CS 696 Applied Large Language Models
Spring Semester, 2025
Doc 17 Assignment 2, Performance Issues
Mar 6, 2025

```
model_name = "mistralai/Mistral-7B-Instruct-v0.2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
            model_name,
            quantization_config=BitsAndBytesConfig(load_in_8bit=True),
            device_map="cuda")
```

```
model_name = "microsoft/Phi-3-mini-4k-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)

#Load default config and set number of attention heads to half of default. Key value heads is als
because it must match the number of attention heads
config = AutoConfig.from_pretrained(model_name)
config.num_attention_heads = 16
config.num_hidden_layers = 32
config.num_key_value_heads = 16

#Load model with custom config, which loads with no trained parameters
model = AutoModelForCausalLM.from_pretrained(model_name, config=config,
quantization_config=BitsAndBytesConfig(load_in_8bit=True), device_map="cuda")
del(config)
```

Memory	size (MB)
CPU	1559.30 MB
GPU	4722.39 MB

Time (s)	Average Time (s)
['5.89', '5.50', '5.51', '5.52', '5.52']	**5.59**

Memory	size (MB)
CPU	1564.88 MB
GPU	1374.19 MB

Time (s)	Average Time (s)
['2.47', '2.05', '2.06', '2.04', '2.05']	**2.13**

> Memory used: 7642165248 bytes
>
> Time: 0:00:17.815929

```
# Define Attention Head Pruning Function
def prune_attention_heads(layer, heads_to_prune):
    """ Zeroes out the weights of specified attention heads in a Llama model layer. """
    self_attn = layer.self_attn  # Get the attention module

    total_heads = self_attn.q_proj.weight.shape[0]
    head_size = self_attn.q_proj.weight.shape[1]

    for head in heads_to_prune:
        start_idx = head * head_size
        end_idx = (head + 1) * head_size

        # Move tensors to `device`
        self_attn.q_proj.weight.data[start_idx:end_idx, :].to(device)  # Q
        self_attn.k_proj.weight.data[start_idx:end_idx, :].to(device)  # K
        self_attn.v_proj.weight.data[start_idx:end_idx, :].to(device)  # V
        self_attn.o_proj.weight.data[:, start_idx:end_idx].to(device)  # Output projection
```

```
print("Initial Memory usage:", psutil.Process().memory_info().rss / (1024 * 1024), "MB")
print("Initial VRAM usage:", torch.cuda.memory_allocated(torch.device('cuda:0')) / (1024 * 1024), "MB")

prompt = "Generate 5 unconventional project ideas for an Applied Large Language Model."
inputs = tokenizer(prompt, return_tensors="pt").to(device)

pad_token_id = tokenizer.pad_token_id if tokenizer.pad_token_id else tokenizer.eos_token_id
pad_token_id = torch.tensor(pad_token_id, device=device).item()

start_time = time.time()
outputs = model.generate(
    inputs["input_ids"],
    max_length=200,
    do_sample=True,
    pad_token_id=pad_token_id,  # ✅ Ensure `pad_token_id` is moved to CUDA
    attention_mask=inputs["attention_mask"],
)
end_time = time.time()

generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print("\nGenerated Text:\n", generated_text)

# Print Final Memory Usage
print("Final Memory usage:", psutil.Process().memory_info().rss / (1024 * 1024), "MB")
print("Final VRAM usage:", torch.cuda.memory_allocated(torch.device('cuda:0')) / (1024**2), "MB")
```

Pruning Strategy	Initial Memory Usage (MB)	Initial VRAM Usage (MB)	Final Memory Usage (MB)	Final VRAM Usage (MB)	Time Taken (s)	Generation Quality
Reduce 2 Heads	1866.83	6136.47	1871.79	6136.46	7.13	Readable, diverse ideas
Reduce Half Heads	1875.82	6136.46	1876.04	6136.46	7.09	Readable, structured ideas
Reduce to 1 Head	1876.04	6136.46	1876.11	6136.47	7.09	Readable, but slightly generic
Reduce Half Heads & Half Layers	1876.12	3448.30	1876.14	3448.30	4.06	Corrupted output, unreadable
Reduce Half Layers (Keep All Heads)	1876.14	2296.23	1876.14	2296.24	2.54	Corrupted output, unreadable

```
1. Identify the target audience: Determine the demographic group or group of individuals who are most likely to be affected by the disease. This may include women, women aged 50 or older, women aged 75 or older, or women aged 75 and above.

2. Research the target audience: Research the demographic group and their health care providers, health providers, and other relevant stakeholders. This will help you understand their preferences, preferences, and access to resources.

3. Identify the target audience's health care providers: Identify the health care providers in the target audience, including doctors, nurses, doctors, doctors, doctors,
Final Memory usage: 2502.29296875 MB
Final VRAM usage: 12656.5869140625 MB
Time Elapsed: 6.402444362640381 s
```

```python
from transformers import AutoModelForCausalLM, AutoTokenizer, AutoConfig
import torch
import time
import psutil

# Start time for tracking execution duration
start = time.time()

# Load the original model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("ministral/Ministral-3b-instruct")

# Load the model's configuration
config = AutoConfig.from_pretrained("ministral/Ministral-3b-instruct")

# Modify the configuration to reduce the number of layers and attention heads
config.num_hidden_layers = config.num_hidden_layers // 2  # Decrease the number of hidden layers by 1/2
config.num_attention_heads = config.num_attention_heads // 2  # Decrease the number of attention heads by 1/2
```

```
1. Identify the target audience: Determine the demographic group or group of individuals who are most likely to be affected by the
disease. This may include women, women aged 50 or older, women aged 75 or older, or women aged 75 and above.

2. Research the target audience: Research the demographic group and their health care providers, health providers, and other relev
ant stakeholders. This will help you understand their preferences, preferences, and access to resources.

3. Identify the target audience's health care providers: Identify the health care providers in the target audience, including doct
ors, nurses, doctors, doctors, doctors,
Final Memory usage: 2502.29296875 MB
Final VRAM usage: 12656.5869140625 MB
Time Elapsed: 6.402444362640381 s
```

Run with 1/2 layers and heads

```python
from transformers import AutoModelForCausalLM, AutoTokenizer, AutoConfig
import torch
import time
import psutil

# Start time for tracking execution duration
start = time.time()

# Load the original model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("ministral/Ministral-3b-instruct")

# Load the model's configuration
config = AutoConfig.from_pretrained("ministral/Ministral-3b-instruct")
```

3. Identify the target audience's health care providers: Identify the health care providers in the target audience, including doctors, nurses, doctors, doctors, doctors, Tokenization Time: 0.0003 seconds Generation Time: 3.0071 seconds Decoding Time: 0.0003 seconds Final Memory usage: 10289.5234375 MB Final VRAM usage: 12656.5869140625 MB Total time taken: 0.0021 seconds

Processing time was reasonable and I did not run into issues. I found it interesting that the text generated was focused on women and womens health as a specific example. There were also some redundancies in wording, but overall the model provided an ouput that most can understand.

