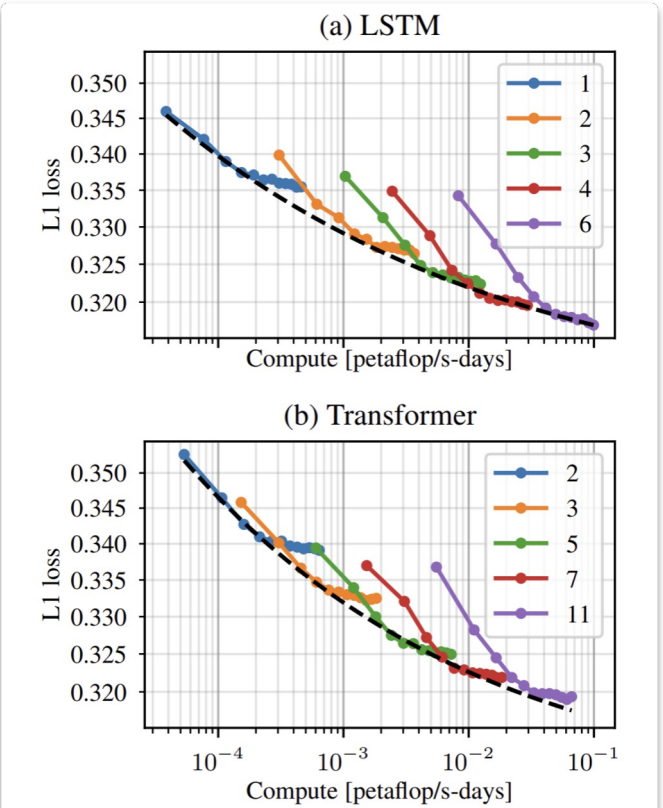


Figure 5: *Development set loss for both LSTM and Transformer models for models with the indicated number of layers. The dashed line represents the computationally efficient frontier defined in Eq. 4.*

Zhai, Xiaohua, et al. "**Scaling vision transformers**." *arXiv preprint arXiv:2106.04560* (2021).

Droppo, Jasha, and Oguz Elibol. "**Scaling Laws for Acoustic Models**." *arXiv preprint arXiv:2106.09488* (2021).


IT IS MORE THAN JUST ACCURACY

Importance of Dataset size

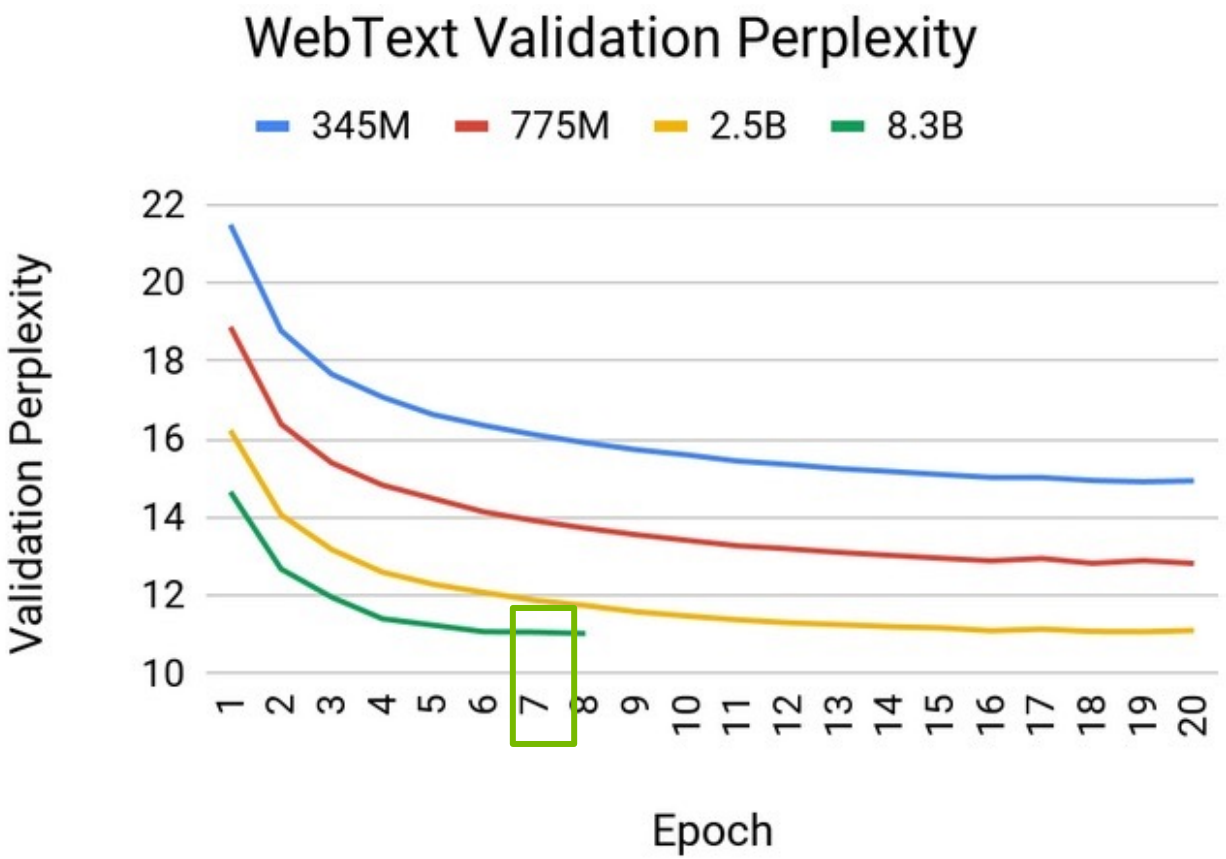
Dataset size more important than neural network design

"… more importantly, we find that the precise architectural hyperparameters are unimportant compared to the overall scale of the language model."

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.

Even more importantly

Large Neural Networks use data more efficiently

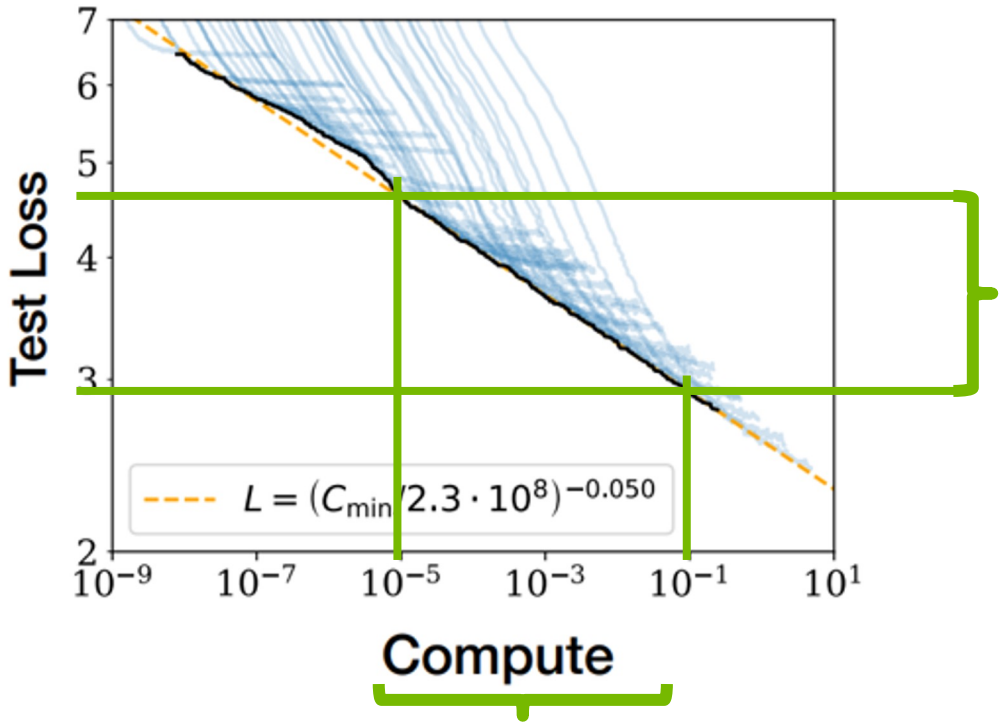


Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. arXiv preprint arXiv:1909.08053

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Agarwal, S. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*..

Are Large language models worth it?

The cost of incremental improvement



Are we building those models only for the small incremental improvement in their performance?

Is it worth all the engineering and computational investment?

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, Sam McCandlish. Scaling Laws for Autoregressive Generative Modeling.2020

Few shot learning

Learning from far fewer examples

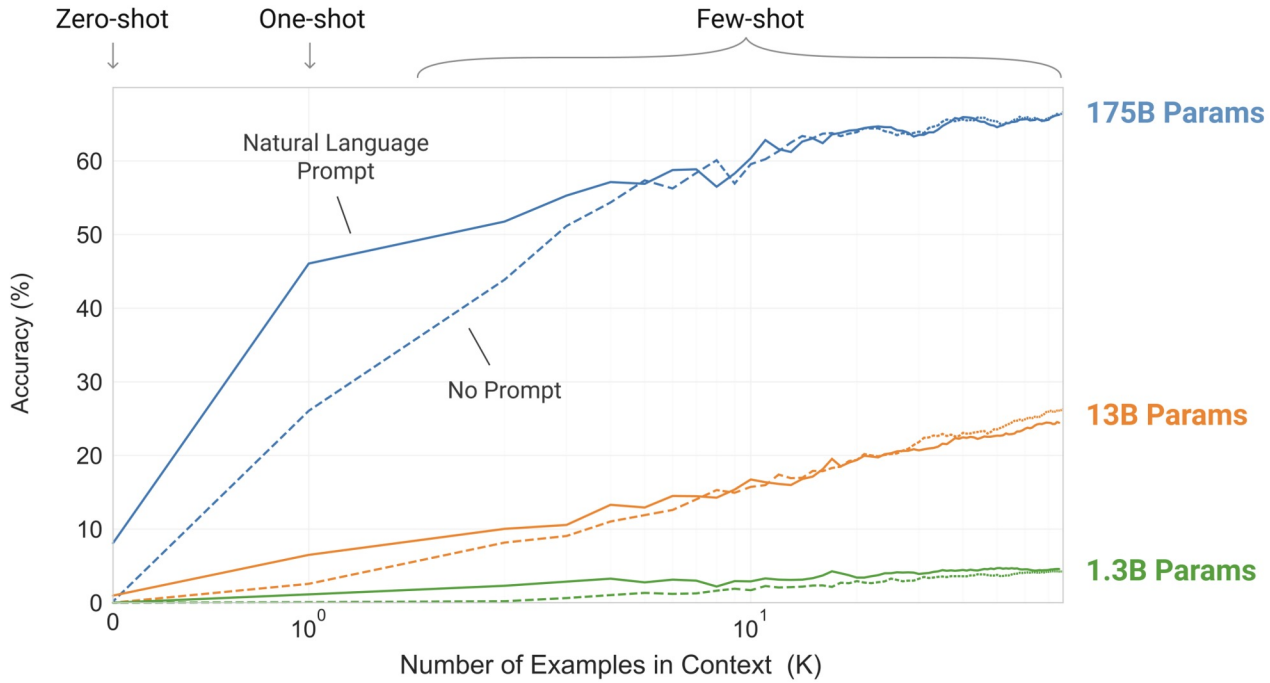
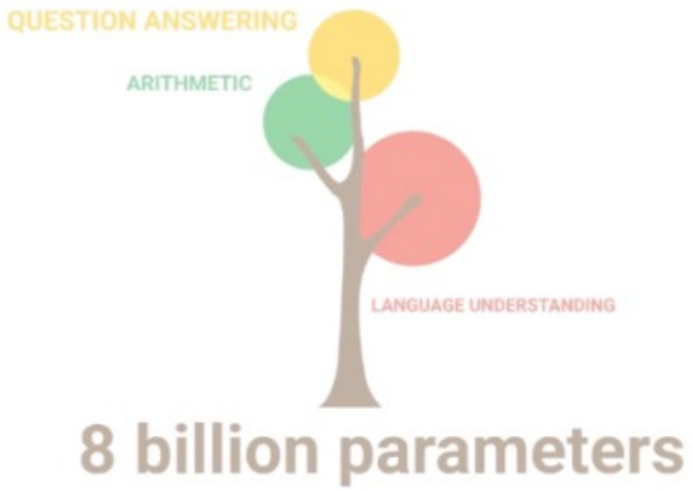


Figure 1.2: Larger models make increasingly efficient use of in-context information. We show in-context learning performance on a simple task requiring the model to remove random symbols from a word, both with and without a natural language task description (see Sec. 3.9.2). The steeper "in-context learning curves" for large models demonstrate improved ability to learn a task from contextual information. We see qualitatively similar behavior across a wide range of tasks.

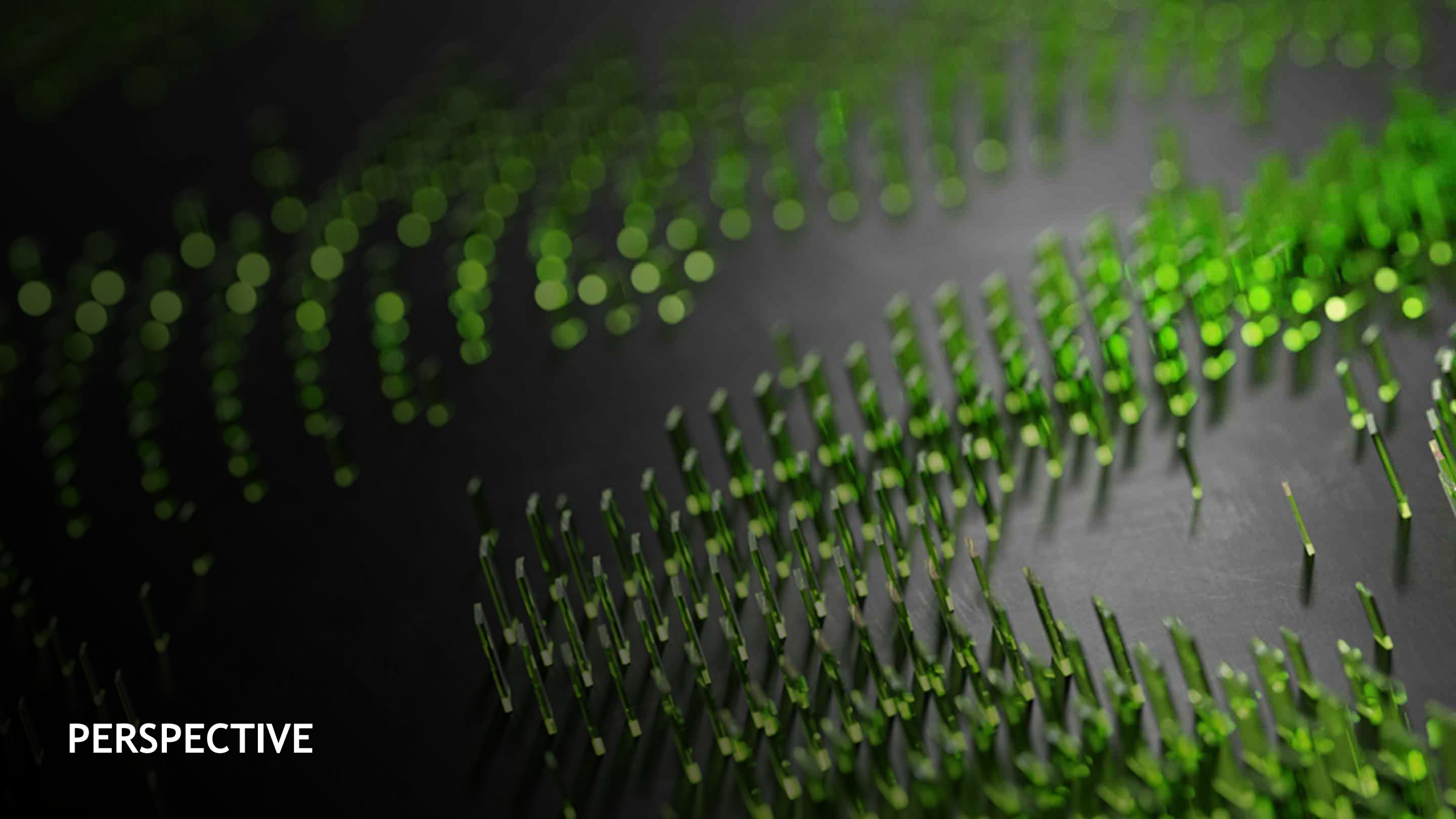
Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Agarwal, S. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165.*.

