Deep Learning Optimisé - Jean Zay

Best Practices and State Of The Art

Institut du Développement et des Ressources en Informatique Scientifique

Performance Tuning Guide

PyTorch

- Enable asynchronous data loading and augmentation

```
torch.utils.data.DataLoader
num_workers > 0
pin_memory=True
```

- Disable gradient calculation for validation or inference

```
with torch.no_grad():
    val_outputs = model(val_images)
    val_loss = criterion(val_outputs, val_labels)
```

- Use mixed precision and AMP

```
from torch.cuda.amp import autocast, GradScaler
with autocast():
```

- Use efficient data-parallel backend

```
torch.nn.parallel.DistributedDataParallel
```

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Performance Tuning Guide

PyTorch

- Disable bias for convolutions directly followed by a batch norm

```
nn.Conv2d(..., bias=False, ....)
Models available from torchvision already
implement this optimization.
```

- Enable channels_last memory format for computer vision models

```
x = x.to(memory_format=torch.channels_last)
```

- Disable debugging APIs

```
anomaly detection: torch.autograd.detect_anomaly or torch.autograd.set_detect_anomaly(True)
profiler related: torch.autograd.profiler.emit_nvtx, torch.autograd.profiler.profile
autograd gradcheck: torch.autograd.gradcheck or torch.autograd.gradgradcheck
```

- Create tensors directly on the target device

~~torch.rand(size).cuda()~~

```
torch.rand(size, device='cuda')
```

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Performance Tuning Guide

PyTorch

- Fuse pointwise operations
  Pointwise operations (elementwise addition, multiplication, math functions - sin(), cos(), sigmoid() etc.) can be fused into a single kernel to amortize memory access time and kernel launch time. PyTorch JIT can fuse kernels automatically.

```
@torch.jit.script
def fused_gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

- Enable cuDNN auto-tuner
  For convolutional networks

```
torch.backends.cudnn.benchmark = True
```

- Avoid unnecessary CPU-GPU synchronization