Once I adjusted the hidden layers and attention heads, the model had issues with processing the input. See below the output that was generated.

Generate creative project ideas for a fine-tuning approach for cancer risk prediction.razrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazrazraz suggested Tokenization Time: 0.0003 seconds Generation Time: 1.5095 seconds Decoding Time: 0.0003 seconds Final Memory usage: 7870.62890625 MB Final VRAM usage: 6384.36474609375 MB Total time taken: 0.0018 seconds

Model Name	Initial RAM	Final RAM	Initial VRAM	Final VRAM	Execution Time
LLAMA-3.1-8B-Instruct	579.44 MB	1629.3 MB	0.0 MB	8679.11 MB	25.87 s
Mistral-7B-Instruct-v0.2	577.79 MB	1530.89 MB	0.0 MB	7363.71 MB	32.41 s
Phi-3-Mini-4K-Instruct	581.82 MB	1539.64 MB	0.0 MB	3878.21 MB	13.40 s
Phi-3-Mini-4K-Instruct w/ 16AH	581.79 MB	1515.39 MB	0.0 MB	3844.8 MB	13.53 s
Phi-3-Mini-4K-Instruct w/ 16HL	581.09 MB	1458.09 MB	0.0 MB	2130.81 MB	6.98 s
Phi-3-Mini-4K-Instruct w/ 16AH&HL	582.04 MB	1444.13 MB	0.0 MB	2130.72 MB	6.88 s

Generate the model and print resource utilization

```
model = AutoModelForCausalLM.from_config(
    config,
    torch_dtype=torch.float16
)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)

if torch.cuda.is_available():
    print("Allocated memory in bytes: ",  torch.cuda.memory_allocated())
    print("Cached memory in bytes:", torch.cuda.memory_reserved())
    print("Max memory allocated in bytes:", torch.cuda.max_memory_allocated())

print(model)
```

```
Allocated memory in bytes:  4018347520
Cached memory in bytes: 4020240384
Max memory allocated in bytes: 4018347520
Phi3ForCausalLM(
  (model): Phi3Model(
    (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
    (layers): ModuleList(
      (0-15): 16 x Phi3DecoderLayer(
        (self_attn): Phi3Attention(
          (o_proj): Linear(in_features=3072, out_features=3072, bias=False)
          (qkv_proj): Linear(in_features=3072, out_features=9216, bias=False)
```

```python
model.config.num_hidden_layers = 16
# model.model.layers = model.model.layers[1:32]

# Remove the inner 16 layers from the model
for i in range(16):
    model.model.layers.pop(8) # Continuously pops only the inner layers
print(model)
if torch.cuda.is_available():
    print("Allocated memory in bytes: ",  torch.cuda.memory_allocated())
    print("Cached memory in bytes:", torch.cuda.memory_reserved())
    print("Max memory allocated in bytes:", torch.cuda.max_memory_allocated())
```

```
start = time.time()

model = AutoModelForCausalLM.from_pretrained(model_name, config=config, load_in_8bit=T
device_map="cuda")
del(config)

input_text = "Generate ideas for projects involving a Large Language Model."
inputs = tokenizer(input_text, return_tensors="pt").to('cuda')
outputs = model.generate(inputs['input_ids'], max_length=1000, do_sample=True)
generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(generated_text)

total_time_2 = time.time() - start
```

Performance Issues

https://levelup.gitconnected.com/how-to-train-your-pytorch-models-much-faster-14737c8c9770

Automatic Mixed Precision Training

	Sign - bits	Exponent Bits	Mantissa (fraction) Bits
FP32	1	8	32
FP16	1	5	10
BF16	1	8	7

FP16 & BF16
- Perform some operations faster
- Use less space
- Lose some accuracy
- Can cause instabilty in calculations

Hardware support
- NVIDIA A100, H100 (Ampere) FP16, BF16
- Prior NVIDA FP16
- Google TPUs FP16, BF16
  - Matrix multiplication uses BF16

Automatic Mixed Precision Training

During training:

Source — Sebastian Raschka's Blog

Automatic Mixed Precision Training - autocast

torch.cuda.amp.autocast()

Allow regions of your script to run in mixed precision

Automatically chooses the best-suited numerical precision (FP16/BF16 vs. FP32) for each operation within its context, speeding up computations on GPUs while preserving accuracy.

float64 or non-floating-point dtypes are not eligible

In-place variants and calls that explicitly supply an out=...