Model sizes vs tasks

LLM



https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html


PERSPECTIVE

WHAT DO I MEAN BY BIG

GPT-3 size comparison: 538x Bigger than BERT-Large

Not a linear scale

3.14*10^8 PFLOPS

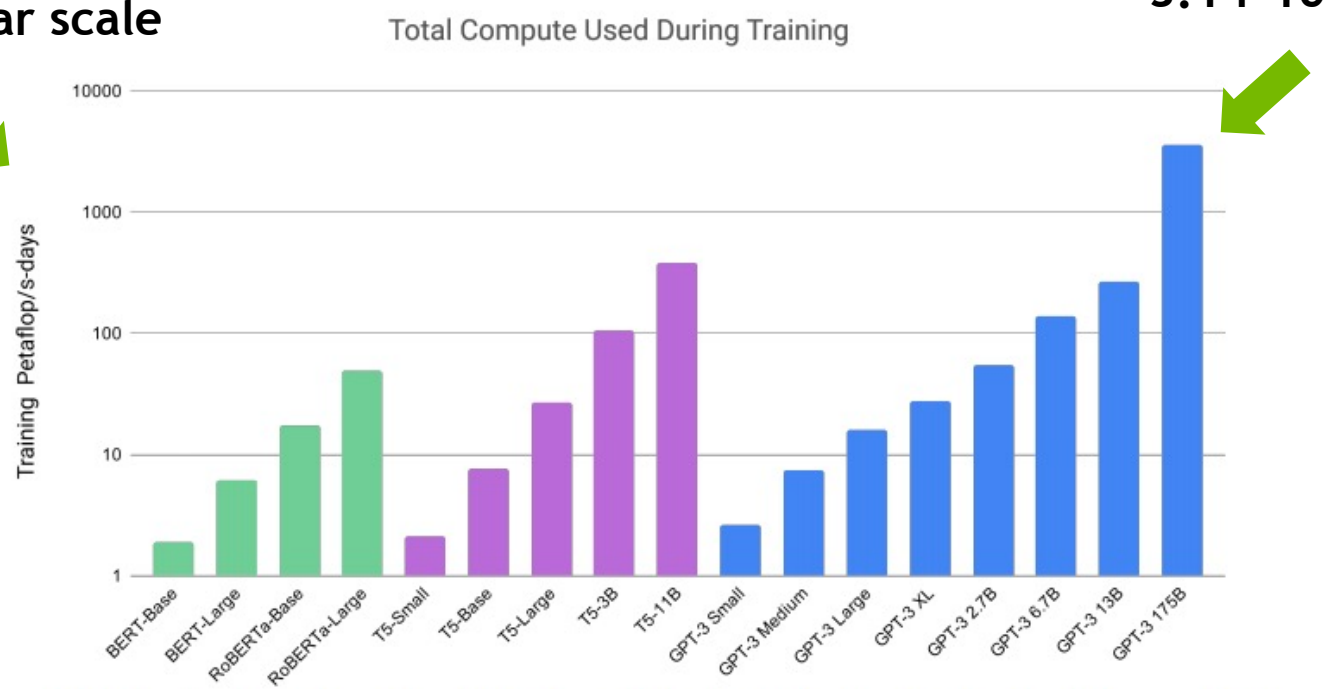


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

WHAT DO I MEAN BY BIG

GPT-3 size comparison: 538x Bigger than BERT-Large

Not a linear scale

~31 years on a single A100

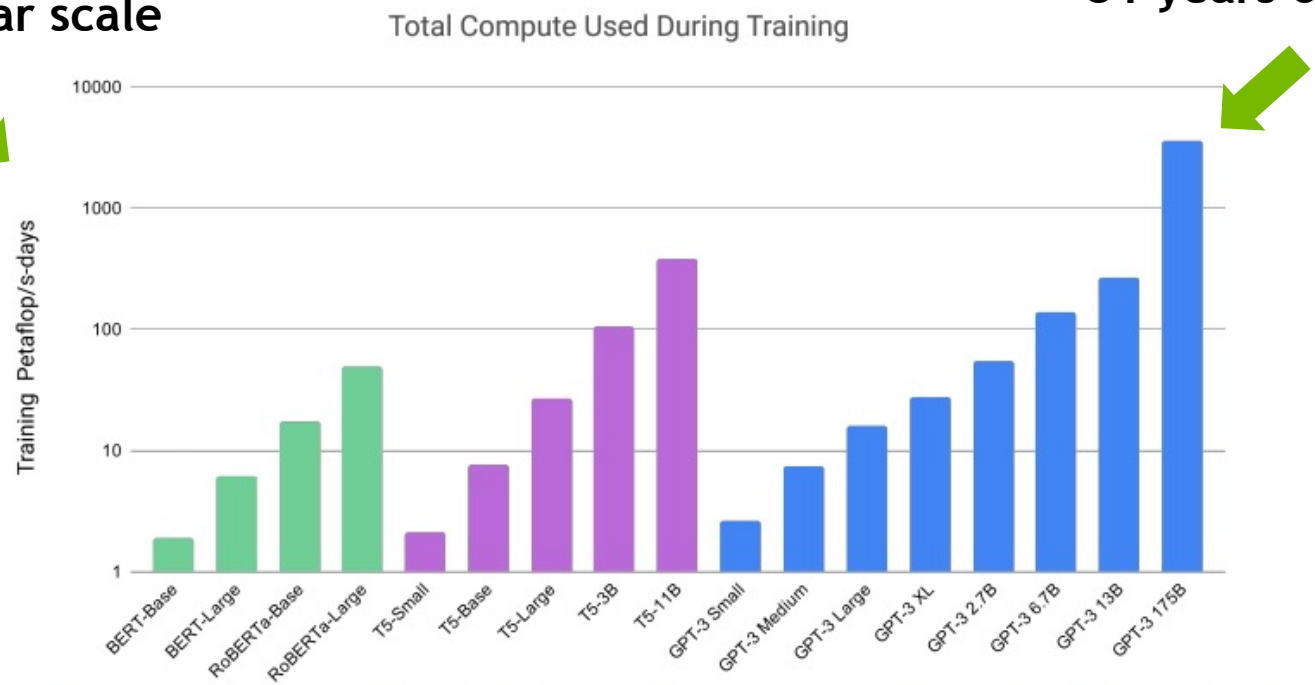


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

ESTIMATE COMPUTE NEEDED

Calculate how many hours/days compute resource need

paper : https://arxiv.org/pdf/2005.14165.pdf

[2]:
```
import numpy as np
T=300*1e+9 #oftokens in the dataset
#P=175*1e+9 # number of model parameters
n= 480 # Berzelius 480 # number of GPUs in the compute cluster

def calculate_days_needed(T , P , n ,x):
    if x is None:
        return '1-2 weeks'
    else:
        #x=140*1e+12 # TeraFlop/s per GPU
        tot=8*T*P
        div=n*x
        compute_sec=tot/div
        #convert compute seconds to days
        to_days=round(compute_sec/(3600*24),1)
        return to_days

GPT3_models_labels=[ 'gpt3_2.7B', 'gpt3_6.7B','gpt3_13B', 'gpt3_175B']
GPT3_model_params=[ 2.7*1e+9, 6.7*1e+9 , 13*1e+9, 175*1e+9,1*1e+12 ]
GPT3_model_params_str=['1.3 Billion' ,'2.7 Billion', '13 Billion', '175 Billion']
#according to the table above
GPT3_X=[127*1e+12, 130*1e+12,135*1e+12,140*1e+12 ]
print("all below are measured with dataset size **300 billion** measured in tokens \n")
for gpt3_name, gpt3_params, gpt3_param_str, x in zip(GPT3_models_labels,GPT3_model_params,G
    days_needed=calculate_days_needed(T,gpt3_params,n,x)
    print(" ----------------------------------------------------------------------------------
    print(" language model :{} with {} number of parameters , it will need {} days to compu
```

```
all below are measured with dataset size **300 billion** measured in tokens

 ----------------------------------------------------------------------------------
 language model :gpt3_2.7B with 1.3 Billion number of parameters , it will need 1.2 days to compute

 ----------------------------------------------------------------------------------
 language model :gpt3_6.7B with 2.7 Billion number of parameters , it will need 3.0 days to compute

 ----------------------------------------------------------------------------------
 language model :gpt3_13B with 13 Billion number of parameters , it will need 5.6 days to compute

 ----------------------------------------------------------------------------------
 language model :gpt3_175B with 175 Billion number of parameters , it will need 72.3 days to compute
```

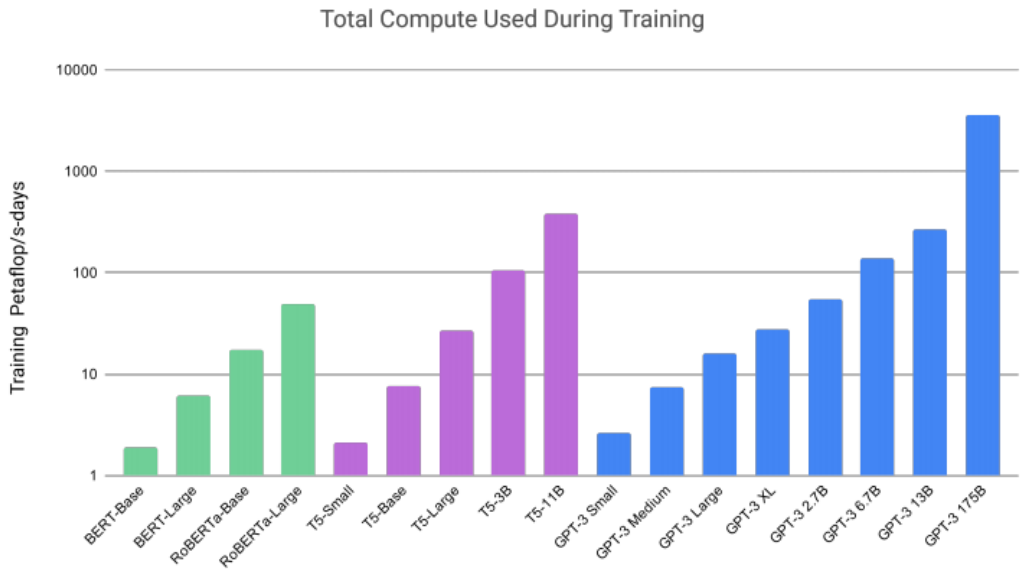


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

Source :https://arxiv.org/pdf/2005.14165.pdf

Scale of compute

Within reach of most companies

Model size	Attention heads	Hidden size	Number of layers	Number of parameters (billion)	Model-parallel size	Number of GPUs	Microbatch size	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7B	24	2304	24	1.7	1	32	16	512	137	44%	4.4
3.6B	32	3072	30	3.6	2	64	16	512	138	44%	8.8
7.5B	32	4096	36	7.5	4	128	16	512	142	46%	18.2
18B	48	6144	40	18.4	8	256	8	1024	135	43%	34.6
39B	64	8192	48	39.1	16	512	4	1536	138	44%	70.8
76B	80	10240	60	76.1	32	1024	2	1792	140	45%	143.8
145B	96	12288	80	145.6	64	1536	2	2304	148	47%	227.1
310B	128	16384	96	310.1	128	1920	1	2160	155	50%	297.4
530B	128	20480	105	529.6	280	2520	1	2520	163	52%	410.2
1T	160	25600	128	1008.0	512	3072	1	3072	163	52%	502.0

Weak scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

- ~6 weeks on 1 x DGX A100
- ~2 weeks on 4 x DGX A100

- ~65 weeks on 1 x DGX A100
- ~16 weeks on 4 x DGX A100

- ~5 years on 1 x DGX A100
- ~1 year on 4 x DGX A100

- ~69 years on 1 x DGX A100
- ~17 year on 4 x DGX A100

Narayanan, Deepak, et al. "**Efficient large-scale language model training on GPU clusters using megatron-LM**." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.

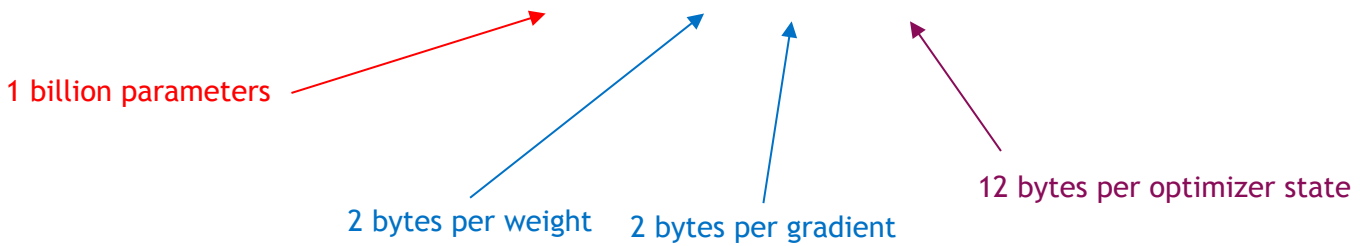
Going bigger

The challenge

Consider 1 billion parameters model in **FP16** and do the math:

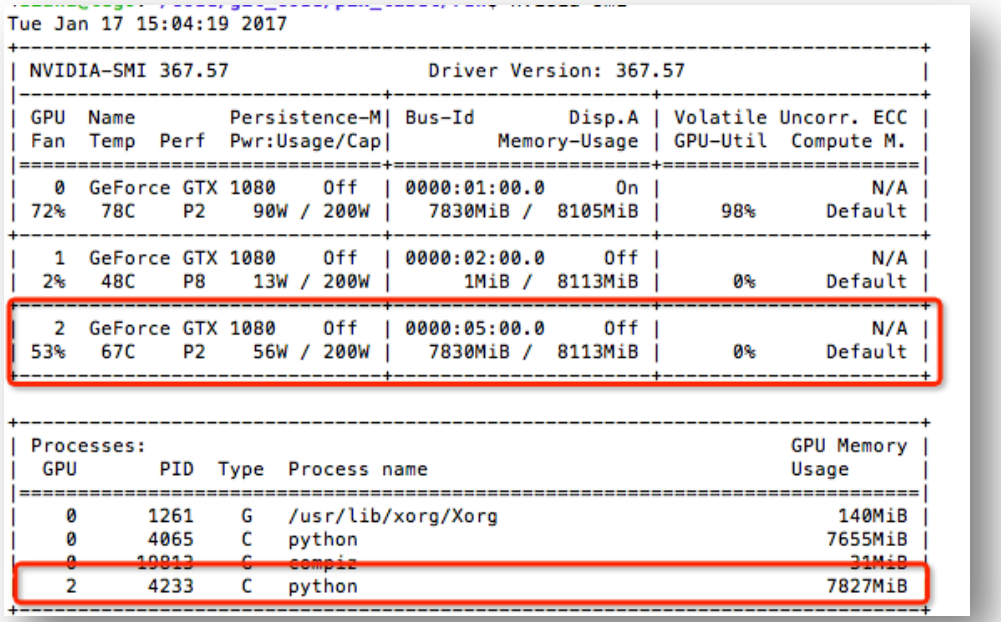
- Data representation: Weights and Gradients in FP16
- Adam optimizer: Store 12 bytes per weight in FP16

$$10^9 * (2B + 2B + 12B) = \mathbf{14.90GB}$$



GPU Memory occupation

1. model weights
2. optimizer states
3. gradients
4. forward activations saved for gradient computation
5. temporary buffers
6. functionality-specific memory



GPU Memory occupation

In details

- **Model Weights**
  - 4 bytes * number of parameters for fp32 training
  - 6 bytes * number of parameters for mixed precision training (maintains a model in fp32 and one in fp16 in memory)
- **Optimizer States**
  - 8 bytes * number of parameters for normal AdamW (maintains 2 states)
  - 2 bytes * number of parameters for 8-bit AdamW optimizers like bitsandbytes
  - 4 bytes * number of parameters for optimizers like SGD with momentum (maintains only 1 state)
- **Gradients**
  - 4 bytes * number of parameters for either fp32 or mixed precision training (gradients are always kept in fp32)
- **Forward Activations**
  - size depends on many factors, the key ones being sequence length, hidden size and batch size

3B-parameter model

T5-3b

- **AdamW** uses 8 bytes for each parameter, here the optimizer will need (8*3) **24GB** of GPU memory.
- **Adafactor** uses slightly more than 4 bytes, so (4*3) **12GB** and then some extra.
- **8bit BNB** quantized optimizer will use only (2*3) **6GB** if all optimizer states are quantized.
- A standard **Adam** uses 16 bytes for each parameter, so (8*3) **48GB** of GPU memory.


HOW CAN WE HANDLE THIS COMPLEXITY?

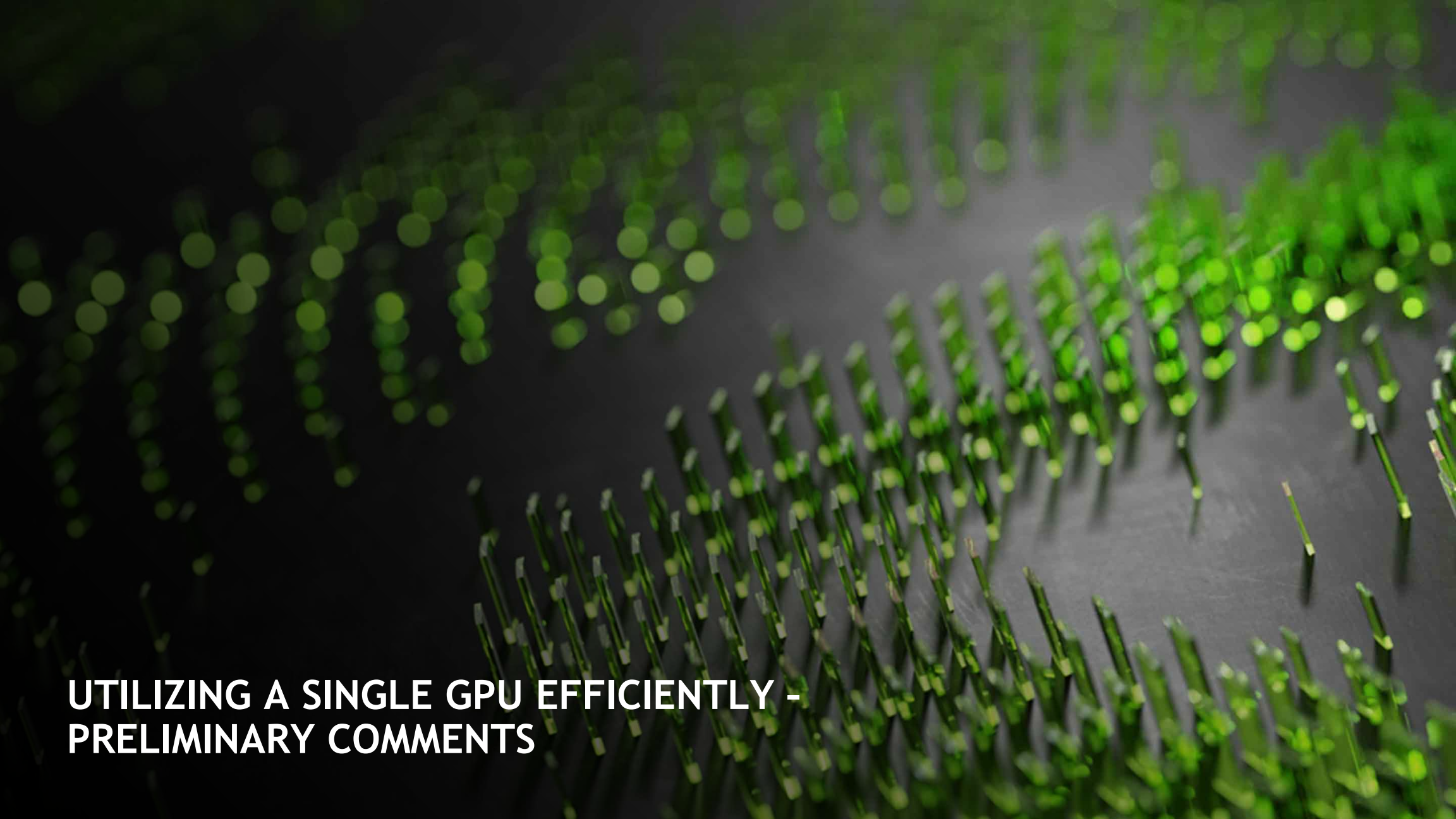
Objectives

- Fit very large models into limited hardware
  - e.g. t5-11b is 45GB in just model params
- Significantly speed up training
  - finish training that would take a year in hours

Optimization techniques

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

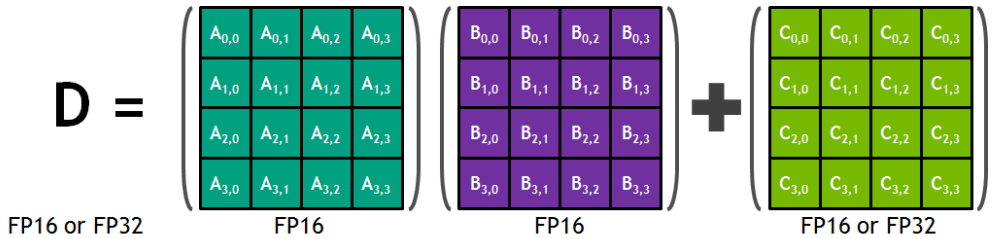
https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one

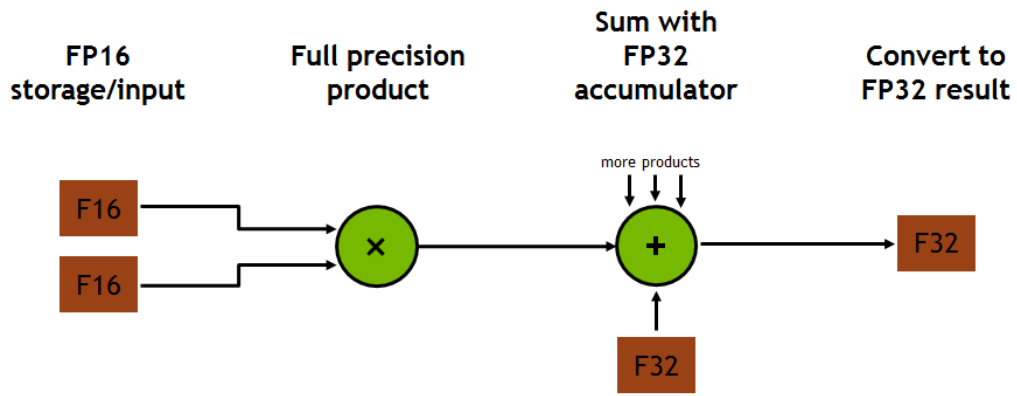


UTILIZING A SINGLE GPU EFFICIENTLY – PRELIMINARY COMMENTS

The Fundamental Operation of DEEP LEARNING
FUSED MULTIPLY ADD

- Fused matrix multiply and accumulate (FMA) operations are the core operations of deep learning training and inference.
- 1st generation Tensor Cores (V100) perform 64 floating point FMA mixed-precision operations per clock (FP16 input multiply with full-precision product and FP32 accumulate), i.e., 4 4x4 matrix tiles.
- Higher generation Tensor Cores support additional precisions.





Arithmetic Intensity

- The operation is said to be **compute-bound** or **data-bound** depending on which one finishes last, with the former scenario being preferable here.

- The threshold for a compute-bound operation is described through the concept of **arithmetic intensity** (ratio between the amount of computation and data transfer).

- A **NVIDIA A100** has a peak computational power of 312 teraflops for half-precision and a memory bandwidth of 2039 GB/s, for an arithmetic intensity threshold of **143 flops/B**.

- A binary addition with an arithmetic intensity of 1/6 lies deeply in the memory-bound region, while the multiplication of two 1024x1024 matrices has an arithmetic intensity of 341 and is compute-bound.

Transformers architecture

- **Tensor Contractions**
  - Linear layers and components of Multi-Head Attention all do batched matrix-matrix multiplications.
- **Statistical Normalizations**
  - Softmax and layer normalization are less compute-intensive than tensor contractions, and involve one or more reduction operations.
- **Element-wise Operators**
  - Biases, dropout, activations, and residual connections.

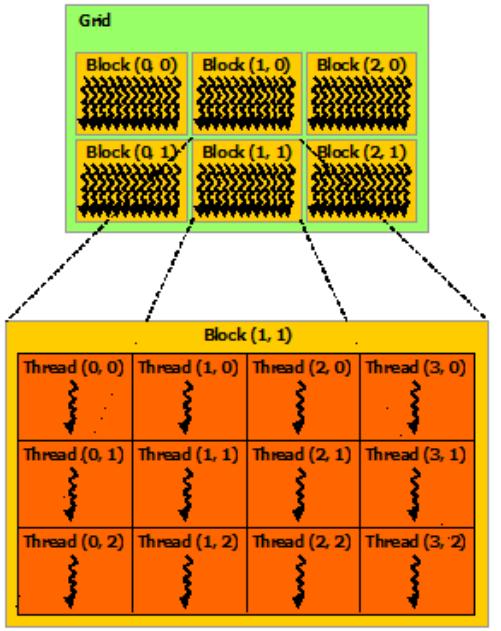
Table 1. Proportions for operator classes in PyTorch.

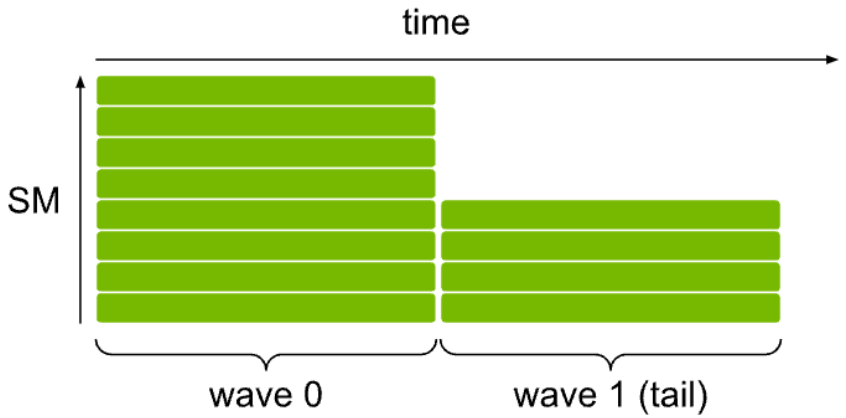
Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

DATA MOVEMENT IS ALL YOU NEED: A CASE STUDY ON OPTIMIZING TRANSFORMERS https://arxiv.org/pdf/2007.00072.pdf

Simplified EXECUTION MODEL

- N *Kernels* are executed in parallel by N different *CUDA threads*.
- Threads are arranged a one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block*. A set of thread blocks are launched to execute a function.
- It is usually better the overcommit w.r.t. the number of threads to facilitate instruction latency.
- When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution.
- It is important to avoid *warp divergence* (“thread blocks of different size”) whenever possible!
- A set of thread blocks running concurrently is called a *wave*. The more waves, the better to minimize tail effects.