```
print(cuda_tensor)
cuda_tensor.item()
memory copies: tensor.cuda(), cuda_tensor.cpu() and equivalent tensor.to(device) calls
cuda_tensor.nonzero()
python control flow e.g. if (cuda_tensor != 0).all()
```

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Performance Tuning Guide

- Load-balance workload in a distributed setting
  The core idea is to distribute workload over all workers as uniformly as possible within each global batch. For example Transformer solves imbalance by forming batches with approximately constant number of tokens (and variable number of sequences in a batch), other models solve imbalance by bucketing samples with similar sequence length or even by sorting dataset by sequence length.

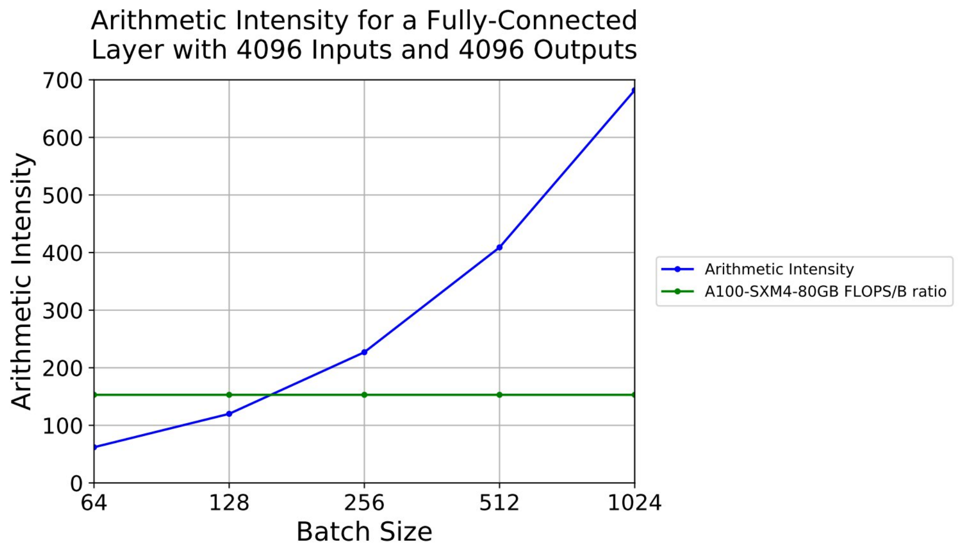
- Preallocate memory in case of variable input length
  For Speech Recognition or NLP, preexecute a forward and a backward pass with a generated batch of inputs with maximum sequence length (either corresponding to max length in the training dataset or to some predefined threshold). This step preallocates buffers of maximum size, which can be reused in subsequent training iterations.

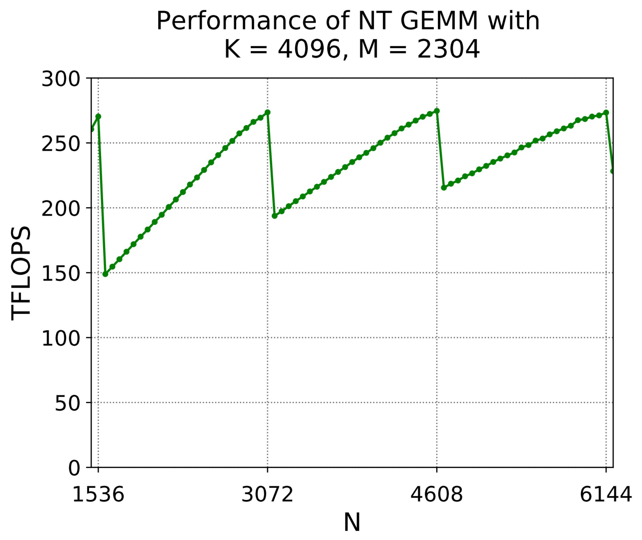
- Match the order of layers in constructors and during the execution if using DistributedDataParallel``(find_unused_parameters=True)
  To maximize the amount of overlap, the order in model constructors should roughly match the order during the execution. If the order doesn't match, then all-reduce for the entire bucket waits for the gradient which is the last to arrive.
  With find_unused_parameters=False it's not necessary to reorder layers or parameters to achieve optimal performance.

https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Deep Learning Performance Documentation

$$\text{Arithmetic Intensity} = \frac{\text{number of FLOPS}}{\text{number of byte accesses}} = \frac{2 \cdot (M \cdot N \cdot K)}{2 \cdot (M \cdot K + N \cdot K + M \cdot N)} = \frac{M \cdot N \cdot K}{M \cdot K + N \cdot K + M \cdot N}$$





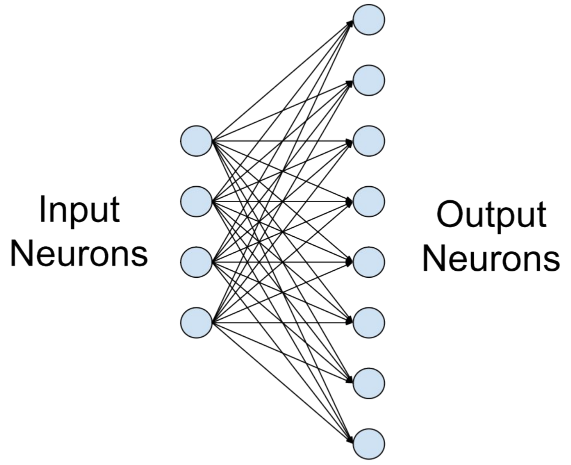
Wave Quantization effect

https://docs.nvidia.com/deeplearning/performance/index.html

Linear/Fully-Connected Layers User's Guide