a.addmm(b, c) can autocast,

a.addmm_(b, c) and a.addmm(b, c, out=d) cannot

Automatic Mixed Precision Training - Gradient Scaling

Gradients can be small

- Maybe too small for FP16

So increase the gradients on backtracking to avoid underflow

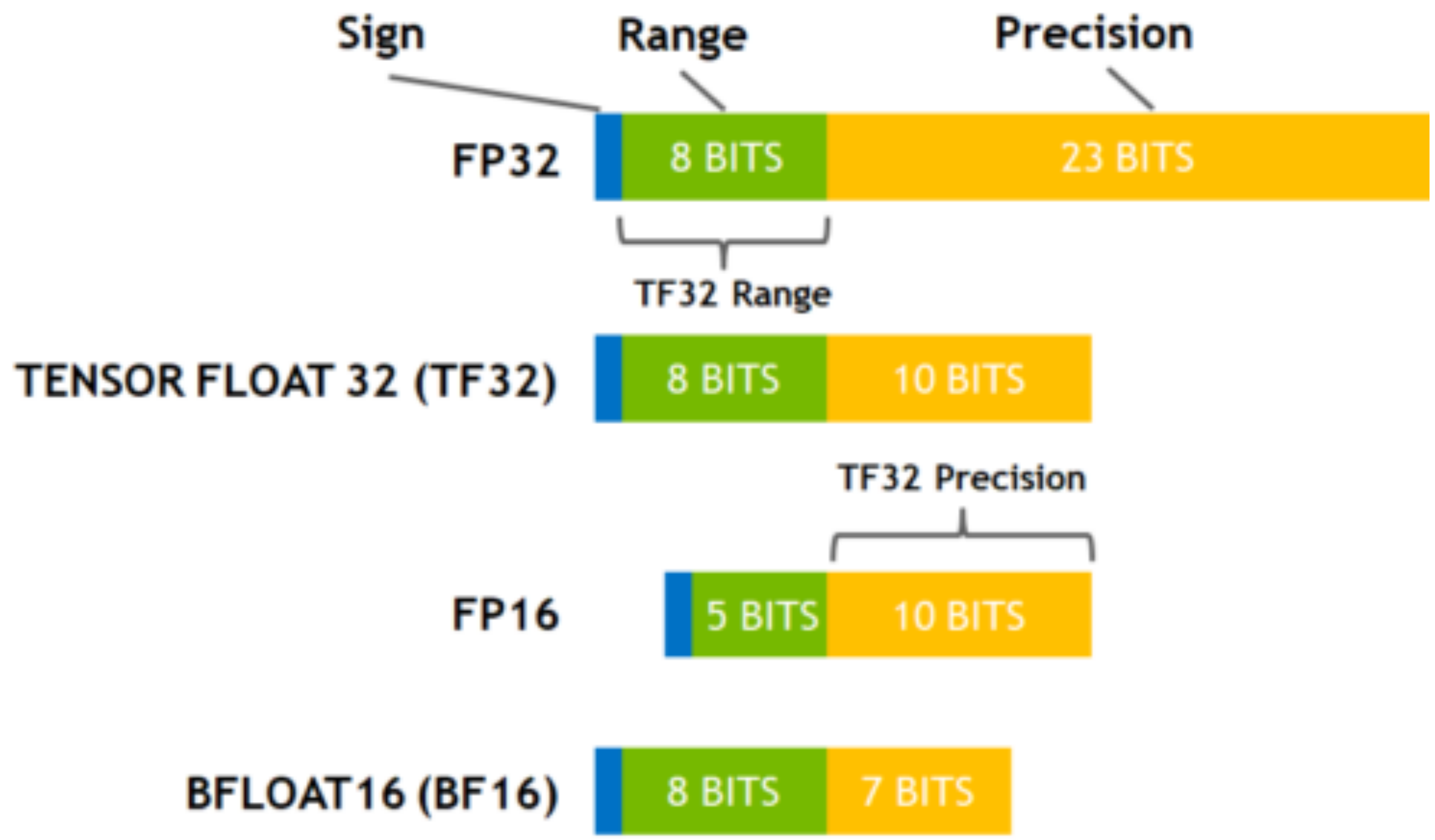
Default values

- scale 65536
- growth_factor 2.0
- growth_interval 20000

Automatic Mixed Precision Training

```
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting.
        with autocast(device_type='cuda', dtype=torch.float16):
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
        # Backward passes under autocast are not recommended.
        # Backward ops run in the same dtype autocast chose for corresponding forward ops.
        scaler.scale(loss).backward()

        # scaler.step() first unscales the gradients of the optimizer's assigned params.
        # If these gradients do not contain infs or NaNs, optimizer.step() is then called,
        # otherwise, optimizer.step() is skipped.
        scaler.step(optimizer)

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

Mixed precision training



Mixed precision training

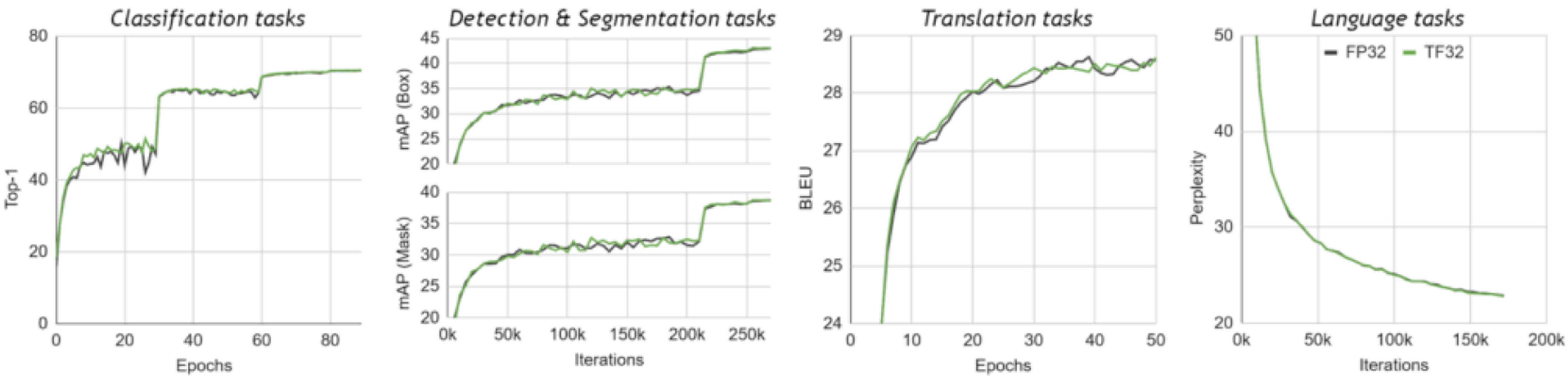
Can reduce precision to save memory with little loss in accuracy

```
training_args = TrainingArguments(per_device_train_batch_size=4, fp16=True, **default_args)
```

If you have Ampere hardware, use bf16

```
training_args = TrainingArguments(bf16=True, **default_args)
```

Mixed precision training - TF32



FP32 - Black

TF32 - Green

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

Mixed precision training - TF32

```
import torch
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True
```

CUDA will automatically switch to using tf32 instead of fp32 where possible

Or always use tf32