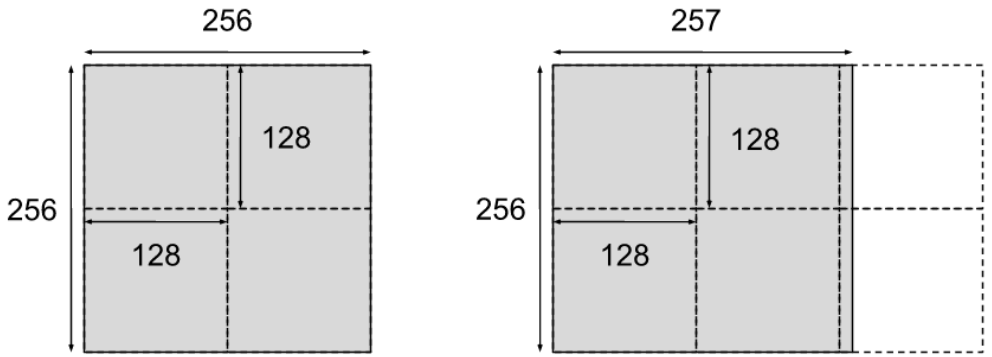


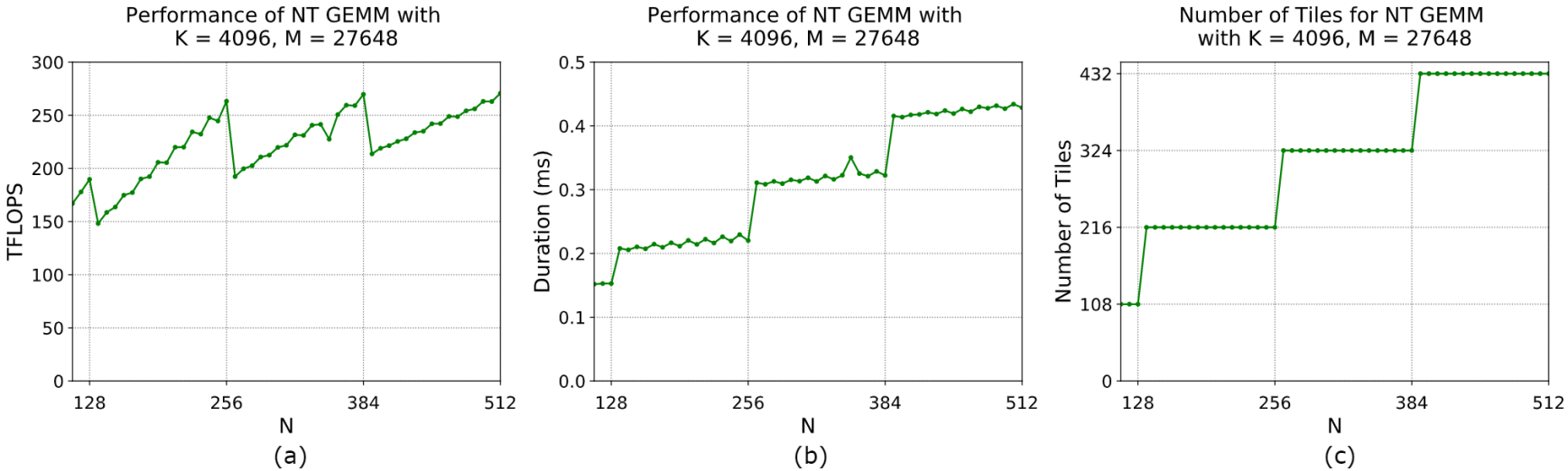


A simple example

GEMMs

- Tile quantization effect on (a) achieved FLOPS throughput and (b) elapsed time, alongside (c) the number of tiles created.
- Measured with a function that forces the use of 256x128 tiles over the MxN output matrix. In practice, cuBLAS would select narrower tiles (for example, 64-wide) to reduce the quantization effect.
- Experiment performed on NVIDIA A100-SXM4-80GB, CUDA 11.2, cuBLAS 11.4.

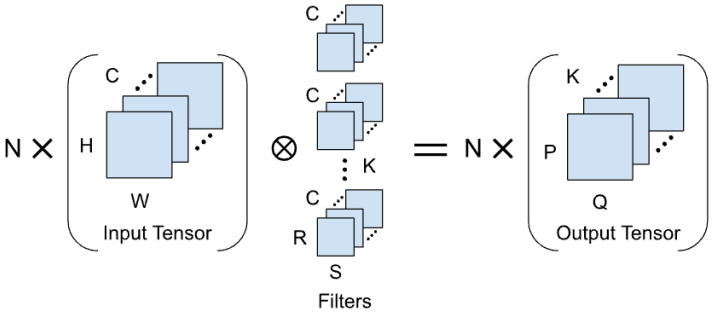




Checklist for CONVOLUTIONAL LAYERS

https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html#checklist

- Choose the number of input and output channels to be divisible by 8 (for FP16) or 4 (for TF32). Also consider padding the input channels.
- Choose parameters (batch size, number of input and output channels) to be divisible by at least 64 and ideally 256 to enable efficient tiling and reduce overhead.
- Larger values for size-related parameters (batch size, input and output height and width, and the number of input and output channels) can improve parallelization and hence increase efficiency.
- Make sure auto-tuning is enabled, if applicable.
- Choose tensor layouts in memory to avoid transposing input and output data. We recommend using the NHWC format where possible.



For specific guidance on 2D and particularly 3D convolutions, please also refer to https://docs.nvidia.com/deeplearning/cudnn/best-practices/index.html

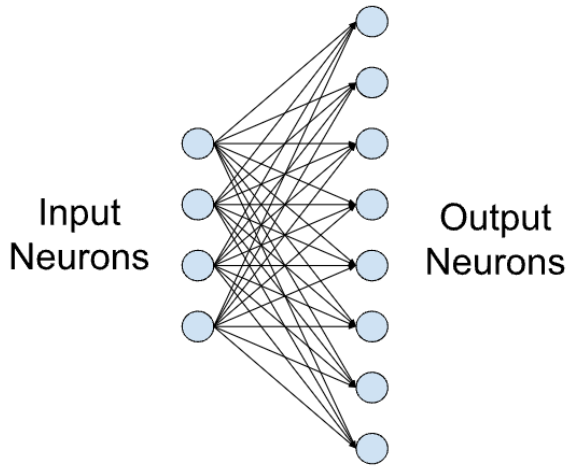
Checklist for FuLLY CONNECTED LAYERS

https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#checklist

- Choose the batch size and the number of inputs and outputs to be divisible by 4 (TF32) / 8 (FP16) / 16 (INT8) to run efficiently on Tensor Cores. For best efficiency on A100, choose these parameters to be divisible by 32 (TF32) / 64 (FP16) / 128 (INT8).

- Especially when one or more parameters are small, choosing the batch size and the number of inputs and outputs to be divisible by at least 64 and ideally 256 can streamline tiling and reduce overhead.
→ Larger values for batch size and the number of inputs and outputs improve parallelization and efficiency.

- As a rough guideline, choose batch sizes and neuron counts greater than 128 to avoid being limited by memory bandwidth (NVIDIA A100-SXM4-80GB; this threshold is similar for other A100 and V100 GPUs).



Important cuDNN flags

https://pytorch.org/docs/stable/backends.html#torch-backends-cudnn

Important aspects to consider:

- In NGC containers, the usage of TensorFloat-32 is enabled by default in order to accelerate FP32 calculations using tensor cores on Ampere or newer GPUs.
- Certain classes of CUDA functions are a potential source of non-determinism, such as atomicAdd, where the order of parallel additions to the same value is undetermined and, for floating-point variables, a source of variance in the results.
- cuDNN can automatically determine which combination of primitives is most optimal. Only use this flag when input sizes of a model are no changing!

```
# get the cuDNN version
torch.backends.cudnn.version()

# check availability
torch.backends.cudnn.is_available()

# enabling cuDNN (default = True)
torch.backends.cudnn.enabled = True

# enabling TF32 (default = True for DL)
torch.backends.cudnn.allow_tf32 = True

# enable determinism (default = False)
torch.backends.cudnn.deterministic = False

# enable auto-tuning (default = False)
torch.backends.cudnn.benchmark = True
```

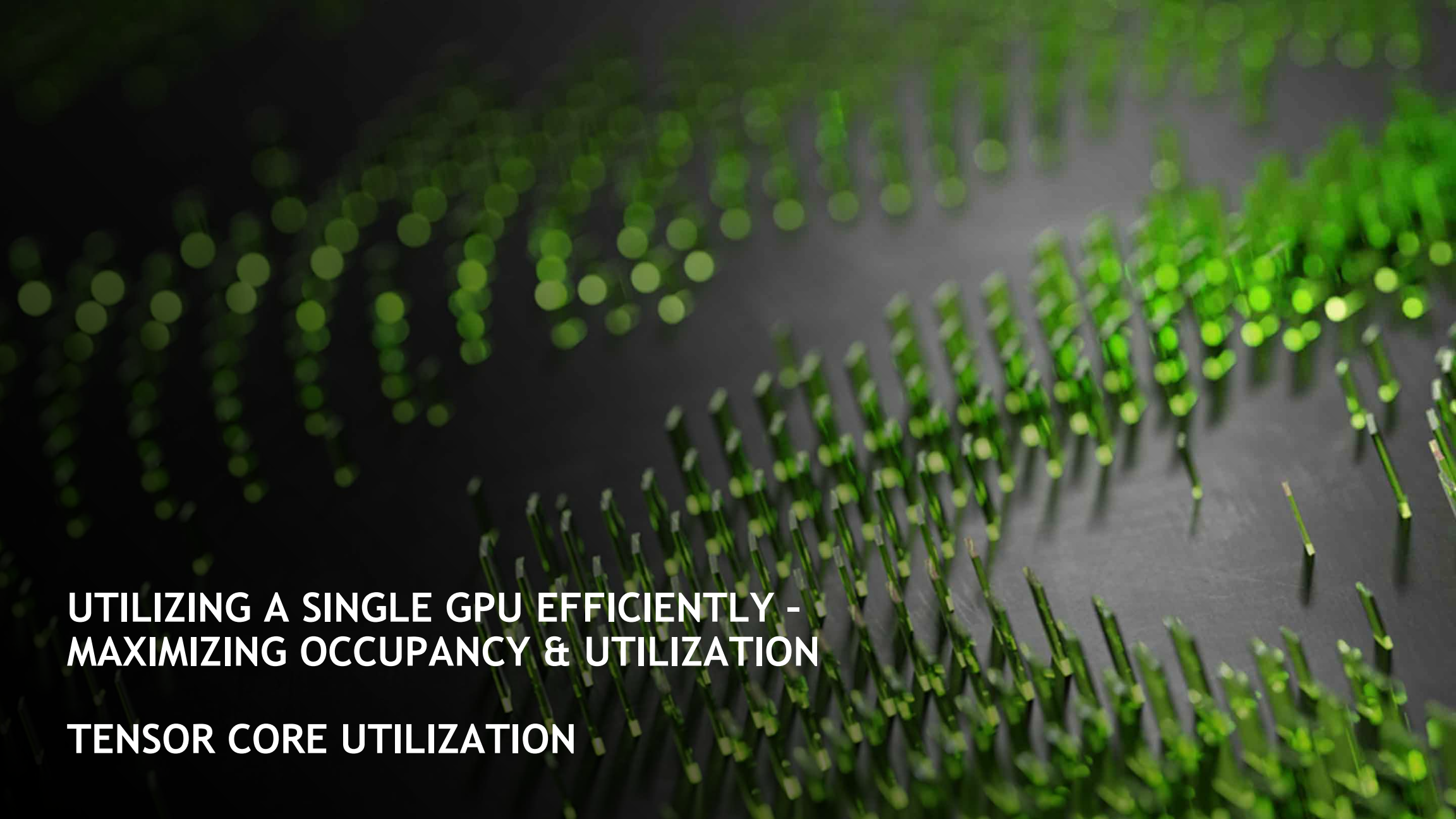
A Note on Time Measurements

https://pytorch.org/docs/stable/backends.html#torch-backends-cudnn

Important aspects to consider:

- Be careful with measuring time on the host
- CUDA events are synchronization markers that can be used to monitor the device's progress, to accurately measure timing, and to synchronize CUDA streams.
- Make sure you are measuring large enough workloads.
- Always perform multiple repetitions and average the results.
- Never measure the 1st API call and perform GPU warmup.

```
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)

start.record()
# code to be measured
…
end.record()

torch.cuda.synchronize()

elapsed_time_in_ms = start.elapsed_time(end)
```


UTILIZING A SINGLE GPU EFFICIENTLY –
MAXIMIZING OCCUPANCY & UTILIZATION
TENSOR CORE UTILIZATION

TENSOR CORE CAPABILITIES

FMA operations per clock per SM

	CUDA Cores				Tensor Cores				
NVIDIA Arch.	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4
Volta	32	64	128	256			512		
Turing	2	64	128	256			512	1024	8192
Ampere (A100)	32	64	256	256	64	512	1024	4096	16384
Ampere, sparse						1024	2048	4096	8192

Example - calculate TF32 throughput for A100:

108 (SMs) x 512 (multiply-add ops) x 2 (floating point ops) x 1.41 GHz (clock rate) = 156 TeraFLOPS

MIXED PRECISION TRAINING - THE IDEA

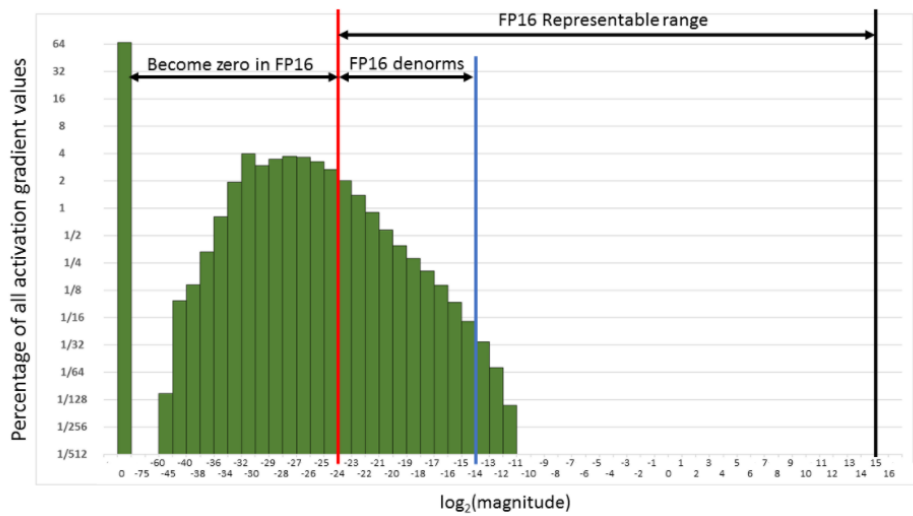
Example: FP32 training of Multibox SSD network

- 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!

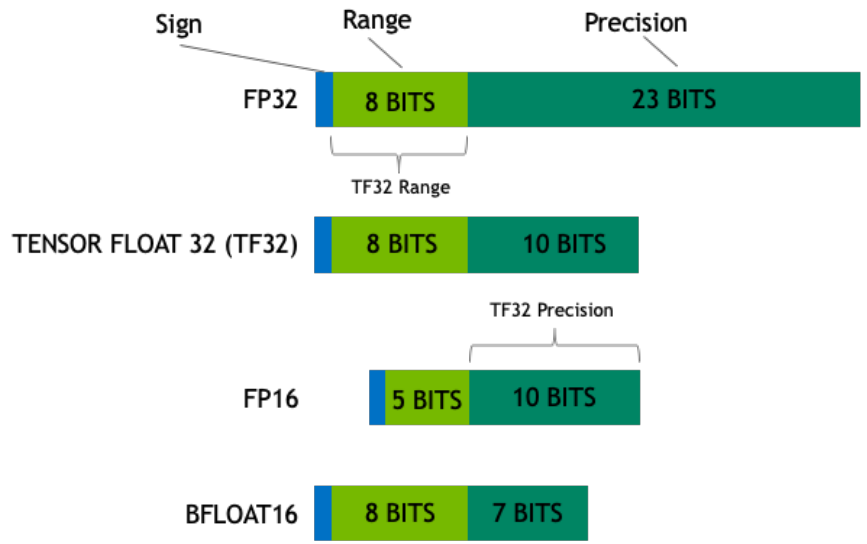
- Concern: Do I need to run a full training in order to find the scaling factor?
  → No, automatic mixed precision comes to the rescue! ☺



A NOTE ON DATA TYPES

Or why TF32 makes sense

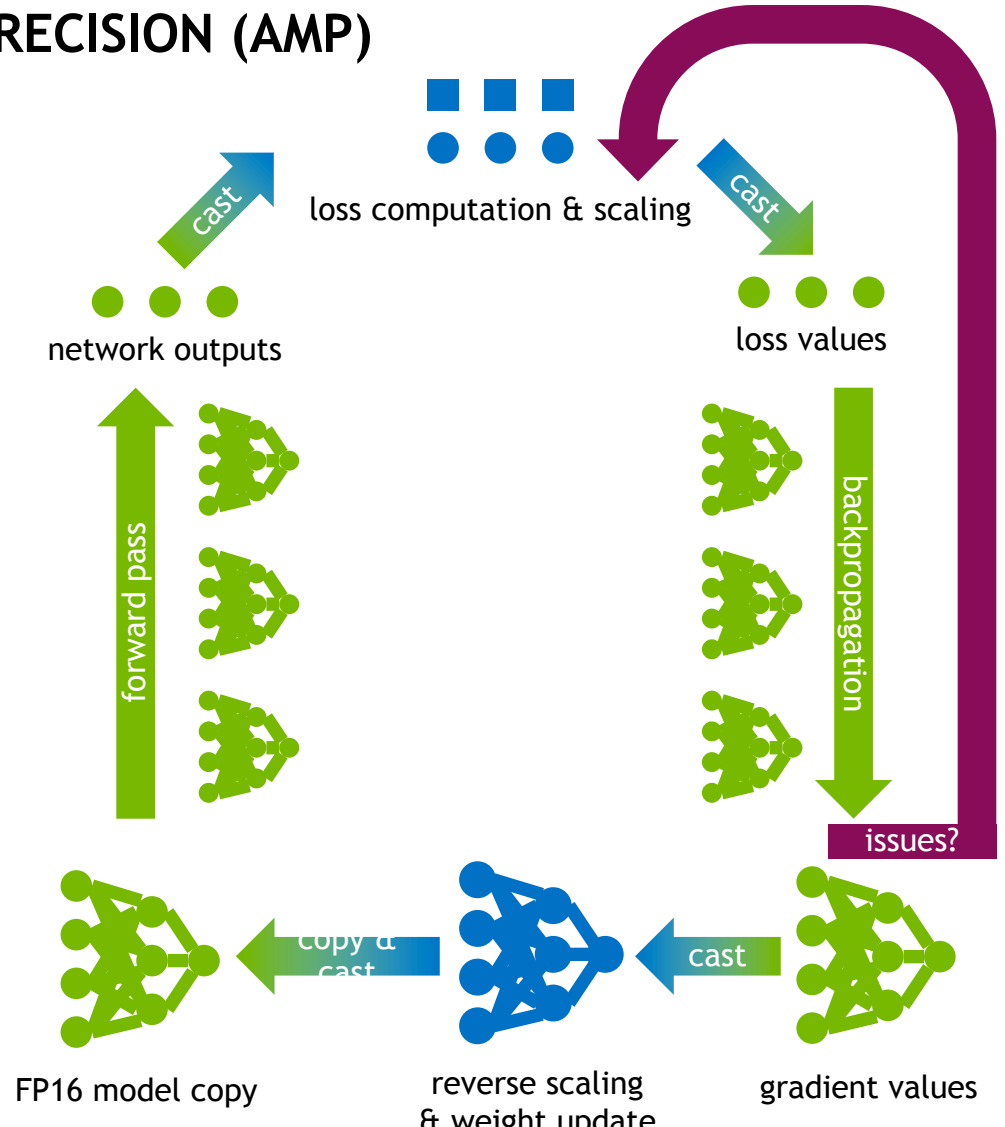
- Mixed precision training is mostly about the dynamic range and less about the precision:
  - exponent → dynamic range
  - significand field → precision
- TF32 is a great compromise between FP32 (same range) and FP16 (same precision)
- TF32 is automatically enabled in NGC containers
- No code change is necessary!