The following quick start checklist provides specific tips for **fully-connected layers**.

- 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 ones 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.

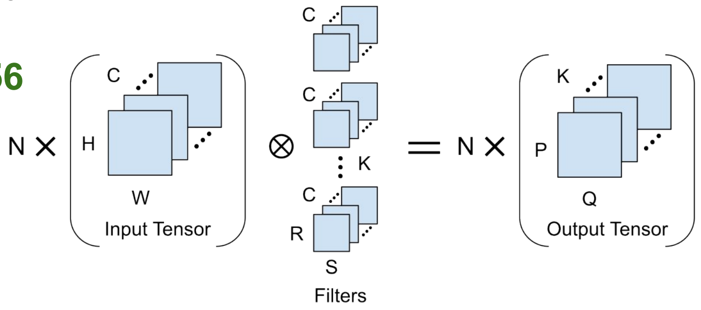


https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html

Convolutional Layers User's Guide

The following quick start checklist provides specific tips for **convolutional layers**.

- Choose the number of **input and output channels** to be divisible **by 8 (for FP16) or 4 (for TF32)** to run efficiently on **Tensor Cores**. For the **first convolutional layer** in most CNNs with **3-channel** images, **padding to 4 channels** is sufficient if a stride of 2 is used.
- Choose parameters to be divisible by **at least 64** and **ideally 256** to enable efficient tiling and reduce overhead.
- **Larger values** for size-related parameters can improve parallelization.
- When the **size of the input is the same** in each iteration, **autotuning** is an efficient method to ensure the selection of the ideal algorithm for each convolution in the network. `torch.backends.cudnn.benchmark = True`.
- Choose tensor layouts in memory to avoid transposing input and output data. We recommend using the **NHWC format** where possible.

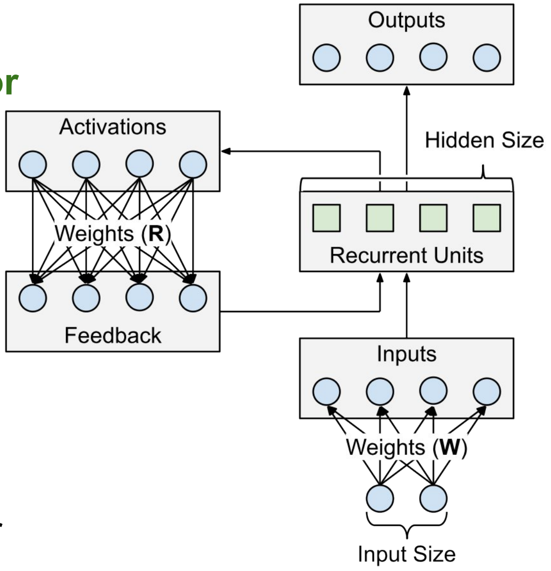


https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html

Recurrent Layers User's Guide

The following quick start checklist provides specific tips for **recurrent layers**.

- **Recurrent operations can be parallelized**. We recommend using NVIDIA® cuDNN implementations, which do this automatically.
- When using the **standard implementation**, minibatch size and hidden sizes should be:
  - **Divisible by 8 (for FP16)** or **4 (for TF32)** to run efficiently on **Tensor Cores**.
  - **Divisible by at least 64** and **ideally 256** to improve tiling efficiency.
  - **Greater than 128 (minibatch size)** or **256 (hidden sizes)** to be limited by computation rate rather than memory bandwidth.
- When using the **persistent implementation** (available for FP16 data only):
  - **Hidden sizes** should be **divisible by 32** to run efficiently on Tensor Cores. Better tiling efficiency may be achieved **by larger multiples of 2, up to 256**.