```
TrainingArguments(tf32=True, **default_args)
```

Requires Nvidia Ampere GPU

FP16 is 2x faster than TF32

Gradient Clipping

Limit (or “clip”) the magnitude of gradients during backpropagation

Prevents Gradient Explosions

- Large values can cause the model parameters to swing wildly

Improving Training Stability

- Large updates from outlier batches can cause instability in optimization

With Gradient Clipping

```
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()
        with autocast(device_type='cuda', dtype=torch.float16):
            output = model(input)
            loss = loss_fn(output, target)
        scaler.scale(loss).backward()

        # Unscales the gradients of optimizer's assigned params in-place
        scaler.unscale_(optimizer)

        # Since the gradients of optimizer's assigned params are unscaled, clips as usual:
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm)

        # optimizer's gradients are already unscaled, so scaler.step does not unscale them,
        # although it still skips optimizer.step() if the gradients contain infs or NaNs.
        scaler.step(optimizer)

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

Profiling

```
import torch.profiler

with torch.profiler.profile(
    schedule=torch.profiler.schedule(wait=1, warmup=1, active=3),
    on_trace_ready=torch.profiler.tensorboard_trace_handler('./log'),
    record_shapes=True,
    with_stack=True
) as prof:
    for inputs, targets in dataloader:
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        prof.step()
```

torch.profiler.profile Arguments

activities
which device activities to trace

activities=[torch.profiler.ProfilerActivity.CPU,
torch.profiler.ProfilerActivity.CUDA]

schedule
many steps to warm up,
how many steps to record, and
how many steps to skip in a repeating cycle

record_shapes
record input shapes of each profiled operation

with_stack
If True, captures Python stack traces for each operation

with_modules
associates profiler events with your Module hierarchy

Speed Up Your DataLoader

```
from torch.utils.data import DataLoader

dataloader = DataLoader(
    dataset,
    batch_size=64,
    shuffle=True,
    num_workers=4,         # Use as many workers as your CPU cores allow
    pin_memory=True,       # Speeds up data transfer to the GPU
    prefetch_factor=2      # Preload batches (only after PyTorch v1.8.0)
)
```

Parameters

shuffle

Whether to shuffle the data at the start of each epoch

True for training

False for validation/test loaders

batch_size

Number of samples in each mini-batch

pin_memory

Allocates tensors in pinned (page-locked) memory,

Faster host-to-device (CPU to GPU) transfers

But need enough RAM

prefetch_factor

Number of batches loaded in advance by each worker

num_workers Fine Print

Memory Used = number of workers * size of parent process

So, with a large dataset, you could have memory issues

shuffle=True exacerbates the memory issue

Simplest workaround

    Replace Python objects with Pandas, Numpy or PyArrow objects

persistent_workers=True

    If True, worker processes are not shut down after the end of an epoch

Static Compilation

```
compiled_model = torch.compile(
    model,
    backend="inductor",  # Which compiler backend to use (usually "inductor" for best performance)
    mode="default",
   dynamic=False        # If True, tries to handle dynamic shapes more gracefully (with overhead)
)
```

mode
    "default":
        Standard compilation with a good balance of speed and coverage.

    "reduce-overhead":
        May skip some optimizations if they add overhead.

    "max-autotune":
        Most aggressive mode,

Under The Hood

TorchDynamo

Intercepts Python bytecode to trace PyTorch operations.

AOT Autograd (Ahead-of-Time Autograd)

Re-compiles the forward + backward passes as a graph of lower-level ops.

Inductor (by default)

Generates highly optimized kernels from the graph

Works best

When a model has large matrix multiplications

On recent GPUs

Data Parallelism on a Single Machine

torch.nn.DataParallel

From the docs:

It is recommended to use DistributedDataParallel, instead

DistributedDataParallel Docs start with

9 Notes

8 Warnings

Gradient Accumulation

If not enough GPU memory

```
import torch
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(10, 1).cuda()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.MSELoss()

train_loader = ...  # Some DataLoader returning batches of size 16
accum_steps = 4     # We want 16 * 4 = 64 samples per update

for epoch in range(num_epochs):
    for i, (x, y) in enumerate(train_loader):
        x, y = x.cuda(), y.cuda()

        outputs = model(x)
        loss = criterion(outputs, y)
        loss = loss / accum_steps

        loss.backward()

        if (i + 1) % accum_steps == 0:
            optimizer.step()
            optimizer.zero_grad()
```