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

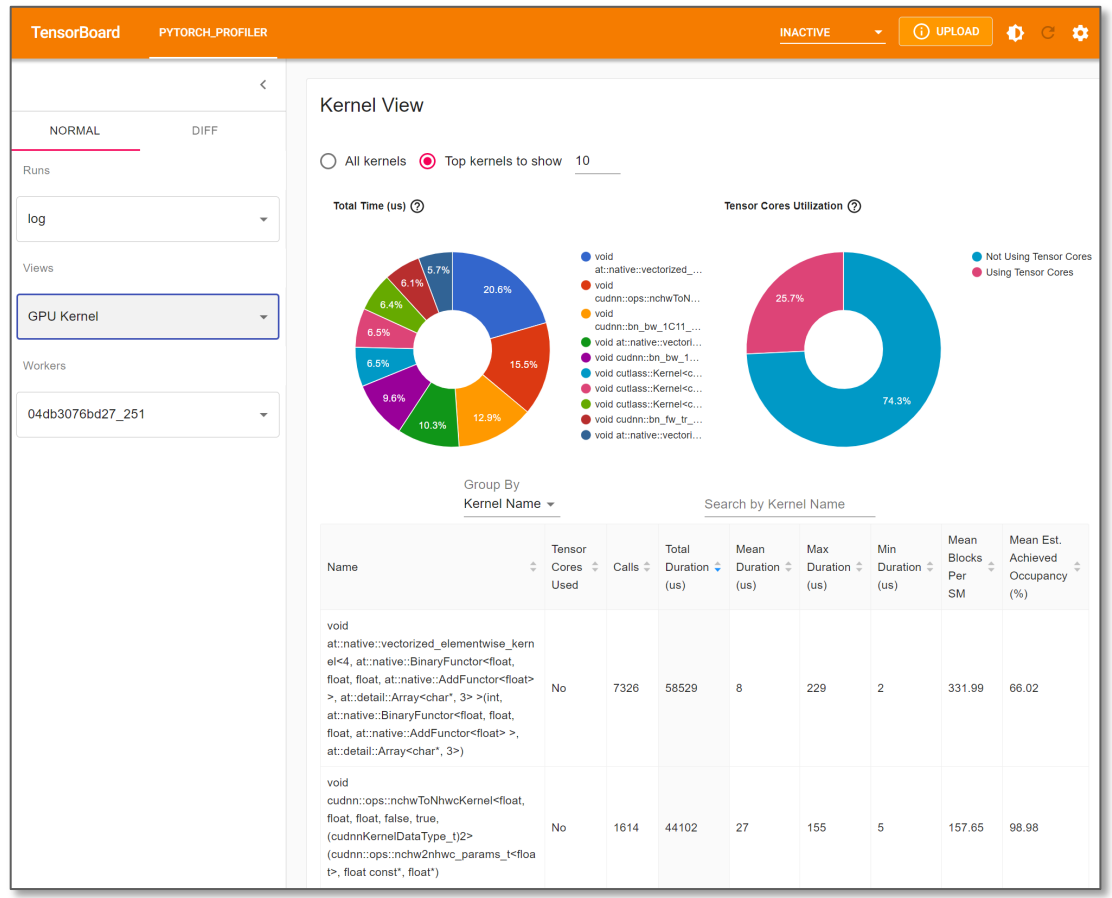
- Backward passes under autocast are not recommended.
- Backward ops run in the same dtype autocast chose for corresponding forward ops.
- scaler.step() first unscales the gradients of the optimizer's assigned params.
- If these gradients contain infs or NaNs, optimizer.step() is skipped.

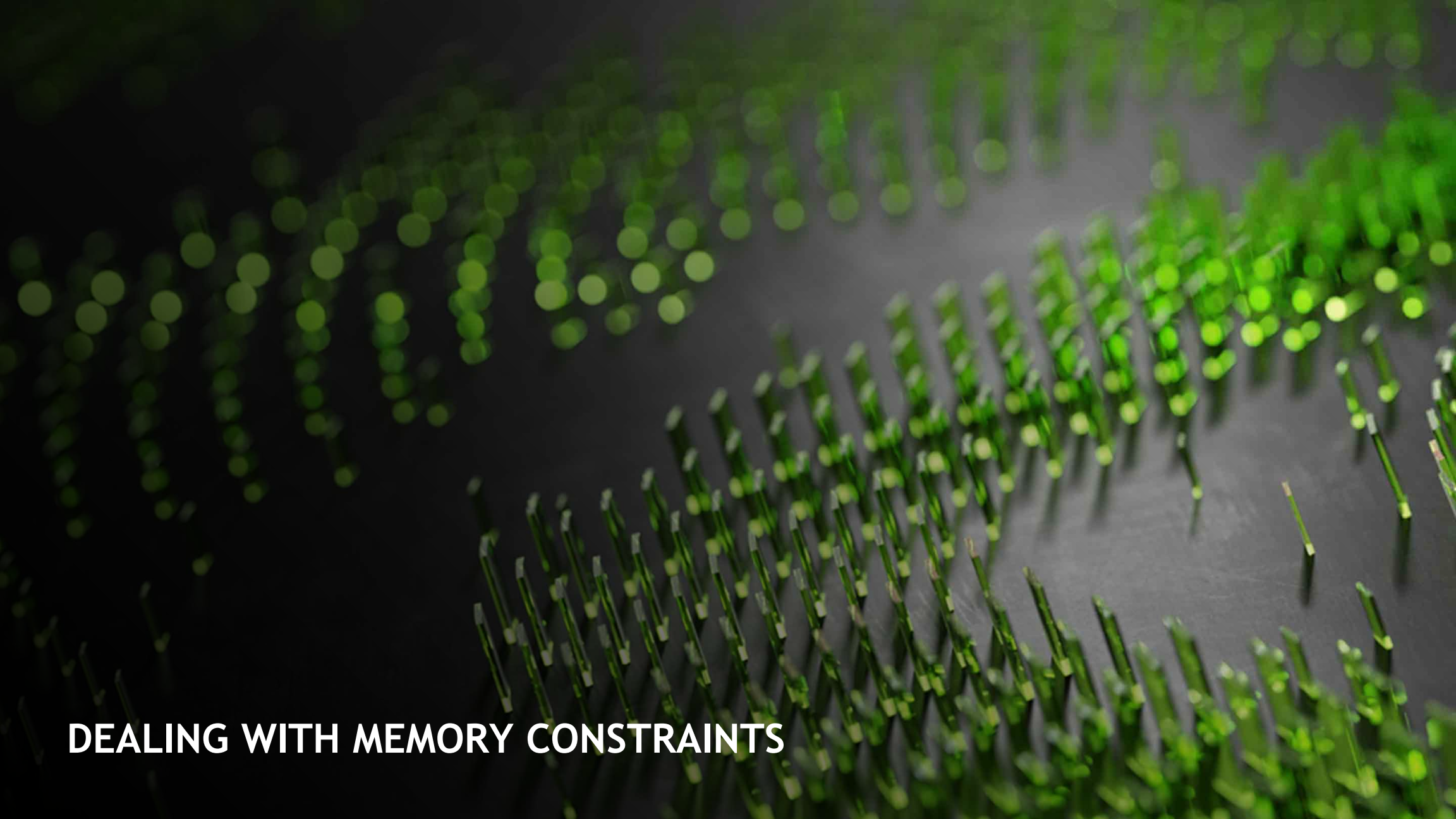
```
# initialize gradient scaler
scaler = GradScaler()

# 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()
```

AM I USING TENSOR CORES?

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

```
from torch import profiler

prof_schedule = profiler.schedule(wait=2,
                                  warmup=2,
                                  active=5,
                                  repeat=0)

callback =  profiler.tensorboard_trace_handler('./log')

prof = profiler.profile(schedule=prof_schedule,
on_trace_ready=callback,
record_shapes=False,
with_stack=False)

prof.start()

for it in range(num_iterations):
    # code to be profiled
    ...
    prof.step()

prof.stop()
```