  - **Minibatch size** should be **divisible by 8** to run efficiently on Tensor Cores...
- **Try increasing parameters for better efficiency**.



https://docs.nvidia.com/deeplearning/performance/dl-performance-recurrent/index.html

Memory-Limited Layers User's Guide

The following quick start checklist provides specific tips **for layers whose performance is limited by memory accesses (Batch Normalization, Activations, Pooling, ...)**.

- Explore the available implementations of each layer in the **NVIDIA cuDNN API** Reference or your framework. Often the best way to improve performance is to choose **a more efficient implementation**.

- **Be aware of the number of memory accesses** required for each layer. Performance of a memory-bound calculation is simply based on the number of inputs, outputs, and weights that need to be loaded and/or stored per pass. We don’t have recommended parameter tweaks for these layers.

- **Be aware of the impact of each layer** on the overall training step performance. **Memory-bound layers** are most likely to take a significant amount of time in small networks where there are no large and computation-heavy layers to dominate performance.

https://docs.nvidia.com/deeplearning/performance/dl-performance-memory-limited/index.html

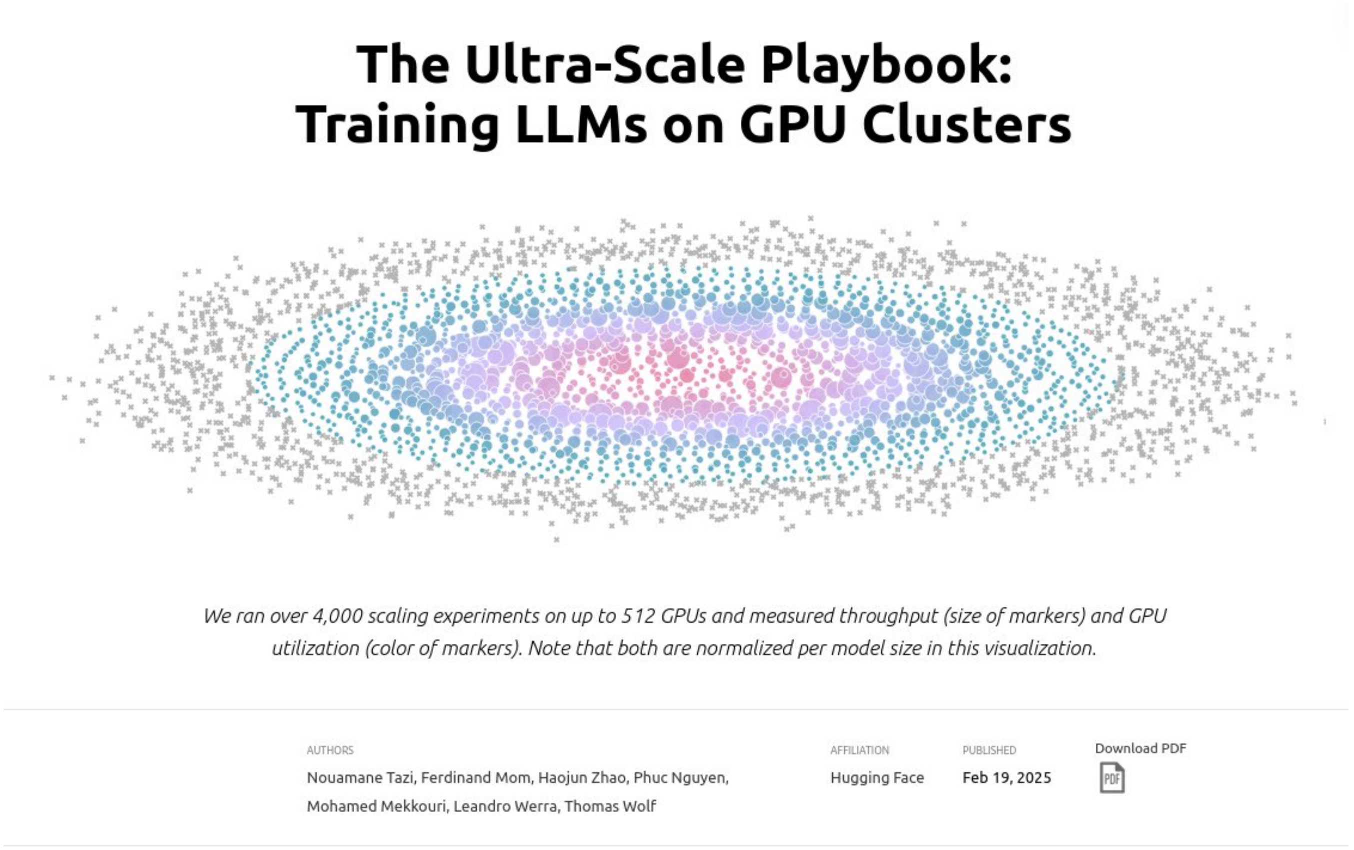
torch.compile, the missing manual

Models tend to fall into one of 3 different regimes:

1. **It just works.** torch.compile friendly (e.g., gpt-fast, torchao).
2. **It works with a little work.** able to get to torch.compile with minimal investment.
3. **It's going to be a slog.** expect to spend a lot of time working with the PyTorch team fixing bugs.

[torch.compile, the missing manual]

The Ultra-Scale Playbook



The Ultra-Scale Playbook:
Training LLMs on GPU Clusters

We ran over 4,000 scaling experiments on up to 512 GPUs and measured throughput (size of markers) and GPU utilization (color of markers). Note that both are normalized per model size in this visualization.

AUTHORS
Nouamane Tazi, Ferdinand Mom, Haojun Zhao, Phuc Nguyen, Mohamed Mekkouri, Leandro Werra, Thomas Wolf

AFFILIATION
Hugging Face

PUBLISHED
Feb 19, 2025

Download PDF

The Ultra-Scale Playbook: Training LLMs on GPU Clusters