Deepspeed

Memory-efficient and fast distributed training

Microsoft Library, works with Huggingface

Eliminates memory redundancies in data- and model-parallel training

Low communication volume and high computational granularity

Potential to scale beyond 1 Trillion parameters

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Some Problems

1.5B parameter GPT-2 model requires 3GB of memory for its weights

- But needs more than 32GB to train on a single GPU

Up to 50% of training time can be spent on GPU-CPU-GPU transfers

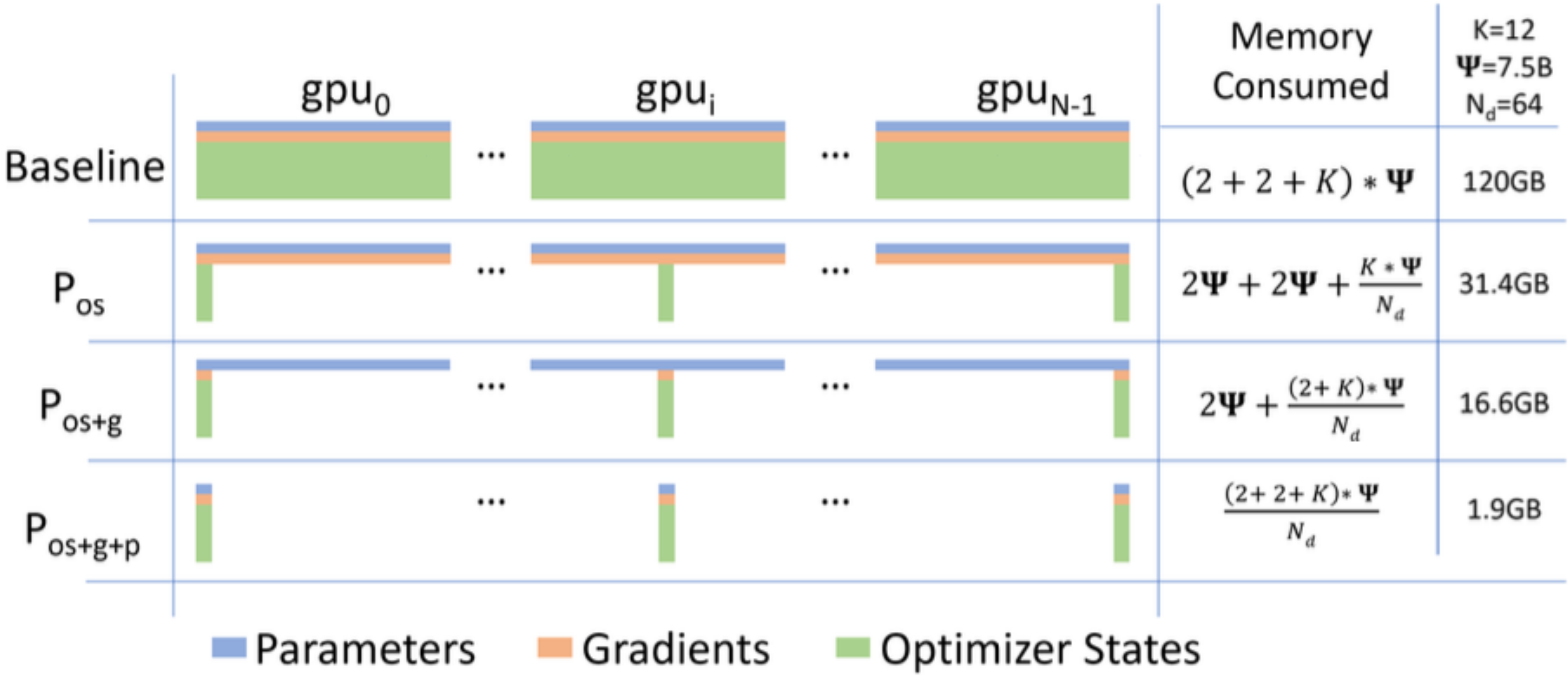
Zero Redundacy Optimizer (ZeRO)

ZeRO-1,
optimizer state partitioning across GPUs

ZeRO-2,
gradient partitioning across GPUs

ZeRO-3,
parameter partitioning across GPUs

Deepspeed - Memory

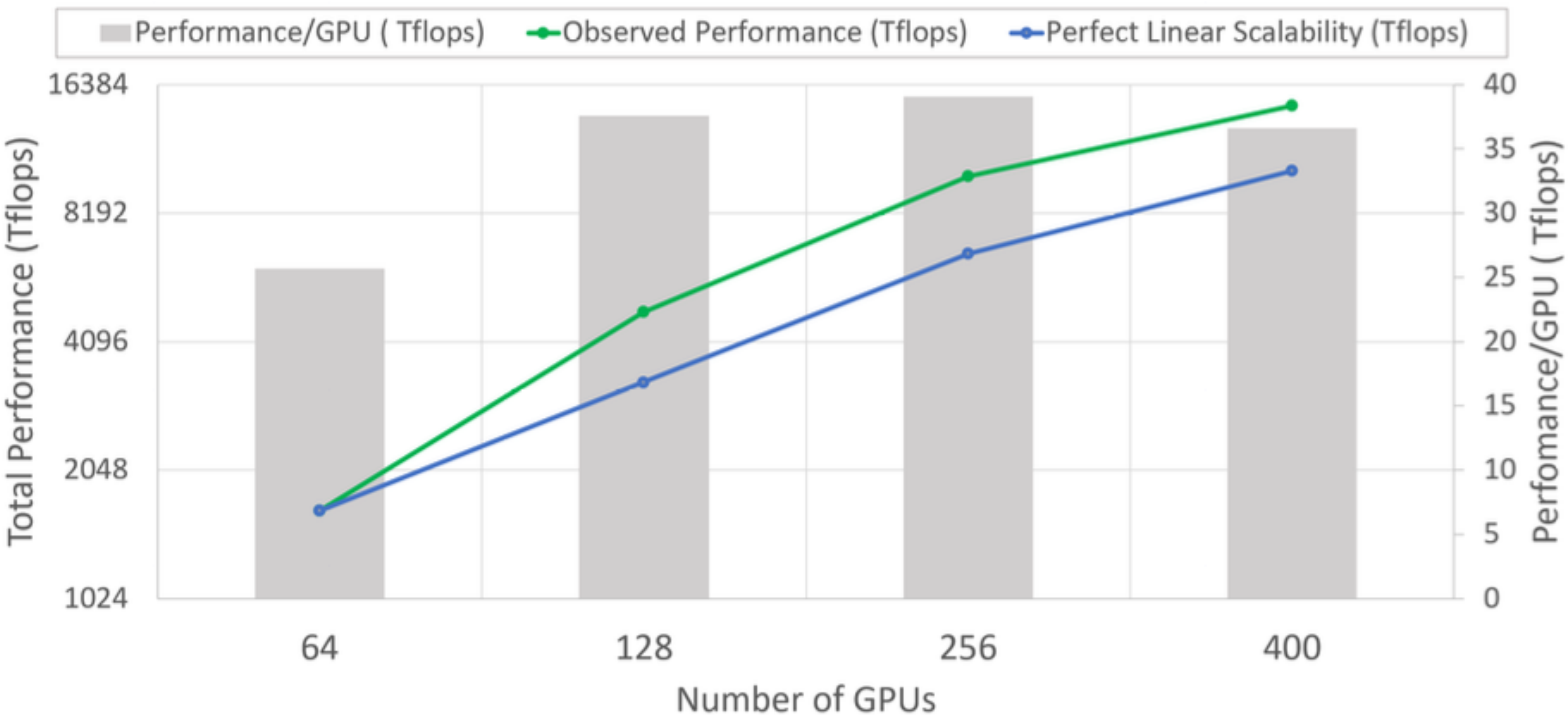


Ψ model size

K denotes the memory multiplier of optimizer states,

Nd denotes DP degree.