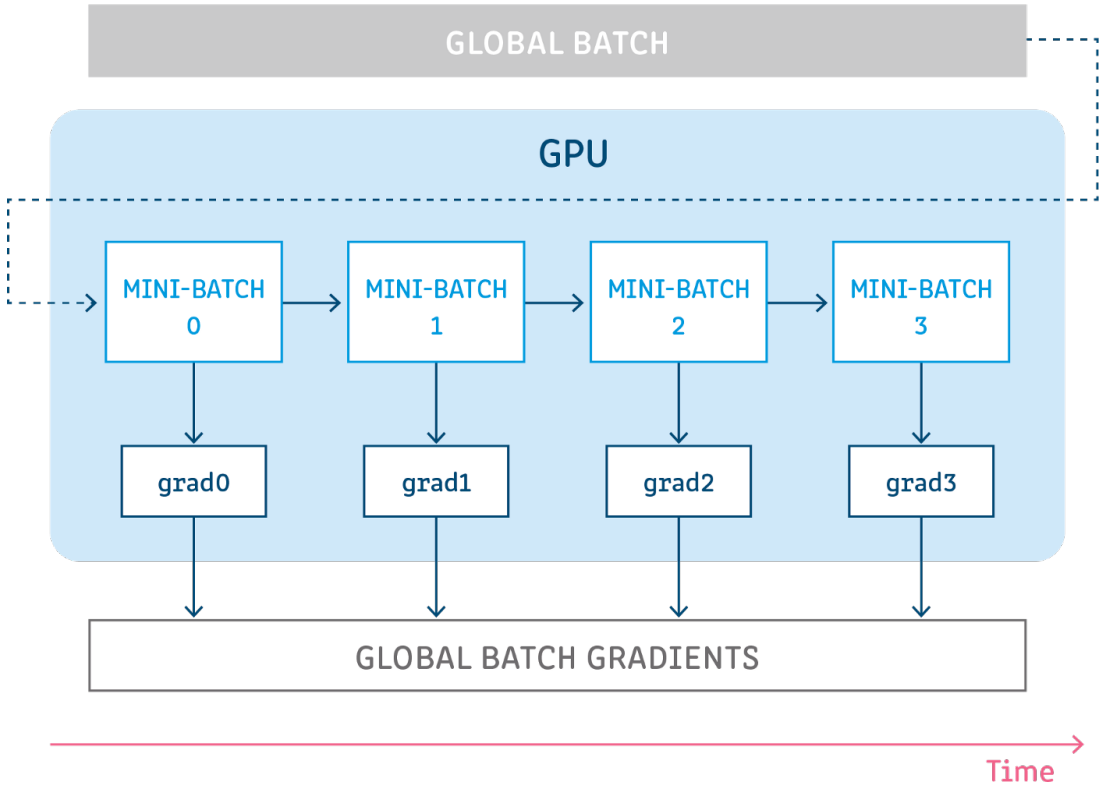




DEALING WITH MEMORY CONSTRAINTS

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

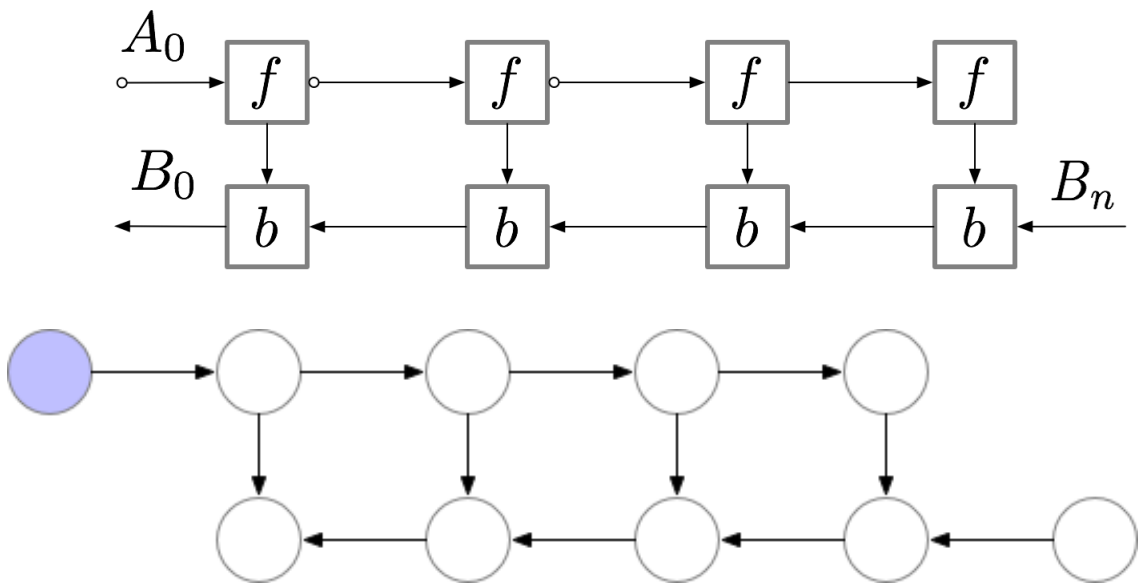
```
optimizer = ...


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

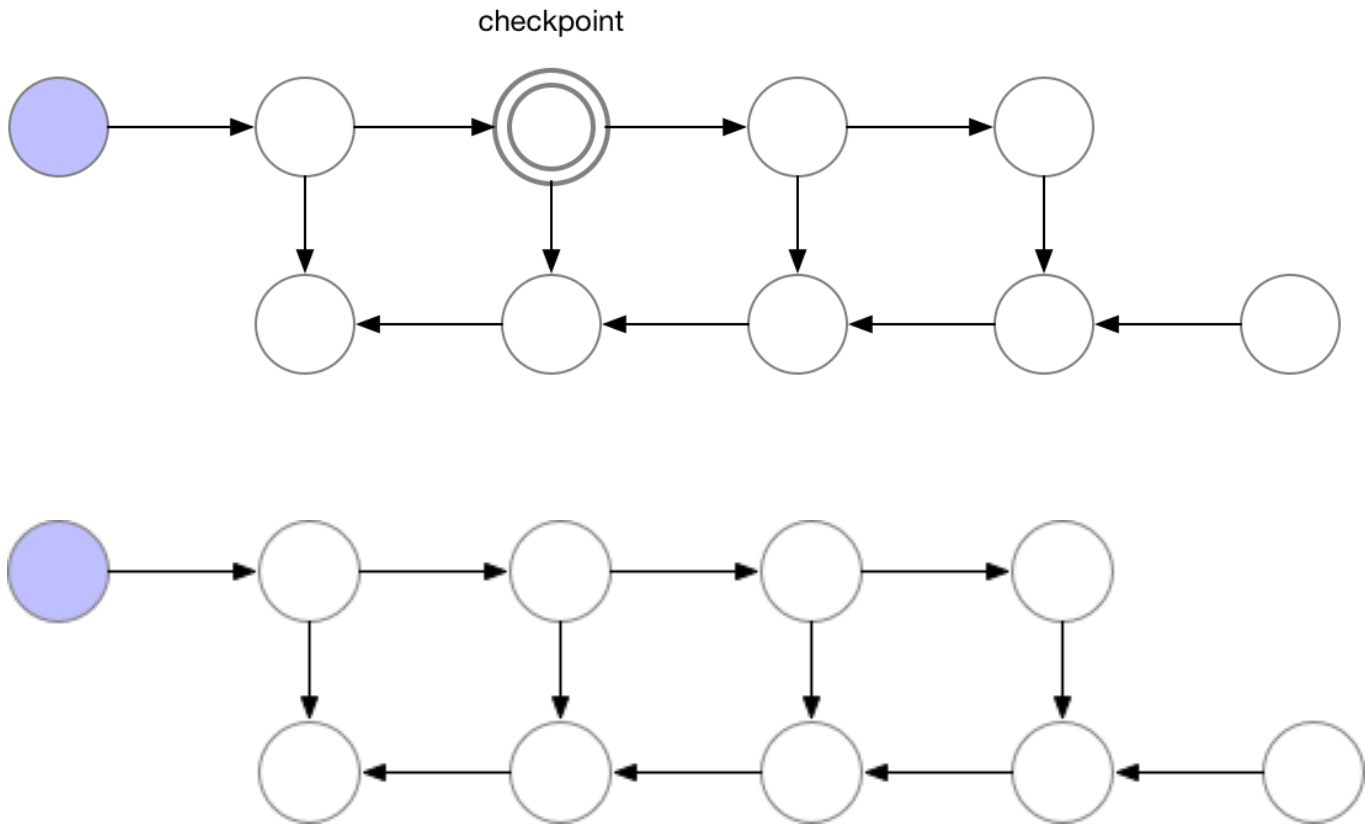
https://pytorch.org/docs/stable/checkpoint.html

- 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.

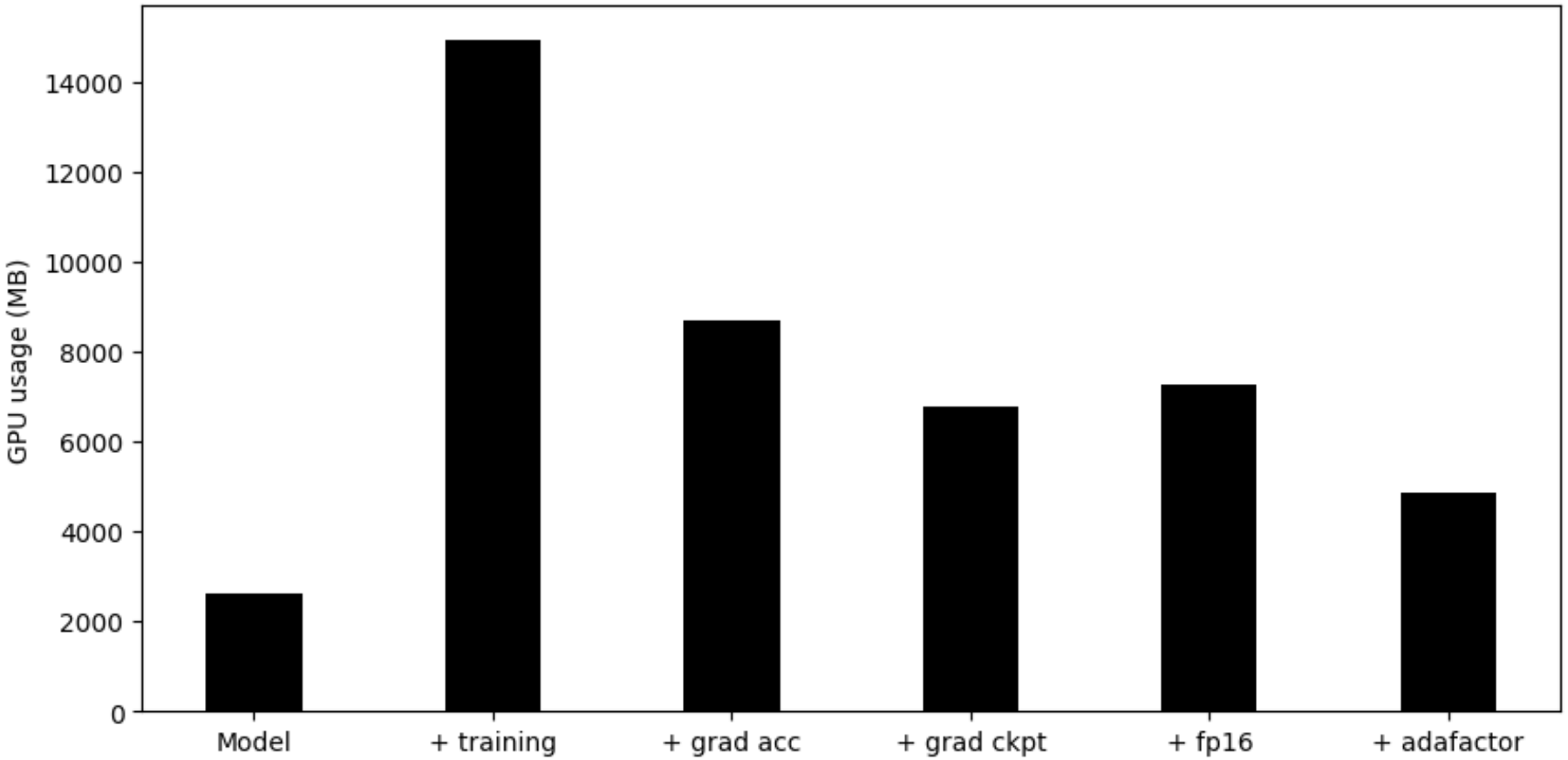


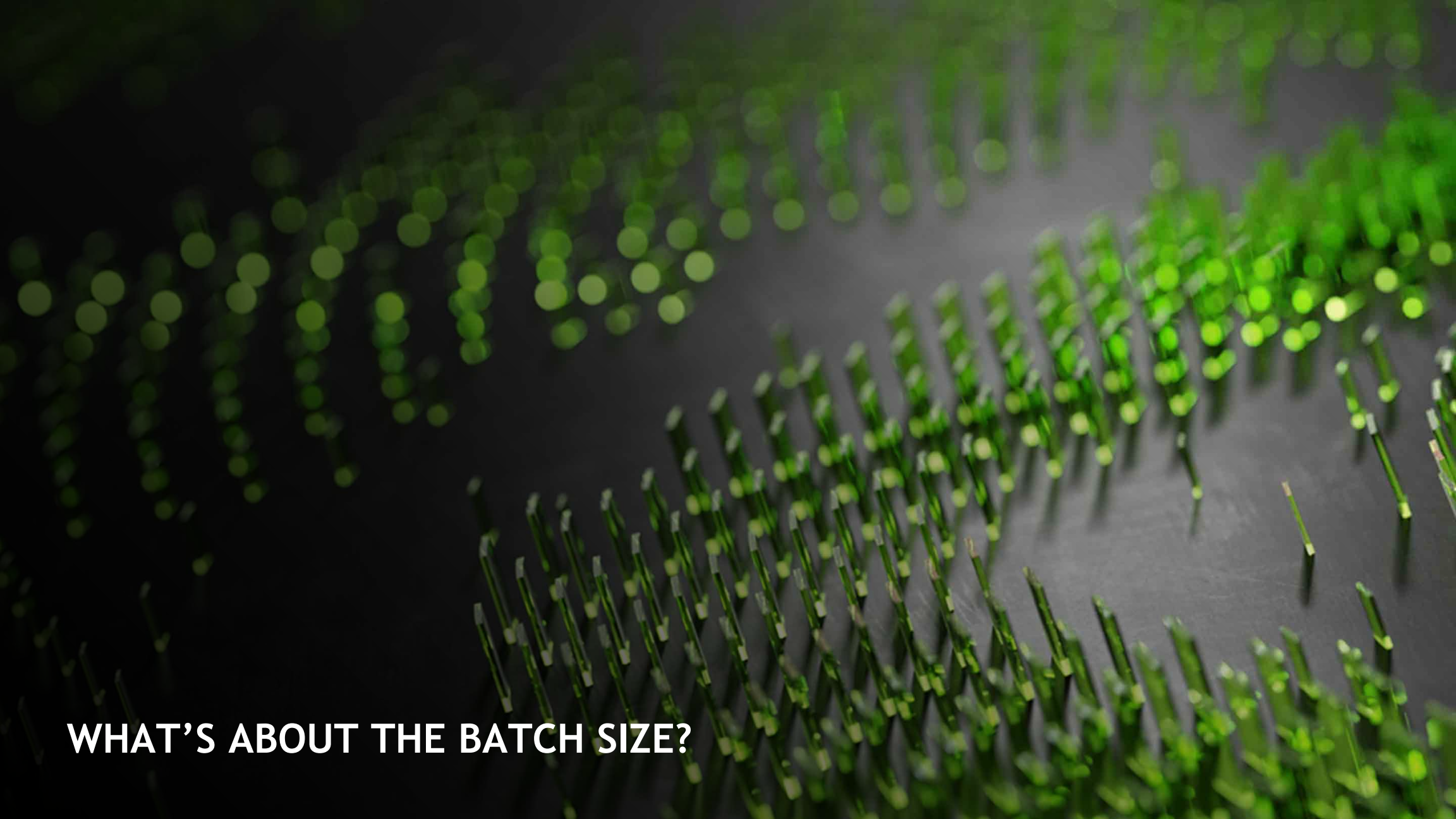
https://github.com/cybertronai/gradient-checkpointing

Activation Re-computation or gradient checkpointing



GPU Memory usage





WHAT'S ABOUT THE BATCH SIZE?

Why Batch size matters

- The most efficient performance when batch sizes and input/output neuron counts are divisible by a certain number, which typically starts at **8**, but can be much higher as well.
- That number varies a lot depending on the specific hardware being used and the dtype of the model:
  - *Tensor Core Requirements define the multiplier based on the dtype and the hardware. For example, for fp16 a multiple of **8** is recommended, but on A100 it's **64**!*
- When parameters are too small, there is also Dimension Quantization Effects to consider, this is where tiling happens, and the right multiplier can have a significant speedup.
- Furthermore, the bigger the batch size the less often the optimizer is run, the faster the training is (considering the same dataset length).

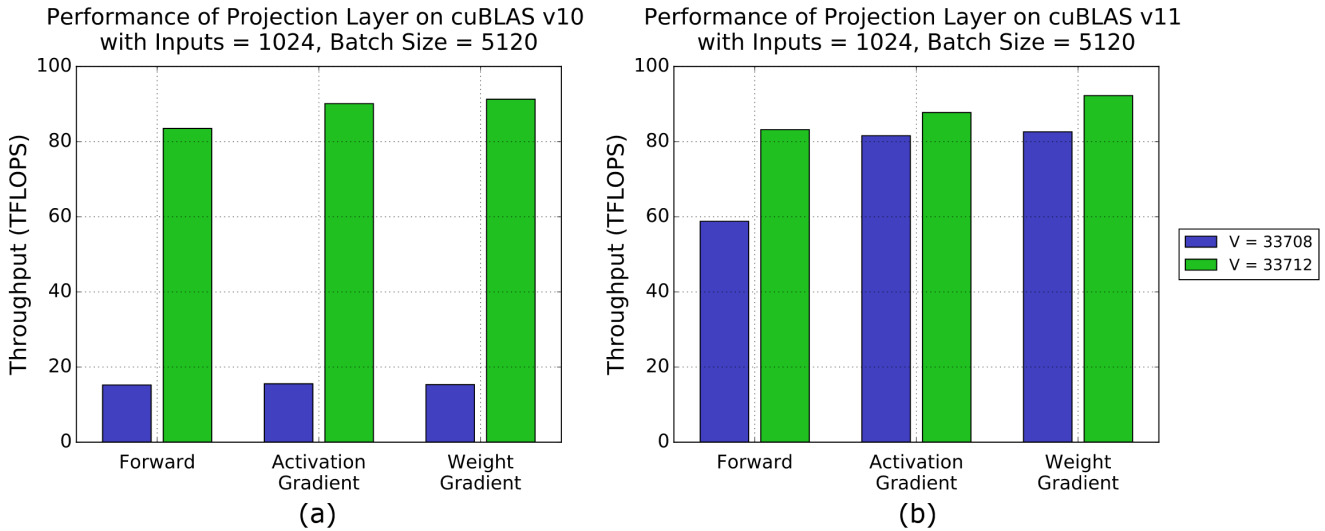
Batch size checklist

fully-connected layers

1. Choose the batch size and the number of inputs and outputs to be divisible by 4 (TF32) / 8 (FP16) / 16 (INT8) to run efficiently on Tensor Cores.
2. For best efficiency on A100, choose these parameters to be divisible by 32 (TF32) / 64 (FP16) / 128 (INT8).
3. Especially when one or more parameters are small, choosing the batch size and the number of inputs and outputs to be divisible by at least 64 and ideally 256 can streamline tiling and reduce overhead.
4. Larger values for batch size and the number of inputs and outputs improve parallelization and efficiency.
5. **As a rough guideline, choose batch sizes and neuron counts greater than 128 to avoid being limited by memory bandwidth.**

The case of transformer

- Transformers are a popular neural network architecture used for sequence-to-sequence mapping tasks, for example for natural language translation. They use an encoder-decoder architecture making heavy use of attention, both to “self-attend” over input sequences, as well as to give the decoder access to the encoder’s context
- From a performance standpoint, Transformers fundamentally process all the tokens in an input sequence in parallel. That makes Transformers very amenable to highly parallel architectures such as GPUs, and leads to large GEMMs that, with a few simple guidelines, can take great advantage of Tensor Core acceleration.

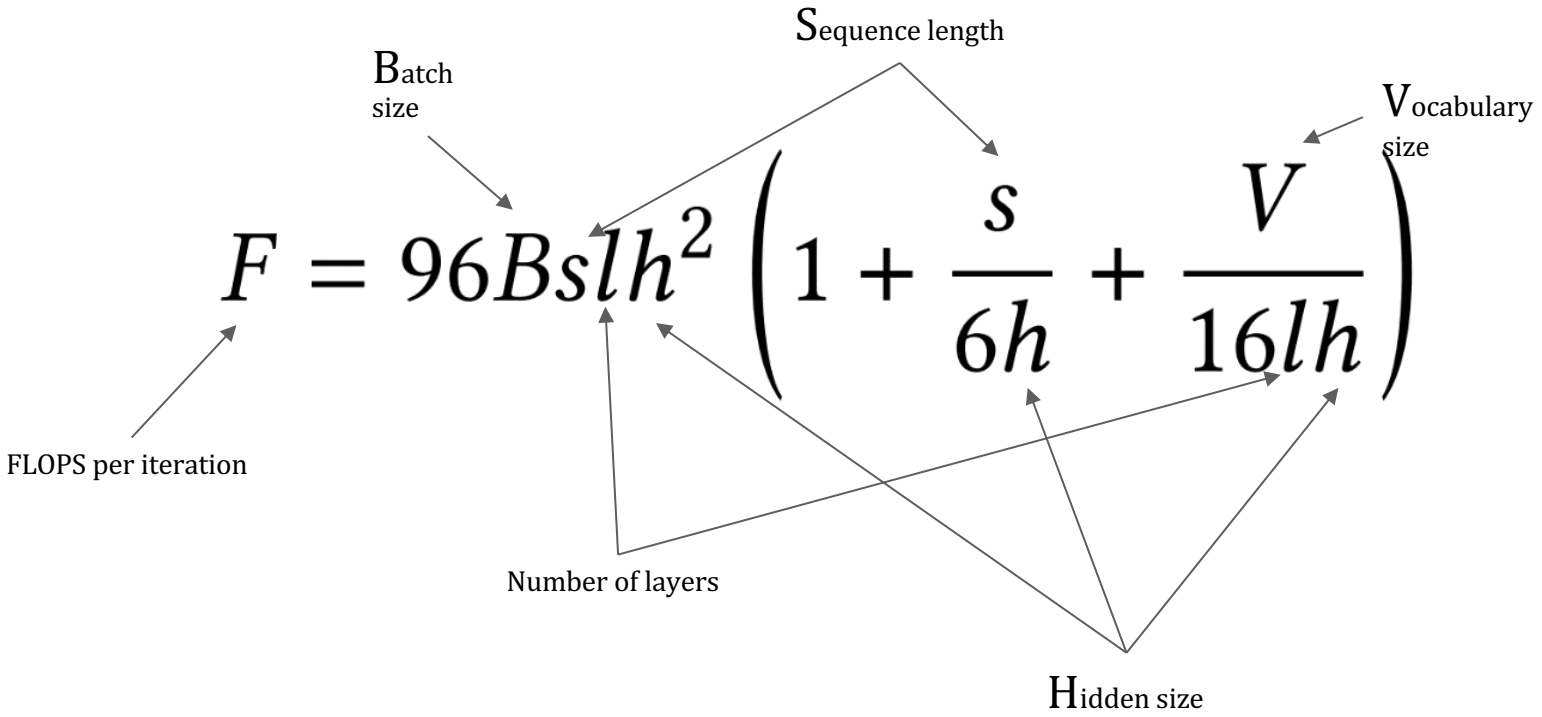


See what happens when the vocabulary size is chosen without regard to alignment. FP16 data is used, so dimensions must be multiples of 8 for best alignment.

PARALLELISM & COLLECTIVE COMMUNICATIONS

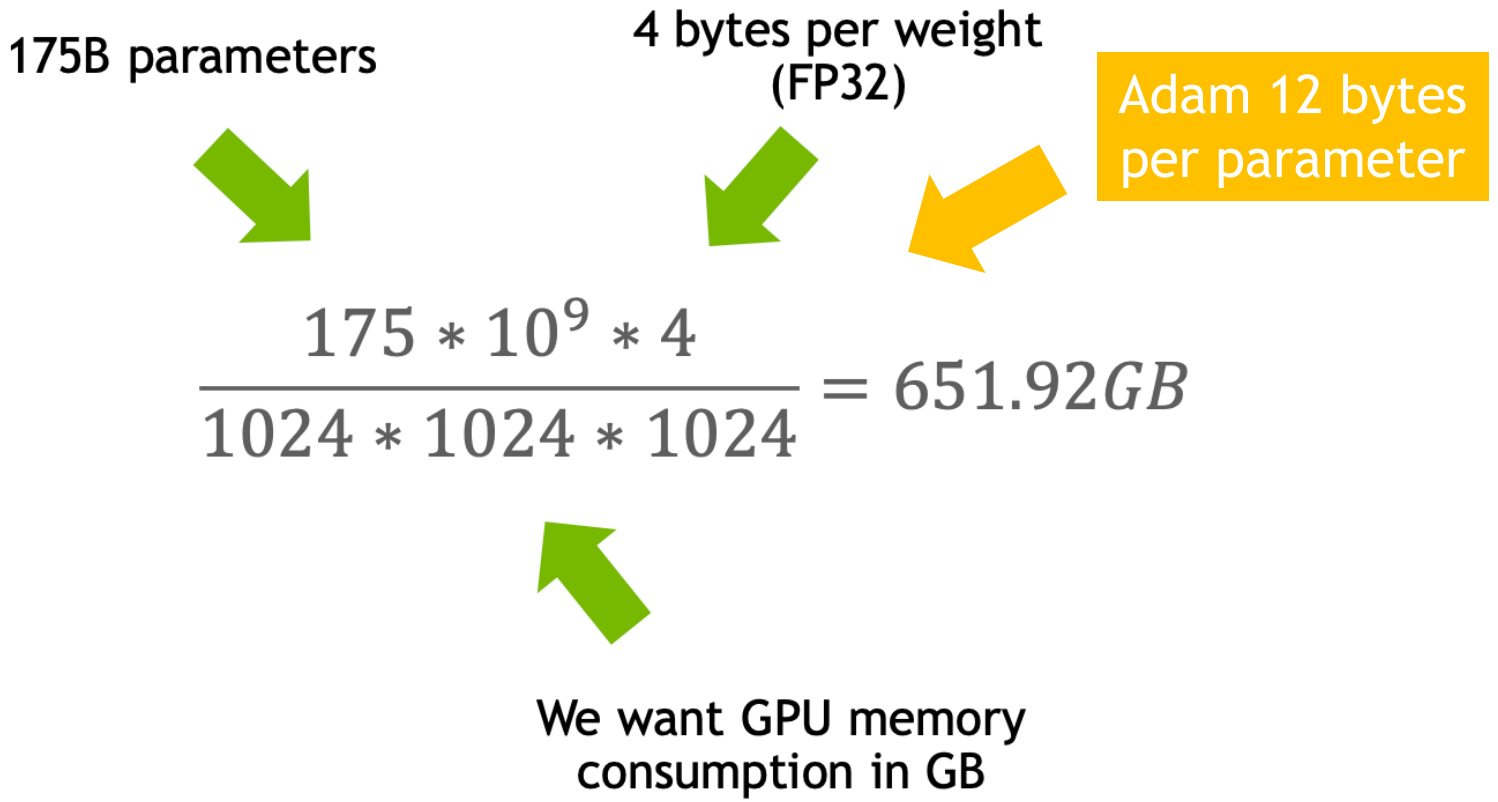
Execution times

Large models require large execution time

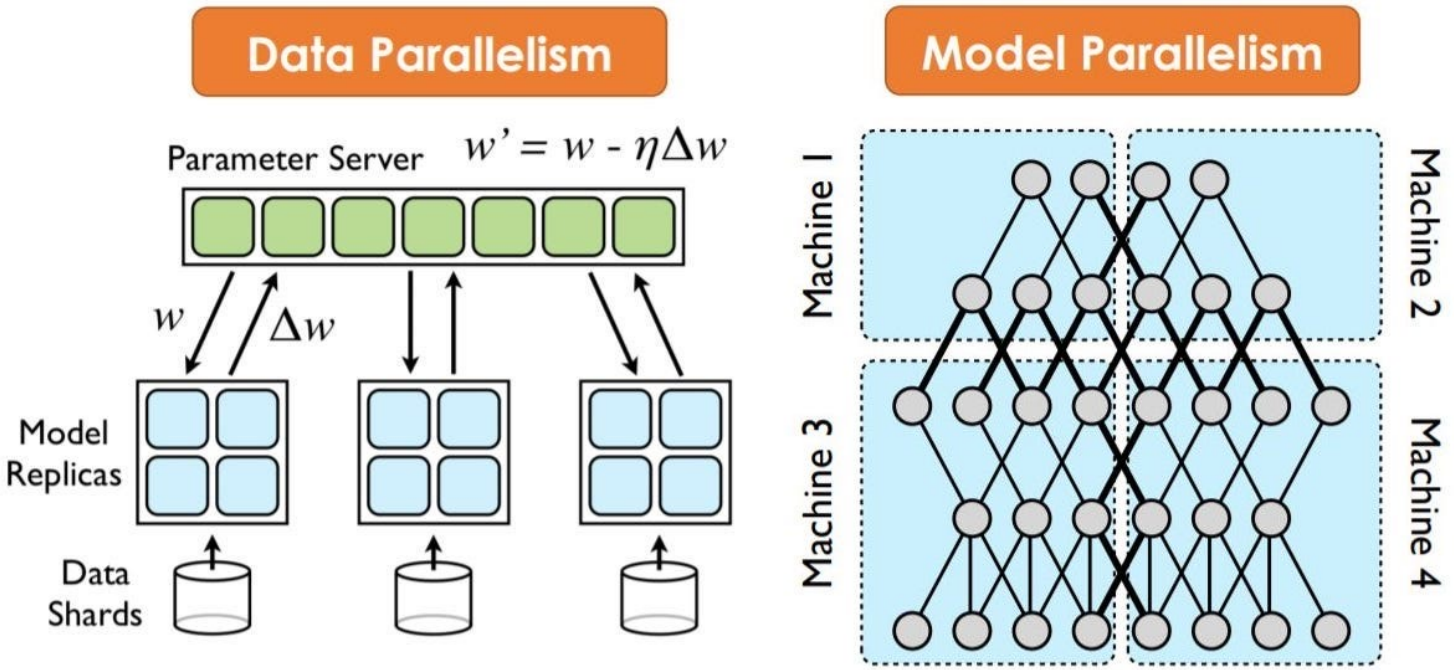


Narayanan, Deepak, et al. "**Efficient large-scale language model training on GPU clusters using megatron-LM**." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.

Memory footprint



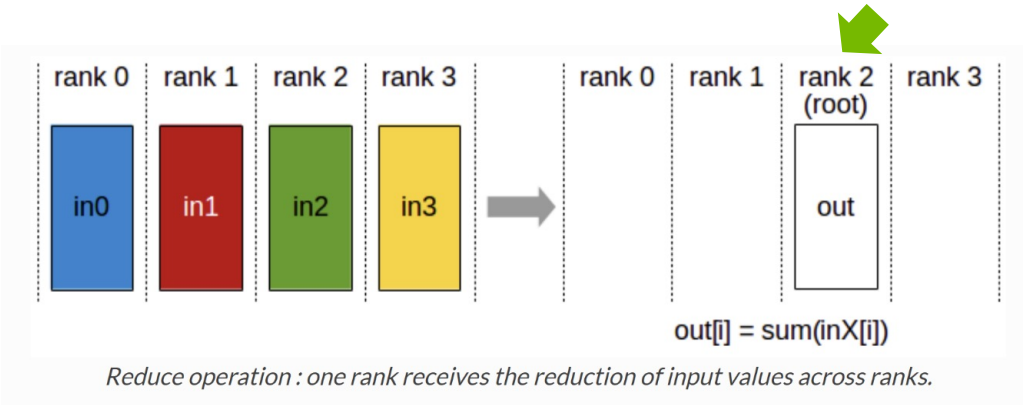
Data Parallelism - Model Parallelism



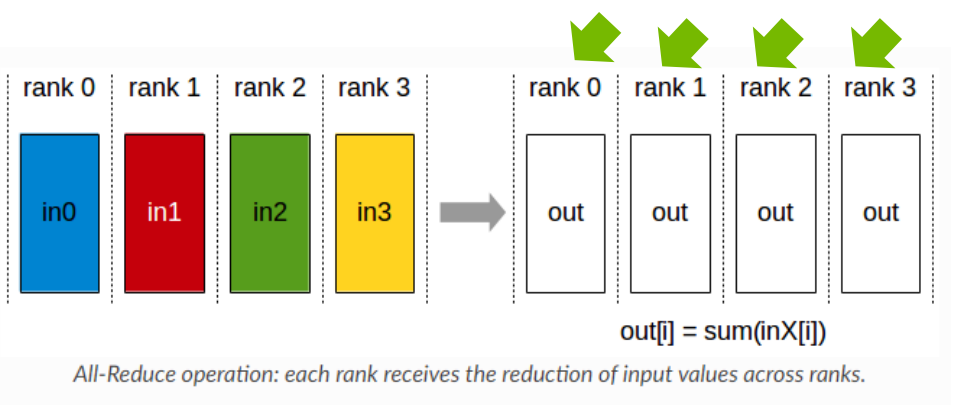
https://medium.com/nerd-for-tech/an-overview-of-pipeline-parallelism-and-its-research-progress-7934e5e6d5b8

Some notions

Collective Communications



Reduce operation : one rank receives the reduction of input values across ranks.

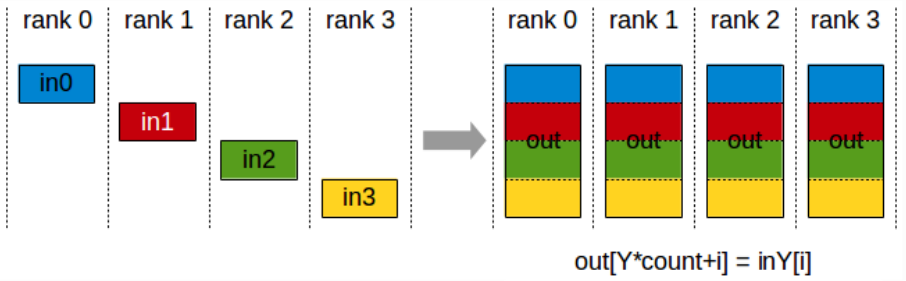


All-Reduce operation: each rank receives the reduction of input values across ranks.

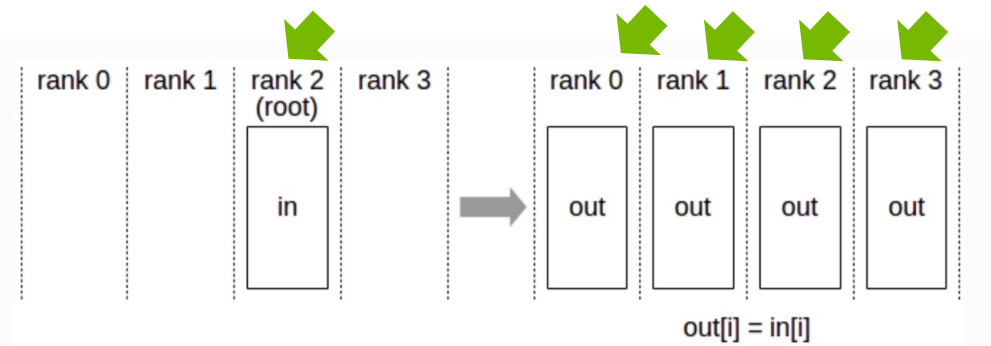
https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/operations.html

Some notions

Collective Communications



AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

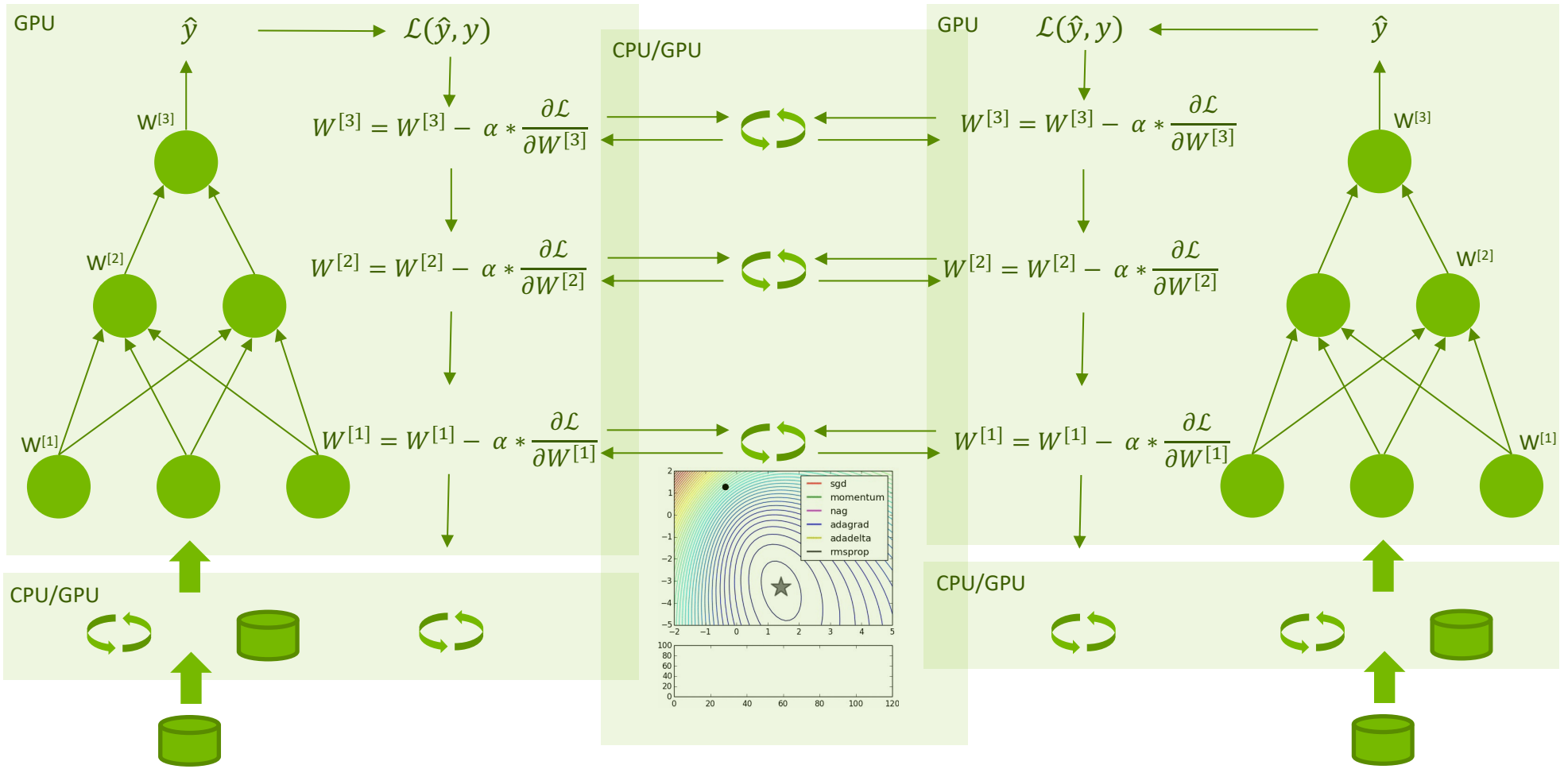


Broadcast operation: all ranks receive data from a "root" rank.

https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/operations.html

Training a Neural Network

Multiple GPUs



Parallelism with PyTorch

- **PyTorch** provides several options for distributed training for applications that gradually scale from simple to complex:
  1. Use single-device training if the data and model can fit in one GPU, and training speed is not a concern.
  2. Use single-machine multi-GPU **DataParallel** to make use of multiple GPUs on a single machine to speed up training with minimal code changes.
  3. Use single-machine multi-GPU **DistributedDataParallel**, if you would like to further speed up training and are willing to write a little more code to set it up.
  4. Use multi-machine **DistributedDataParallel** and the launching script, if the application needs to scale across machine boundaries.
  5. Use **torch.distributed.elastic** to launch distributed training if errors (e.g., out-of-memory) are expected or if resources can join and leave dynamically during training.

Data parallel vs distributed data parallel

- **torch.nn.DataParallel (deprecated)**
  - The DataParallel package enables single-machine multi-GPU parallelism with the lowest coding hurdle. It only requires a one-line change to the application code. Although DataParallel is very easy to use, it usually does not offer the best performance because it replicates the model in every forward pass.
- **torch.nn.parallel.DistributedDataParallel**
  - Compared to DataParallel, DistributedDataParallel requires one more step to set up, i.e., calling init_process_group. **DDP uses multi-process parallelism**. Moreover, the model is broadcast at DDP construction time instead of in every forward pass, which also helps to speed up training.
- **torch.distributed.elastic**
  - With the growth of the application complexity and scale, failure recovery becomes a requirement. **torch.distributed.elastic** adds fault tolerance and the ability to make use of a dynamic pool of machines (elasticity).

Distributed data parallel

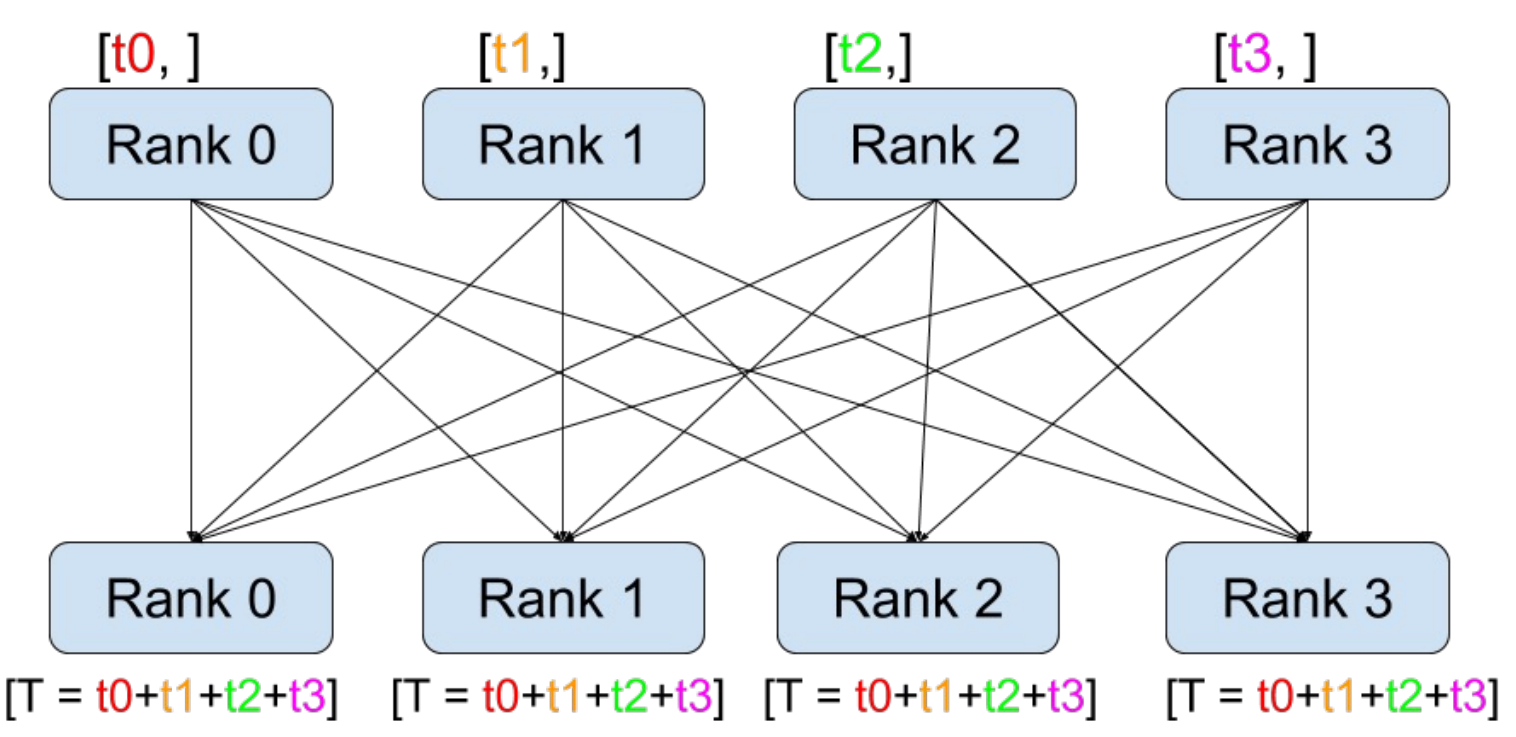
- **DistributedDataParallel** (DDP) implements data parallelism at the module level which can run across multiple machines.
- Applications using DDP should spawn multiple processes and create a single DDP instance per process.
- DDP uses collective communications in the ***torch.distributed*** package to synchronize gradients and buffers.

```
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP


def example(rank, world_size):
    # create default process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)
    # create local model
    model = nn.Linear(10, 10).to(rank)
    # construct DDP model
    ddp_model = DDP(model, device_ids=[rank])
    # define loss function and optimizer
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    # forward pass
    outputs = ddp_model(torch.randn(20, 10).to(rank))
    labels = torch.randn(20, 10).to(rank)
    # backward pass
    loss_fn(outputs, labels).backward()
    # update parameters
    optimizer.step()
```

Parallel launchers

```
# TORCHRUN
$ torchrun --standalone --nproc_per_node $GPUS_PER_NODE \
           --nnodes $NNODES your_training.py


# MPIRUN
$ export MASTER_ADDR=$(scontrol show hostnames \"$SLURM_JOB_NODELIST\" | head -n 1)
$ export MASTER_PORT=6000
$ mpirun -np $NUM_OF_TOTAL_TASKS \
           -x PATH \
           -map-by numa python your_training.py

# SRUN
$ srun -p boost_usr_prod --time 01:00:00 -N 1 \
       --ntasks-per-node=4 --gres=gpu:4 \
       python your_training.py
```

Ranks

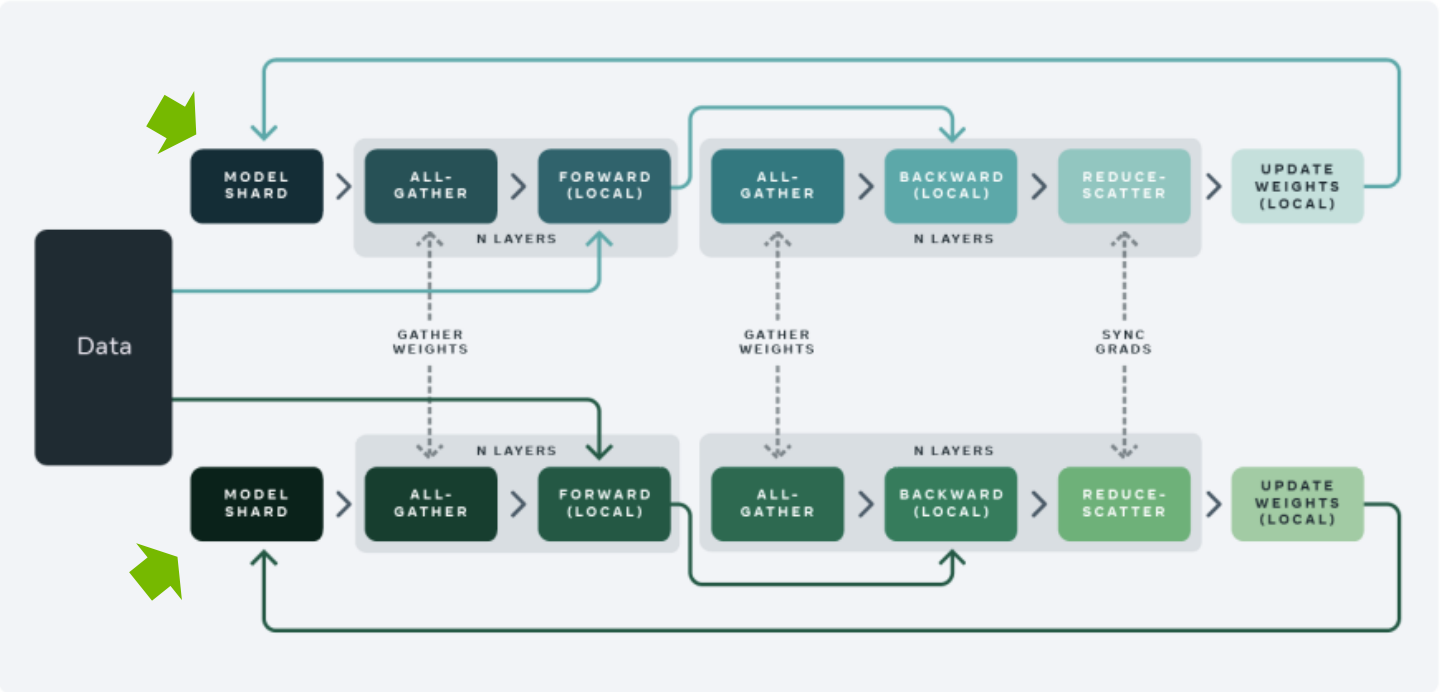


Distributed Data Parallel - DDP

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

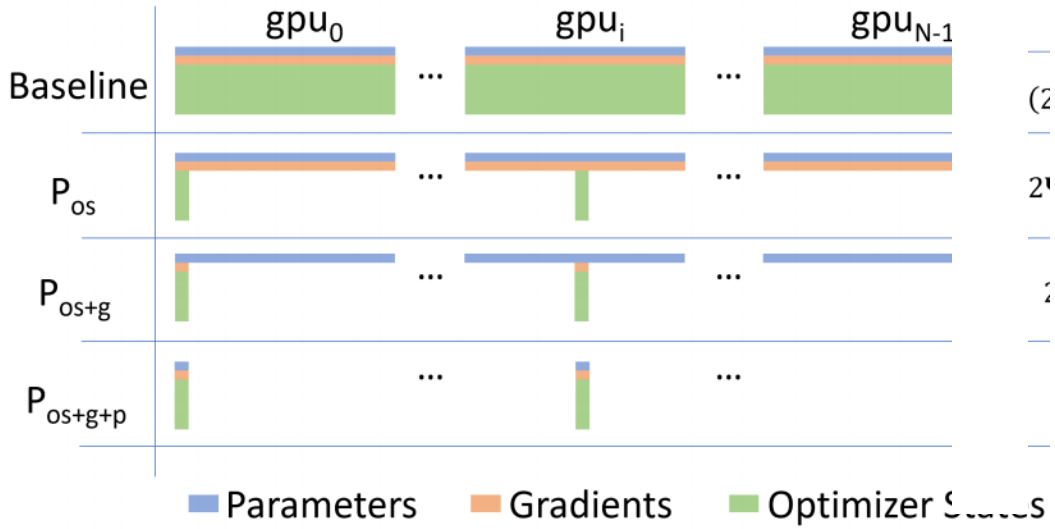


https://fairscale.readthedocs.io/en/stable/api/nn/fsdp.html

Sharded Data Parallelism

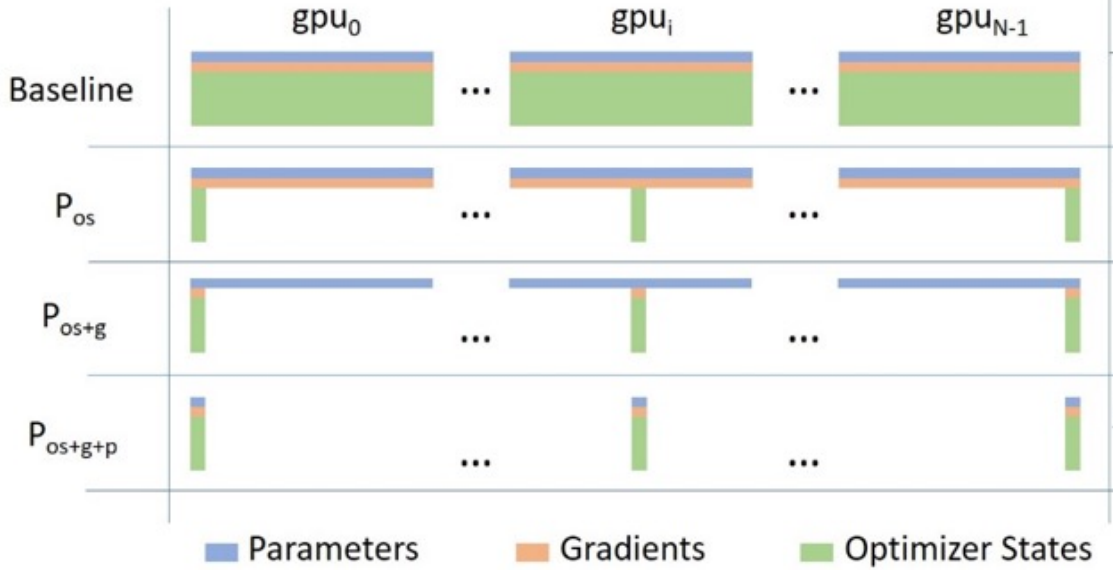
ZeRO: Zero Redundancy Optimizer

- 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

Sharded Data Parallelism

GPT2 example: Enable ZeRO optimization

```
# Construct FP16, distributed, GPT2 model
model = GPT2Model(num_layers=args.num_layers, ...)
model = FP16_Module(model)
model = DistributedDataParallel(model, ...)

...

# Construct FP16 Adam optimizer
optimizer = Adam(param_groups, ...)
optimizer = FP16_Optimizer(optimizer, ...)
```

```
# Forward pass
output = model(tokens, ...)

# Backward pass
optimizer.backward(loss)

# Parameter update
optimizer.step()
```

```
Python train.py <args>
```

```
# Construct GPT2 model
model = GPT2Model(num_layers=args.num_layers, ...)

# Construct Adam optimizer
optimizer = Adam(param_groups, ...)

# Wrap model, optimizer, and lr scheduler
model, optimizer, lr_scheduler, _ = deepspeed.initialize(
    args=args,
    model=model,
    optimizer=optimizer,
    lr_scheduler=lr_scheduler,
    mpu=mpu
)
```

```
# Forward pass
output = model(tokens, ...)