Deepspeed - Speedup



Per GPU training throughput of a 60B parameter model using ZeRO-100B

Using Deepspeed

```
model_engine, optimizer, _, _ = deepspeed.initialize(args=cmd_args,
                                                     model=model,
                                                     model_parameters=params)

deepspeed.init_distributed()


for step, batch in enumerate(data_loader):
    #forward() method
    loss = model_engine(batch)

    #runs backpropagation
    model_engine.backward(loss)

    #weight update
    model_engine.step()
```

Estimating Memory Requirements

```
from transformers import AutoModel;
from deepspeed.runtime.zero.stage3 import estimate_zero3_model_states_mem_needs_all_live
model = AutoModel.from_pretrained("bigscience/T0_3B")
estimate_zero3_model_states_mem_needs_all_live(model, num_gpus_per_node=4, num_nodes=1)
```

```
Estimated memory needed for params, optim states and gradients for a:
HW: Setup with 1 node, 4 GPUs per node.
SW: Model with 2783M total params, 65M largest layer params.
  per CPU  |  per GPU |   Options
   70.00GB |   0.25GB | offload_param=OffloadDeviceEnum.cpu, offload_optimizer=OffloadDeviceEnum.cpu, zero_init=1
   70.00GB |   0.25GB | offload_param=OffloadDeviceEnum.cpu, offload_optimizer=OffloadDeviceEnum.cpu, zero_init=0
   62.23GB |   1.54GB | offload_param=none, offload_optimizer=OffloadDeviceEnum.cpu, zero_init=1
   62.23GB |   1.54GB | offload_param=none, offload_optimizer=OffloadDeviceEnum.cpu, zero_init=0
    1.47GB |  11.91GB | offload_param=none, offload_optimizer=none, zero_init=1
   62.23GB |  11.91GB | offload_param=none, offload_optimizer=none, zero_init=0
```

unsloth https://unsloth.ai/

Library for finetuning & PEFT written by two brothers

2 to 5X faster that Huggingface

Uses less GPU memory

Hand-written, highly optimized CUDA kernel for the backward pass

Flash Attention 2

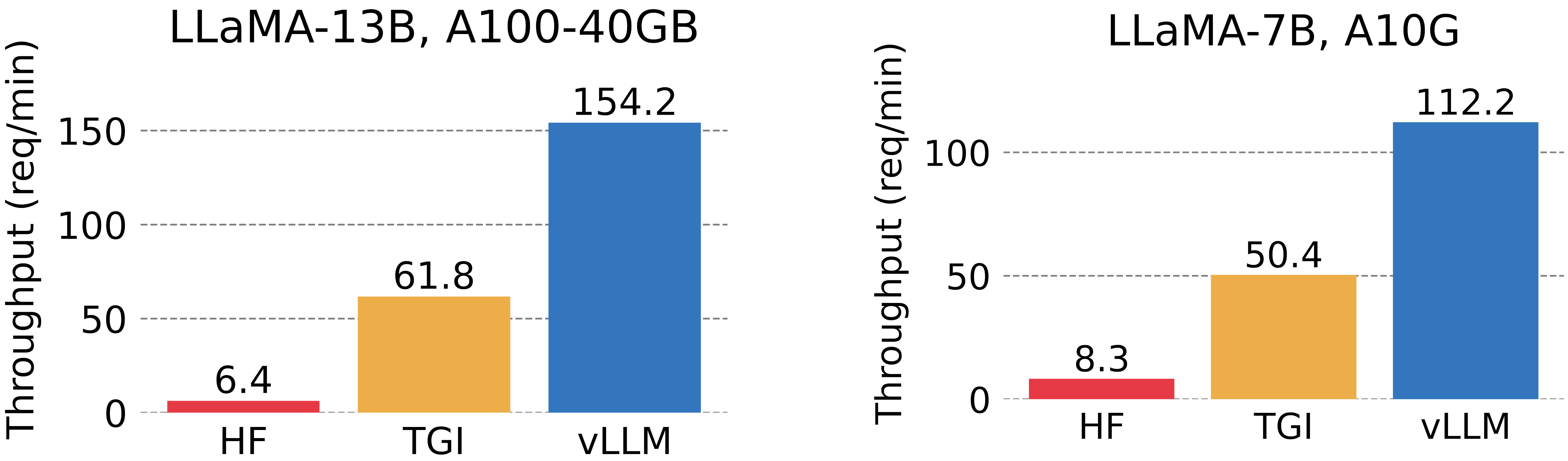
Work seamlessly with Hugging Face transformers library

```python
from unsloth import FastLanguageModel
import torch
max_seq_length = 1024 # Can increase for longer reasoning traces
lora_rank = 32 # Larger rank = smarter, but slower

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "meta-llama/meta-Llama-3.1-8B-Instruct",
    max_seq_length = max_seq_length,
    load_in_4bit = True, # False for LoRA 16bit
    fast_inference = True, # Enable vLLM fast inference
    max_lora_rank = lora_rank,
    gpu_memory_utilization = 0.6, # Reduce if out of memory
)

model = FastLanguageModel.get_peft_model(
    model,
    r = lora_rank, # Choose any number > 0 ! Suggested 8, 16, 32, 64, 128
    target_modules = [
        "q_proj", "k_proj", "v_proj", "o_proj",
        "gate_proj", "up_proj", "down_proj",
    ], # Remove QKVO if out of memory
    lora_alpha = lora_rank,
    use_gradient_checkpointing = "unsloth", # Enable long context finetuning
    random_state = 3407,
)
```

vLLM

Highly optimized engine for running LLMs

Serving throughput when each request asks for one output completion



https://blog.vllm.ai/2023/06/20/vllm.html

Problem - Attention key and value tensors

KV cache

- Large - 1.7GB for a single sequence in LLaMA-13B
- Dynamic:
  - Its size depends on the sequence length, which is highly variable and unpredictable