# Backward pass
model.backward(loss)

# Parameter update
model.step()
```

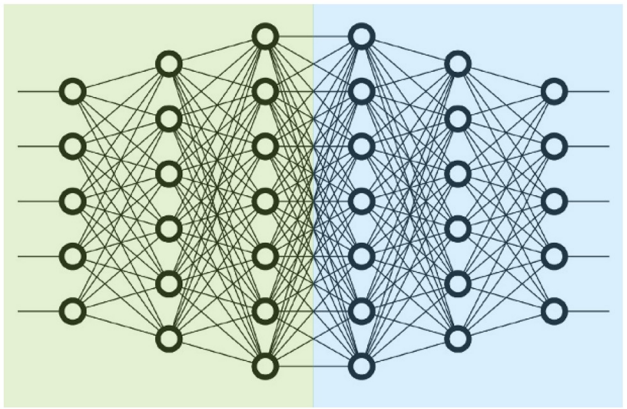
```
deepspeed train.py <args> \
--deepspeed_config ds_config.json
```

```
{
  "train_batch_size": 64,
  "gradient_accumulation_steps": 1,
  "steps_per_print": 1,
  "zero_optimization": {
    "stage": 3,
    "stage3_max_live_parameters": 1e9,
    "stage3_max_reuse_distance": 1e9,
    "stage3_prefetch_bucket_size": 1e7,
    "stage3_param_persitence_threshold": 1e5,
    "reduce_bucket_size": 1e7,
    "contiguous_gradients": true
  },
  "gradient_clipping": 1.0,
  "fp16": {
    "enabled": true,
    "loss_scale": 0,
    "loss_scale_window": 1000,
    "hysteresis": 2,
    "min_loss_scale": 1
  },
  "wall_clock_breakdown": true,
  "zero_allow_untested_optimizer": false
}
```

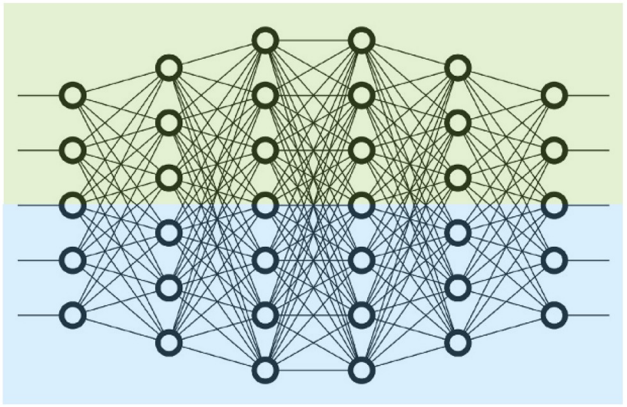
For more details see: https://aka.ms/deepspeed-gpt2

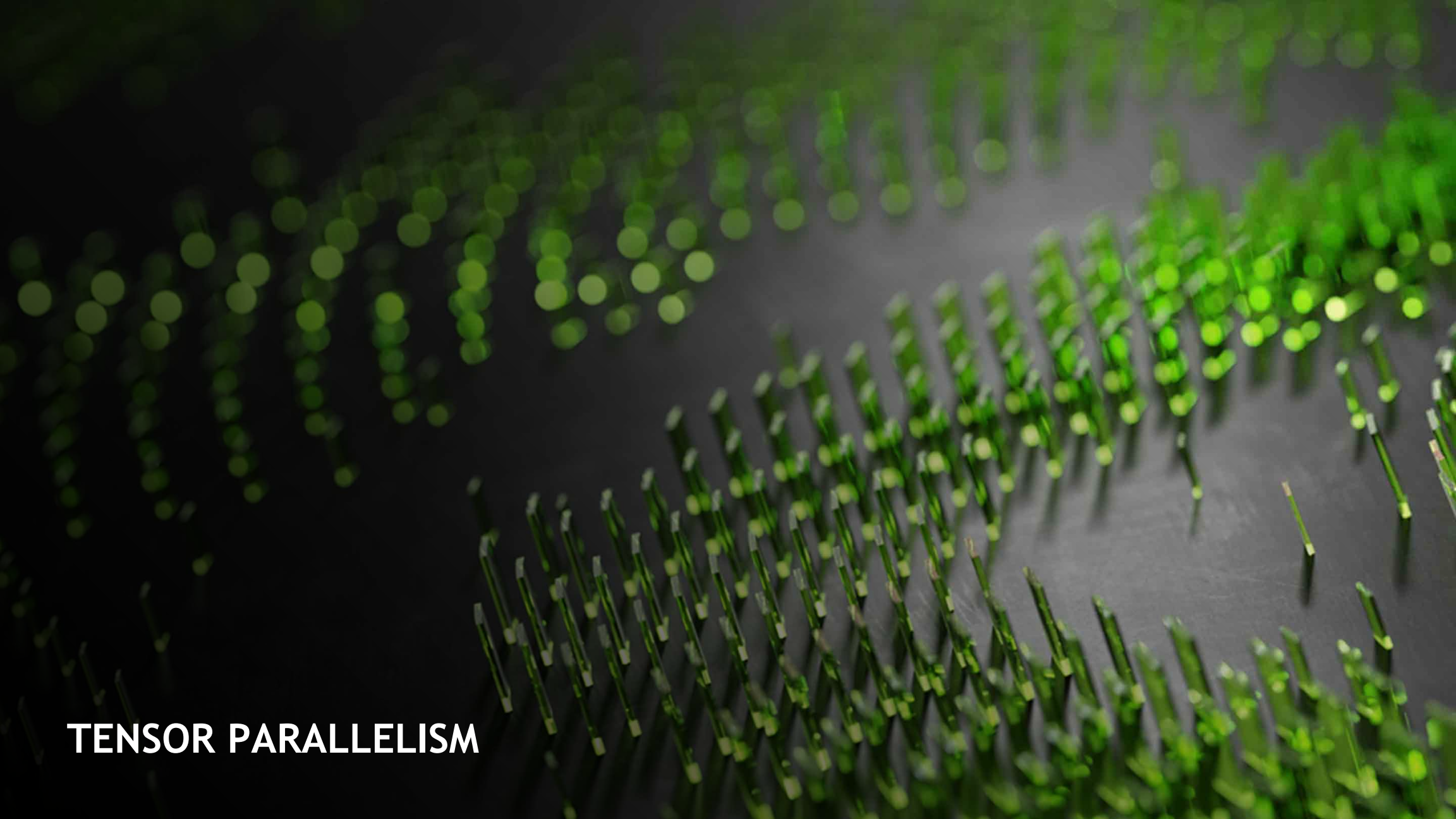
Model Parallelism

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



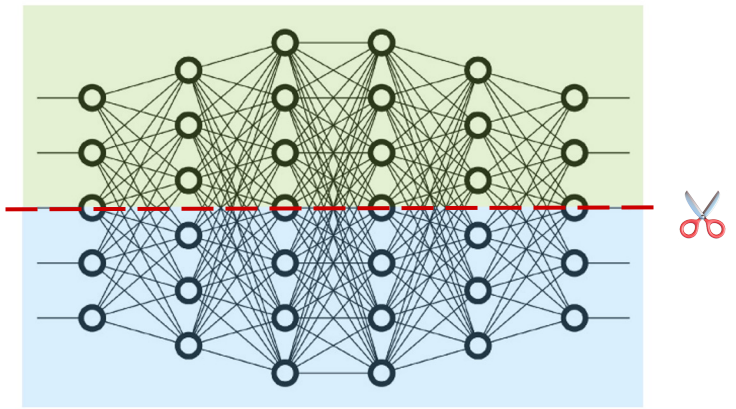
- **Tensor (Intra-Layer) Parallelism**
  - Split individual layers across multiple devices
  - Both devices compute difference parts of Layer 0,1,2,3,4,5



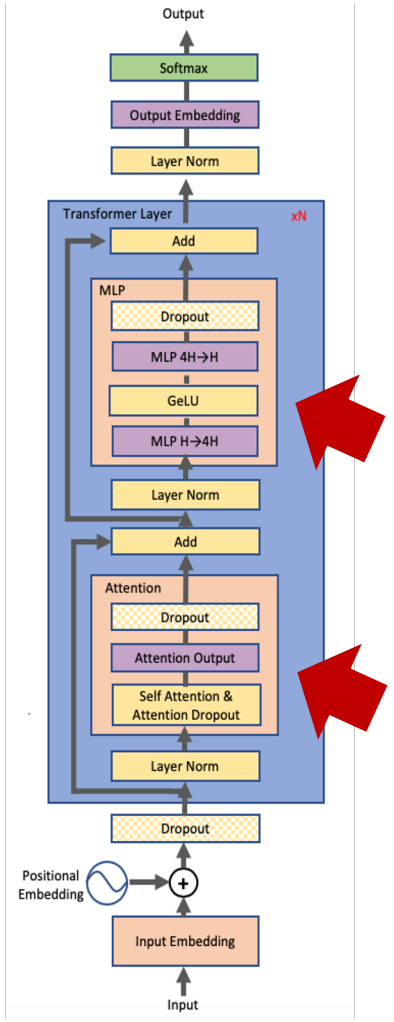

TENSOR PARALLELISM

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



Transformers cell



MLP Tensor Partitioning

Focus on the GeLU operation:

- **Approach 1:** Split X column-wise and A row-wise:

$$X = [X_1, X_2] \quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \Rightarrow Y = \mathrm{GeLU}(X_1 A_1 + X_2 A_2)$$

  - Before GeLU we will need a communication point

- **Approach 2:** Split A column-wise:

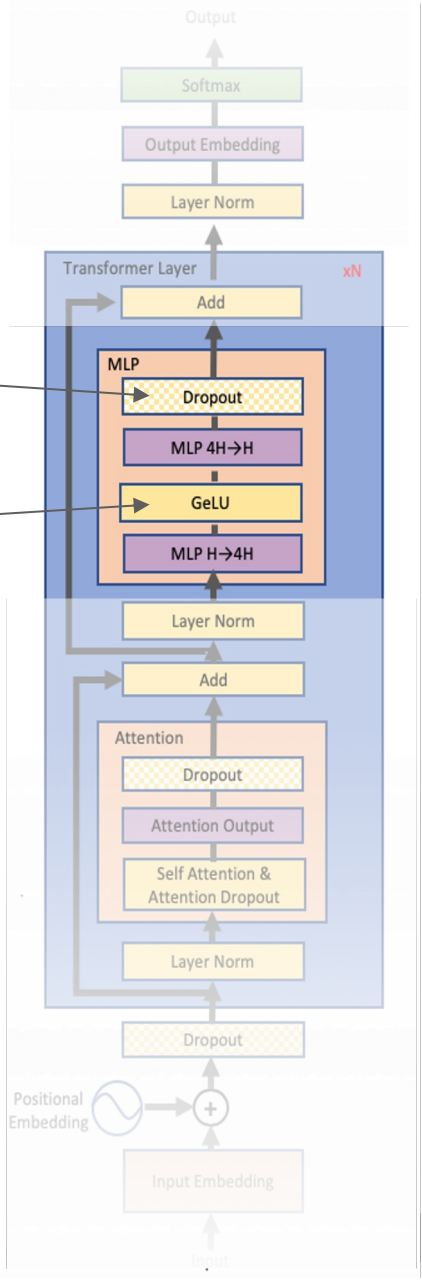
$$A = [A_1, A_2] \Rightarrow [Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$$

  - No communication is required

Chosen approach

$$Z = \mathrm{Dropout}(YB)$$

$$Y = \mathrm{GeLU}(XA)$$

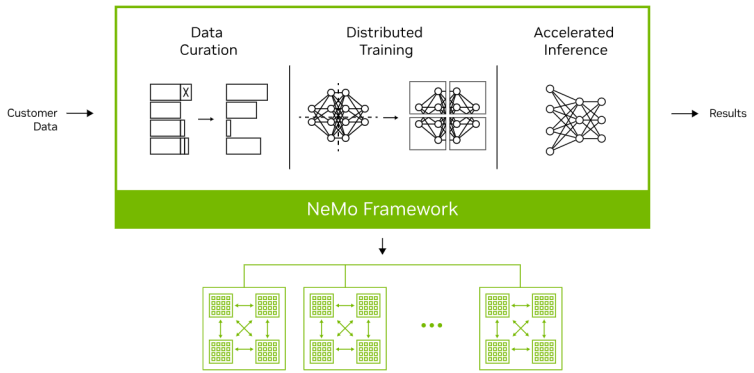


Tensor Parallelism implementations

Libraries examples

NVIDIA/**Megatron-LM**

Ongoing research training transformer models at scale



deepspeed

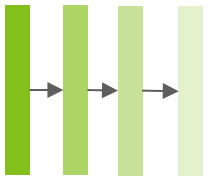
Mesh TensorFlow

FairScale

Colossal-AI

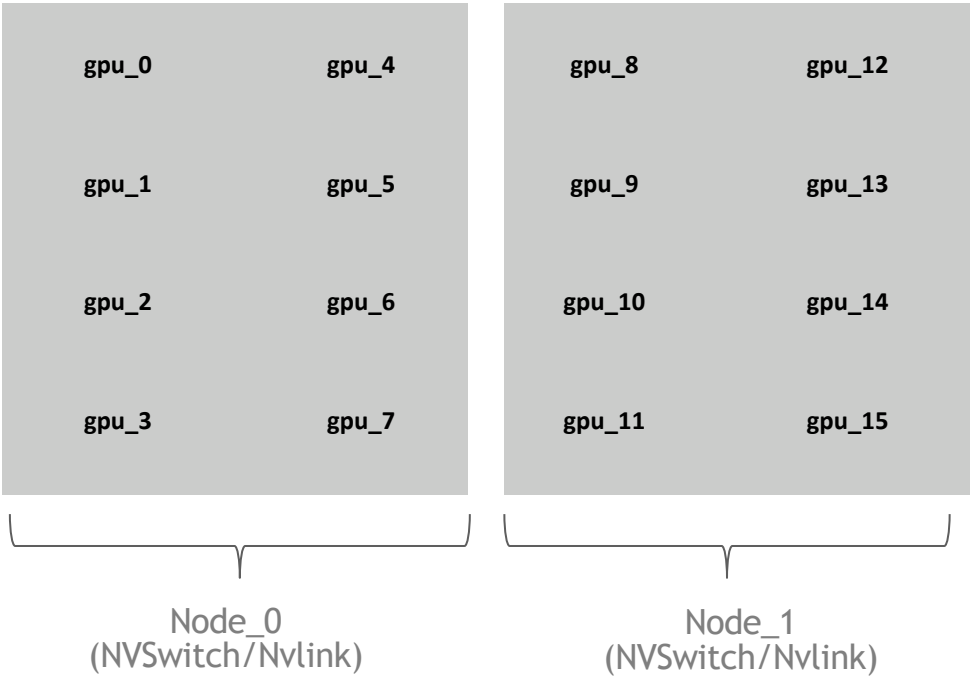
Megatron model parallelism

GPU Affinity Grouping Example



Neural Network: 4 layers
Hardware: 2 nodes , 8 GPUs per node

- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2

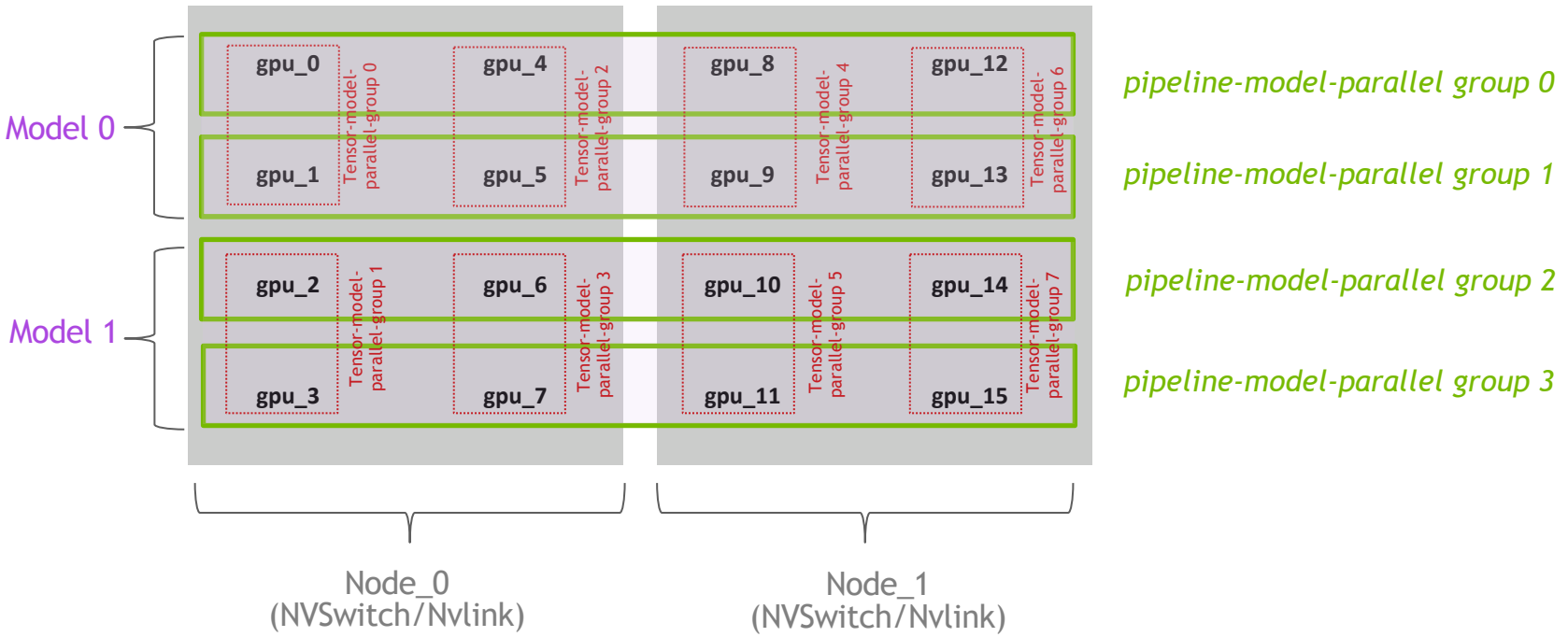


https://github.com/NVIDIA/Megatron-LM

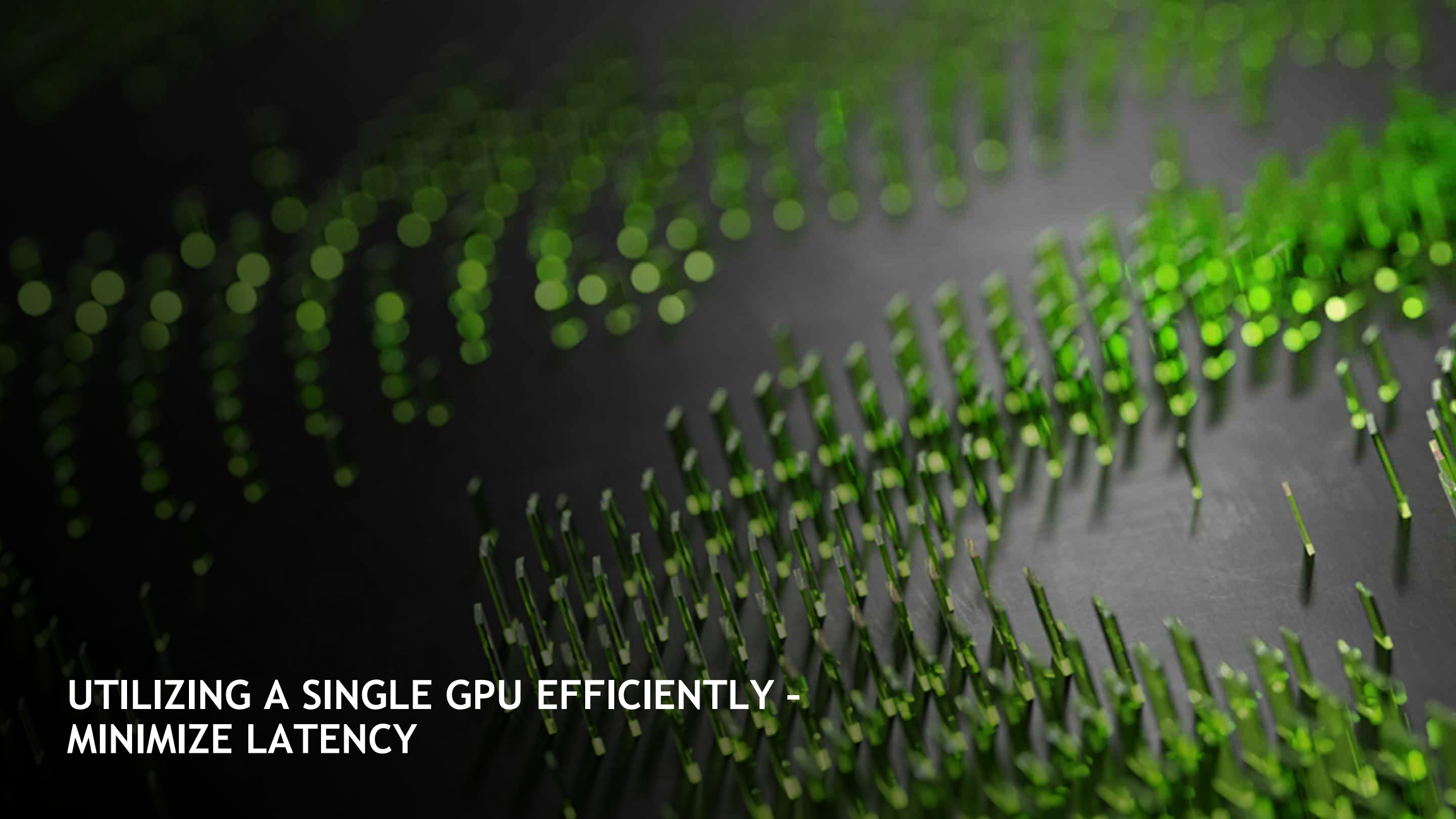
Megatron model parallelism

GPU Affinity Grouping Example

2 nodes , 8 GPUs per node
- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2



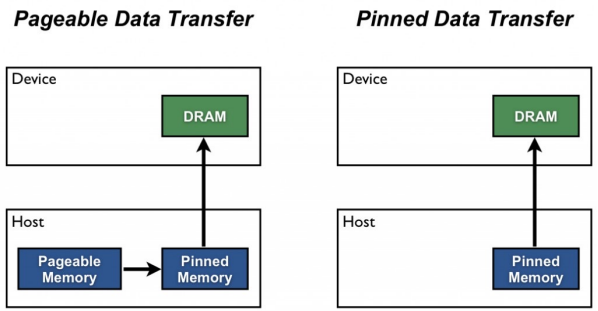
https://github.com/NVIDIA/Megatron-LM


UTILIZING A SINGLE GPU EFFICIENTLY –
MINIMIZE LATENCY

OPTIMIZE DATA LOADING

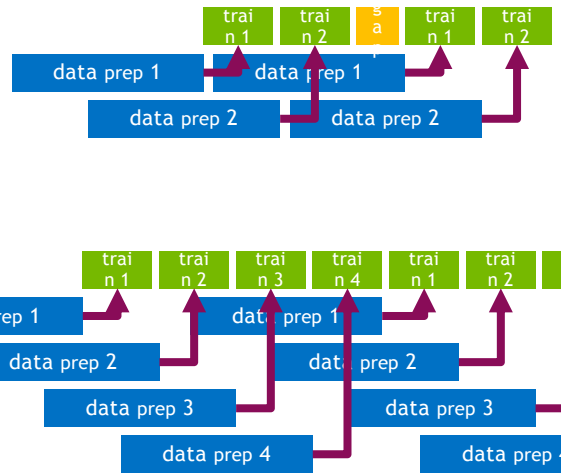
"keep the pan filled"

- If the dataset is small enough, consider moving it entirely onto the GPU.
- Use pinned memory: Host to GPU (H2D) copies are much faster when they originate from pinned (page-locked) memory. This also works for individual tensors!



- Tune the number of workers for loading the data in keep the GPU busy.
- Tune prefetching, i.e., how many samples are prefetched by the dataloader.
- For more information see https://pytorch.org/docs/stable/data.html