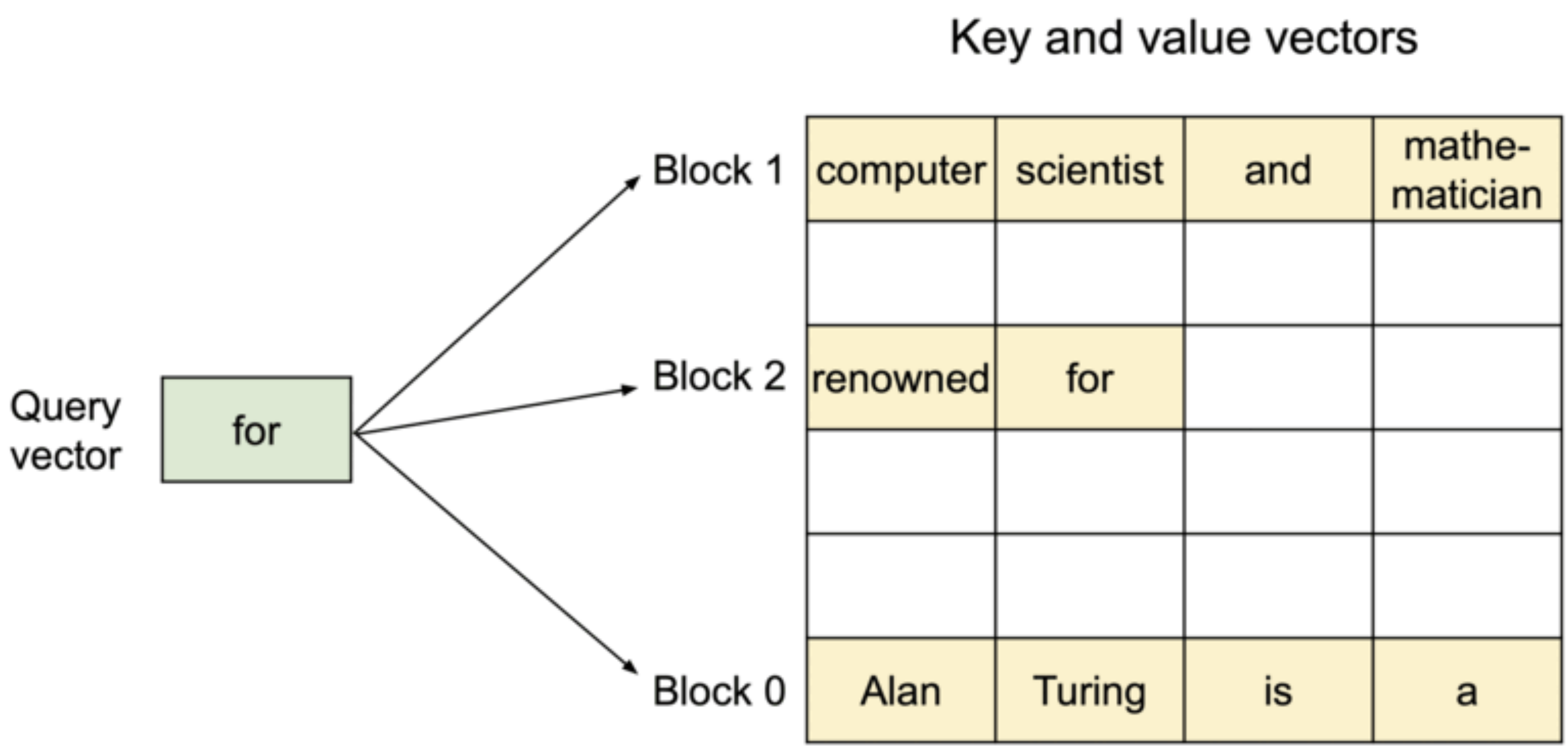
Existing systems waste 60% – 80% of memory due to fragmentation and over-reservation

PagedAttention - The Solution

Partition KV cache into blocks

Block contains keys and values for a fixed number of tokens

Blocks are in non-contiguous memory



Using vLLM

```
pip install vllm
```

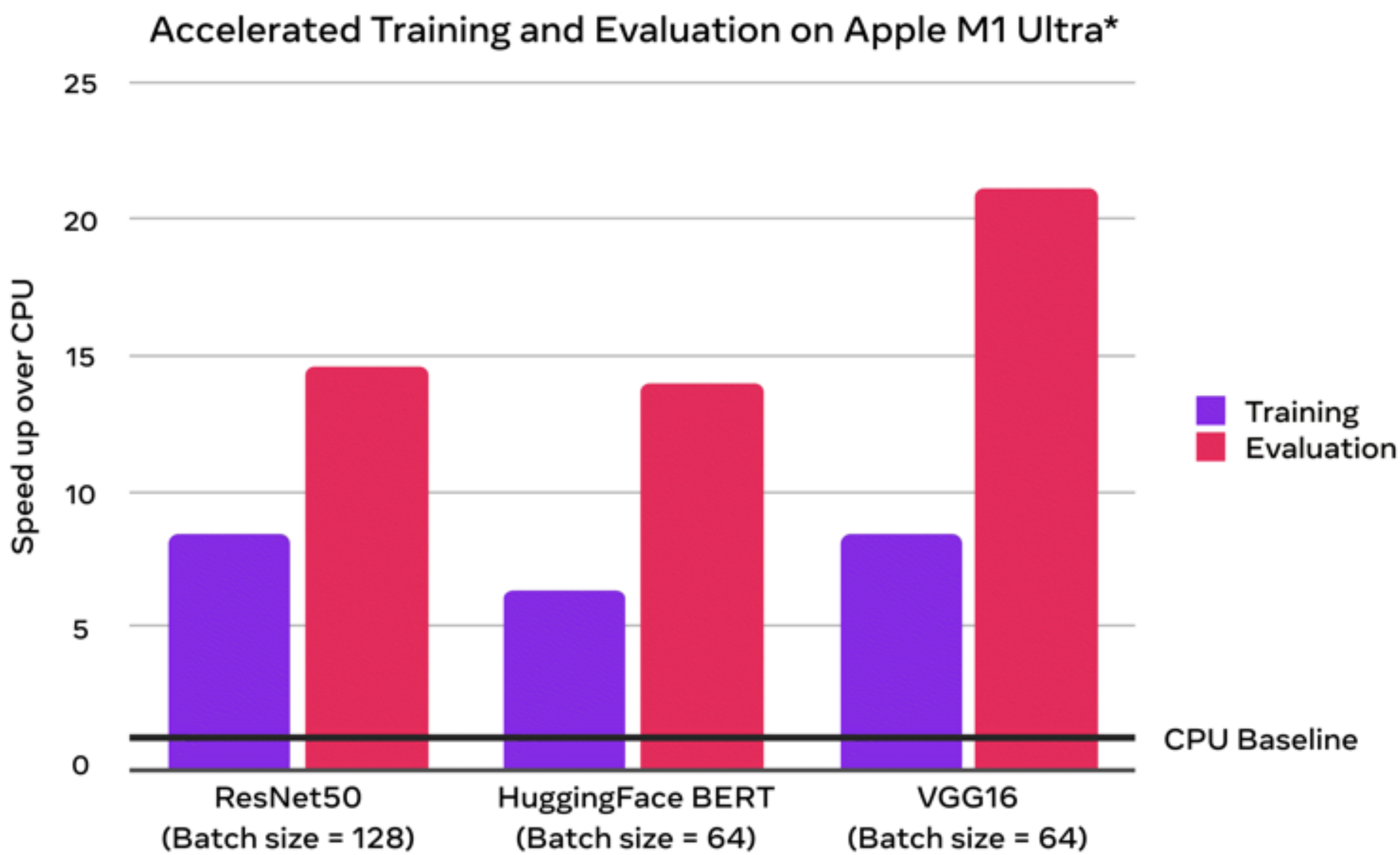
```
from vllm import LLM

prompts = ["Hello, my name is", "The capital of France is"]  # Sample prompts.
llm = LLM(model="lmsys/vicuna-7b-v1.3")  # Create an LLM.
outputs = llm.generate(prompts)  # Generate texts from the prompts.
```