```
# in pytorch
loader = DataLoader(dataset, … ,
                    pin_memory=True,
                    num_workers=4,
                    prefetch_factor=2)

# pinning individual tensors
Tensor.pin_memory()
```



Async data copy

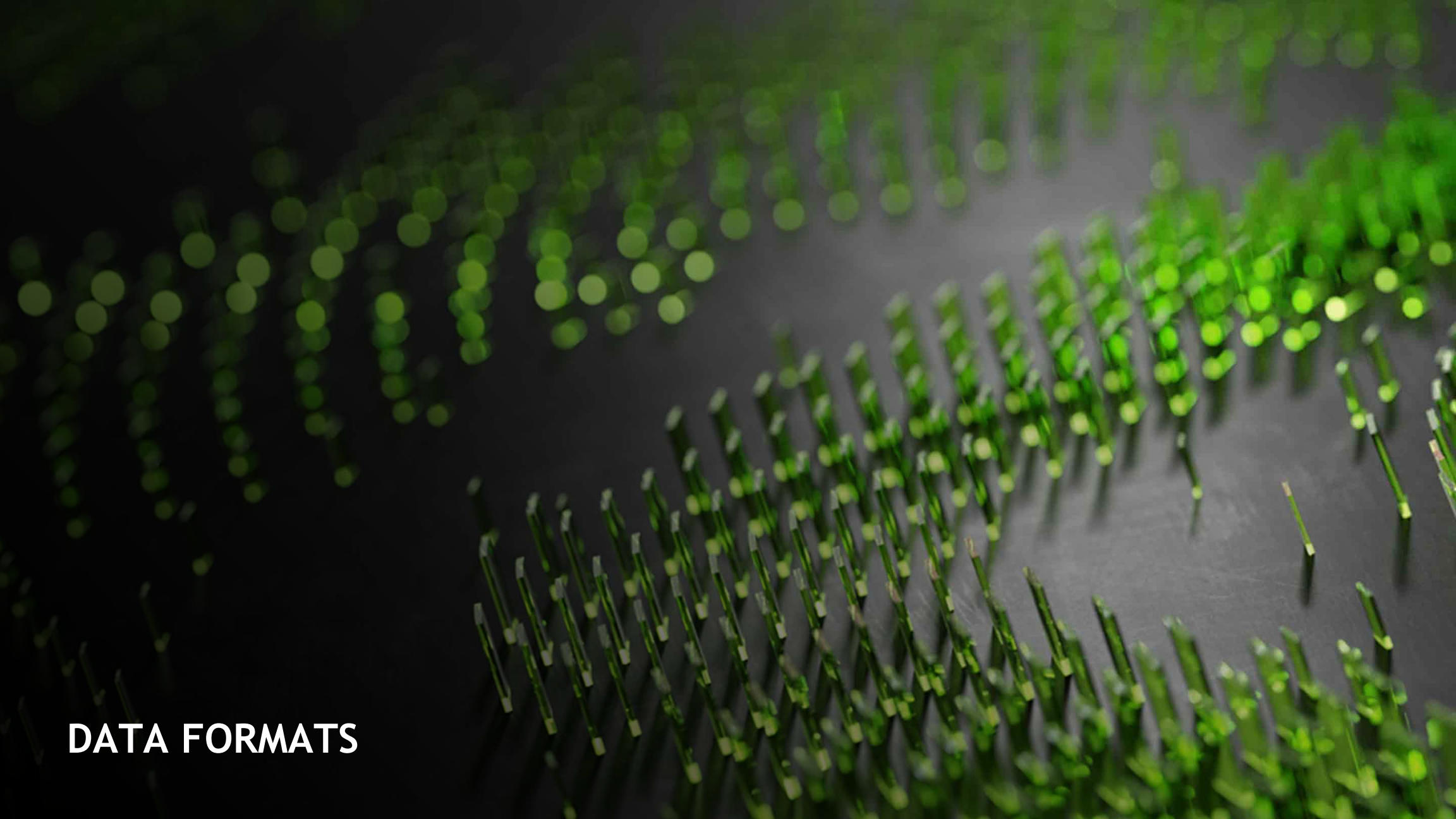
```
dataloader_train = DataLoader(pin_memory = False, **kargs)
```


```
dataloader_train = DataLoader(pin_memory = True, **kargs)
model.to(device, non_blocking=True)
```

Pre-fetching

PyTorch

https://www.jpatrickpark.com/post/prefetcher/


DATA FORMATS

Data Formarts

Packed Dataset

- Many file systems and network storage (including NGC dataset) don't handle well datasets with a vast number of small files *(e.g., Imagenet stored as 1.3M individual .jpeg files)*, which adds an overhead of opening and closing files on top of data transfer over fabrics.
- This problem becomes especially severe if multiple jobs access the same dataset concurrently.
- Typical packed data formats include:
  - **WebDataset**
    - https://webdataset.github.io/webdataset/gettingstarted/
  - **LMDB**
    - http://www.lmdb.tech/doc/
  - **HDF5**
    - https://www.hdfgroup.org/solutions/hdf5/


JIT, TORCHSCRIPT, XLA

JIT and TorchScript

- **JIT** is a dynamic tracing compiler that generates optimized code on the fly during runtime.
- It works by tracing the execution of a PyTorch model during training and generates optimized code that can be reused during inference.
- The advantage of JIT is that it can generate optimized code for a specific input size, making it very efficient for that input size.
- JIT does not provide any static guarantees about the correctness of the generated code.

- **TorchScript** is a static graph compiler that generates optimized code ahead of time.
- It works by converting a PyTorch model into a graph representation that can be optimized and compiled for execution on different devices.
- The advantage of TorchScript is that it provides static guarantees about the correctness of the generated code, making it more reliable for production use.
- However, TorchScript requires a bit more effort upfront to convert the PyTorch model into a graph representation.

JIT and TorchScript

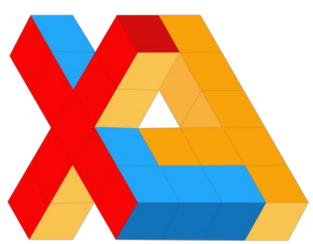
```
import torch
import torch.nn as nn

# Define a simple PyTorch model
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(2, 1)

    def forward(self, x):
        return self.linear(x)

# Create an instance of the model
model = SimpleModel()
```

```
# Define some example input
input = torch.randn(1, 2)

# Use JIT to generate optimized code on the
# fly
jit_model = torch.jit.trace(model, input)

# Use TorchScript to generate optimized
# code ahead of time
script_model = torch.jit.script(model)

# Evaluate the models on the input
jit_output = jit_model(input)
script_output = script_model(input)

# Compare the outputs
print(jit_output)
print(script_output)
```

XLA

Accelerate Linear Algebra

- XLA stands for Accelerated Linear Algebra and it is a domain-specific compiler for linear algebra operations developed by Google.
- It is designed to optimize the performance of machine learning models by compiling and executing them on a variety of devices, including CPUs, GPUs, and TPUs (Tensor Processing Units).
- The main idea behind XLA is to generate highly optimized device-specific code for linear algebra operations. To achieve this, XLA performs a number of optimizations such as loop unrolling, kernel fusion, and memory layout transformations. XLA also supports automatic differentiation, which is an essential feature for training deep learning models.
- XLA can be used with PyTorch through the PyTorch/XLA package, which provides a PyTorch interface to XLA.



XLA

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as
transforms

# Set up the device to use XLA
device = xm.xla_device()

# Define the CNN architecture
class Net(nn.Module):
[...]

# Create an instance of the model and
# move it to the XLA device
net = Net()
net.to(device)
```

```
# Create an instance of the optimizer and
# move it to the XLA device
optimizer = optim.SGD(net.parameters(),
lr=lr)
optimizer = xla.optimizer(optimizer,
device=device)

# Train the model
for epoch in range(num_epochs):
[...]
        xm.optimizer_step(optimizer)


print('Finished training')
```

JIT, TORCHSCRIPT, XLA

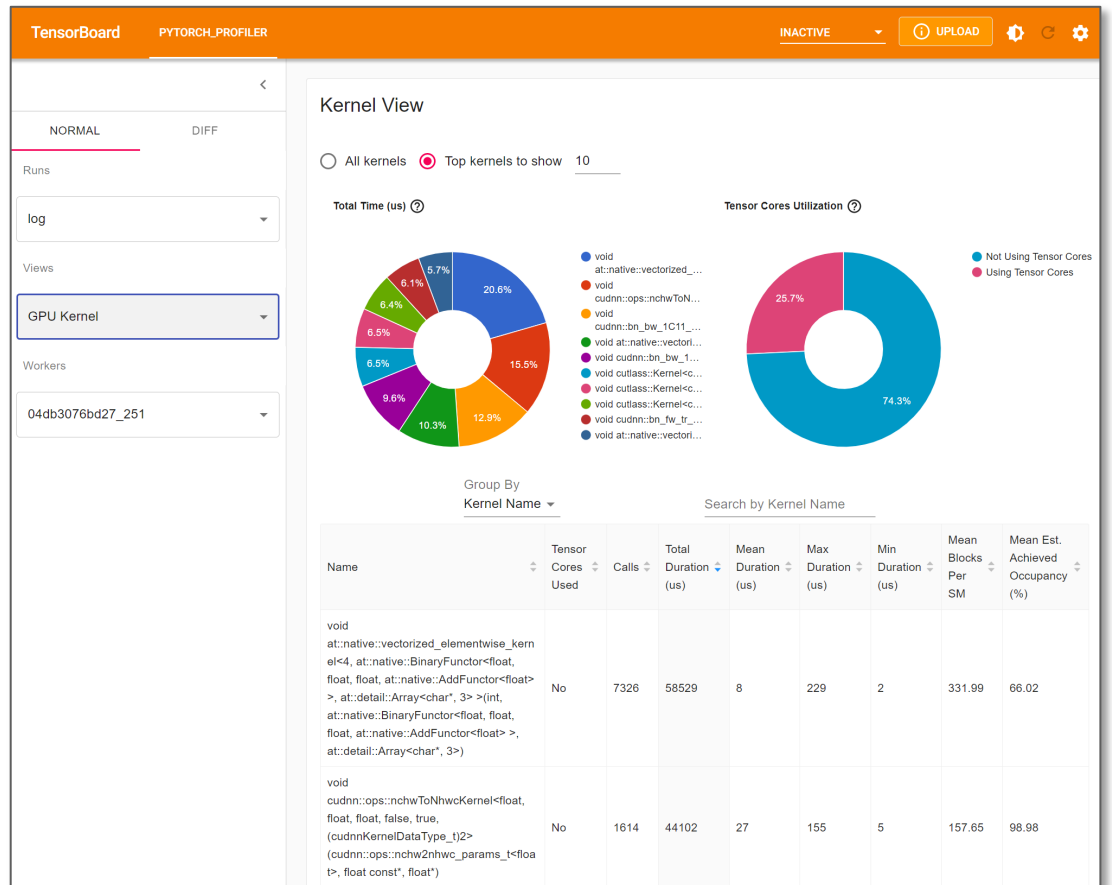
- Performance depends on the specific use case and the hardware being used.
- In general
  - **JIT** provides the most flexibility and can provide significant speedups for inference on a CPU or GPU.
  - **TorchScript** is optimized for deployment and can provide significant speedups and size reductions for models running on a variety of devices.
  - **XLA** is optimized for linear algebra operations and can provide significant speedups for training and inference on a variety of devices, particularly TPUs.


PROFILING YOUR CODE

AM I USING TENSOR CORES?

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

```
from torch import profiler

prof_schedule = profiler.schedule(wait=2,
                                  warmup=2,
                                  active=5,
                                  repeat=0)

callback =
profiler.tensorboard_trace_handler('./log')

prof = profiler.profile(schedule=prof_schedule,
on_trace_ready=callback,
record_shapes=False,
with_stack=False)

prof.start()

for it in range(num_iterations):
    # code to be profiled
    ...
    prof.step()

prof.stop()
```

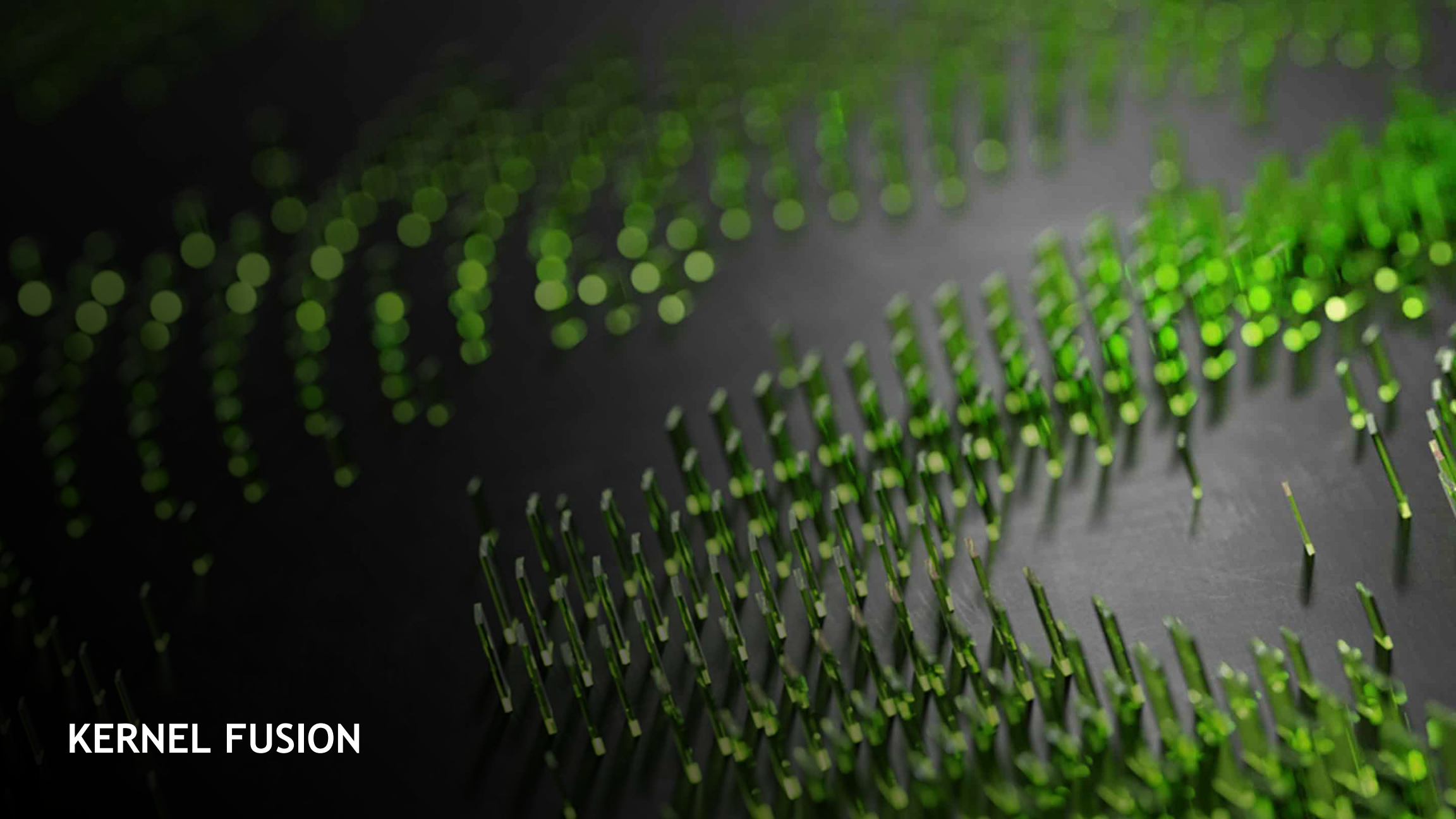


PYTORCH PROFILER

https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html

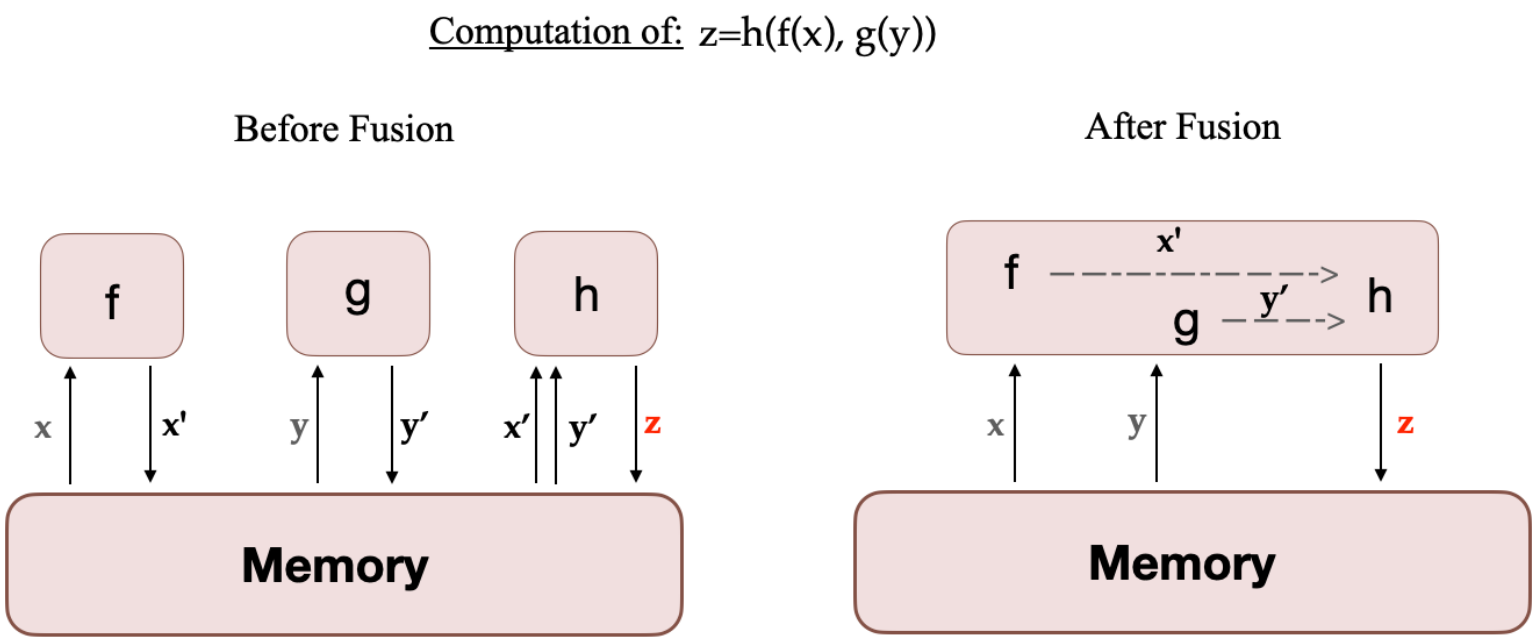
- Build-in in PyTorch
- Allows to improve performance of your models visually
- Features
  - Tensor Core Usage and Eligibility Detection
  - Kernel view
  - Stack view
  - Performance recommendations
  - …

```
# ssh into your machine
ssh -L 9999:localhost:9999 user@<server_ip>

# run docker container
$ docker run --gpus all -d -p 9999:9999 -v
/path/to/my/project/:/workspace
nvcr.io/nvidia/pytorch:21.09-py3

# install tensor board plugin
$ pip install torch_tb_profiler

# run application (example from repo)
$ python tiling_and_tensor_cores_pytorch.py

# start tensorboard
$ tensorboard --logdir=./log --port=9999
```

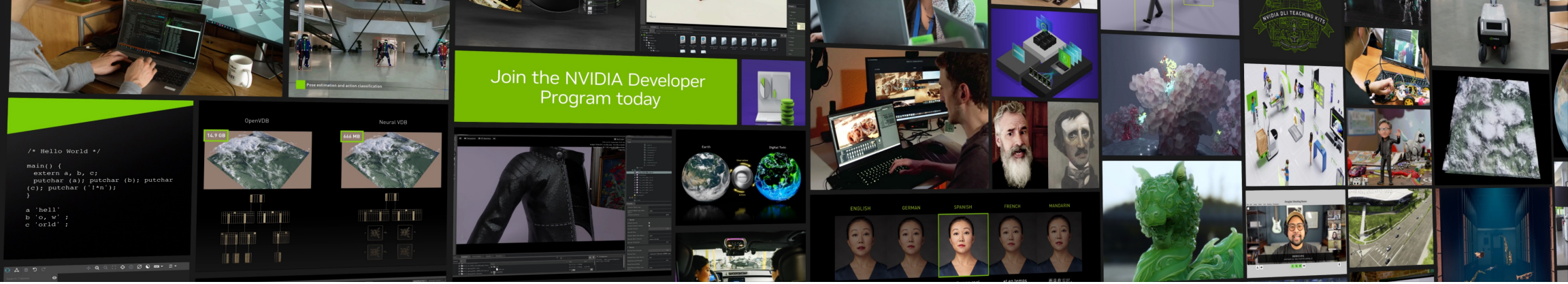

KERNEL FUSION

Fused CUDA Kernels

https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/



https://huggingface.co/blog/megatron-training



NVIDIA Developer Program

The Community that Builds

Program Benefits:

Tools

- 550+ exclusive SDKs and models
- GPU-optimized software, model scripts, and containerized apps
- Early access programs and the NVIDIA Academic Hardware Grant Program*

Training

- Research papers, technical documentation, webinars, blogs, and news
- Technical training and certification opportunities
- 1,000s of technical sessions from industry events On-Demand

Community

- NVIDIA developer forums
- Exclusive meetups, hackathons, and events

Join the Community

** The Hardware Grant Program is available to qualified researchers and educators.*

NVIDIA


NVIDIA
Many thanks!