TRL supports vLLM

https://huggingface.co/docs/trl/main/en/speeding_up_training?vllm+examples=GRPO#vllm-for-fast-generation-in-online-methods

```
from trl import GRPOConfig

training_args = GRPOConfig(..., use_vllm=True)
```

For Mac Users - MPS backend



For Mac Users - MPS backend

pip install torch torchvision torchaudio

```
# Check that MPS is available
if not torch.backends.mps.is_available():
    if not torch.backends.mps.is_built():
        print("MPS not available because the current PyTorch install was not "
              "built with MPS enabled.")
    else:
        print("MPS not available because the current MacOS version is not 12.3+ "
              "and/or you do not have an MPS-enabled device on this machine.")

else:
    mps_device = torch.device("mps")

    # Create a Tensor directly on the mps device
    x = torch.ones(5, device=mps_device)
    # Or
    x = torch.ones(5, device="mps")

    # Any operation happens on the GPU
    y = x * 2

    # Move your model to mps just like any other device
    model = YourFavoriteNet()
    model.to(mps_device)

    # Now every call runs on the GPU
    pred = model(x)
```

For Mac Users - MPS backend

Can only use 1 GPU

Some PyTorch operations are not implemented in MPS yet and will throw an error
Set the environment variable PYTORCH_ENABLE_MPS_FALLBACK=1

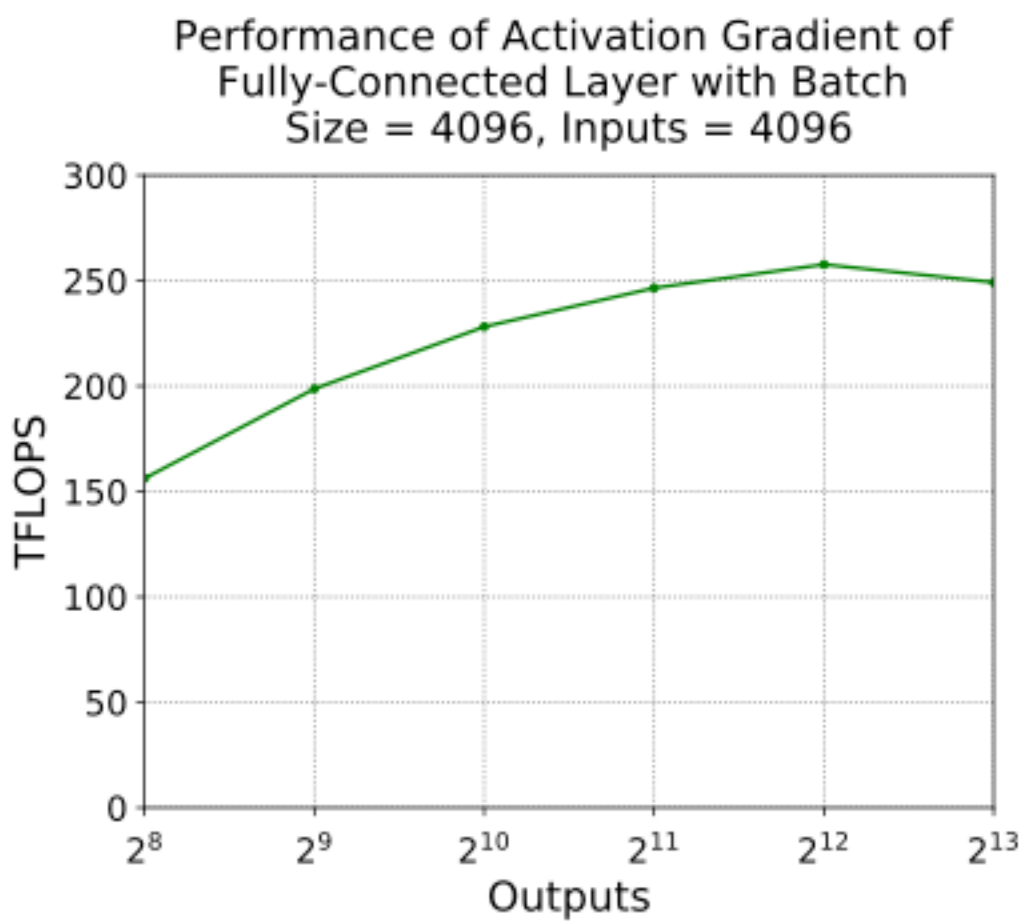
Huggingface Recommendations

Method/tool	Improves training speed	Optimizes memory utilization
Batch size choice	Yes	Yes
Gradient accumulation	No	Yes
Gradient checkpointing	No	Yes
Mixed precision training	Yes	Maybe*
torch_empty_cache_steps	No	Yes
Optimizer choice	Yes	Yes
Data preloading	Yes	No
DeepSpeed Zero	No	Yes
torch.compile	Yes	No
Parameter-Efficient Fine Tuning (PEFT)	No	Yes

Batch size & Layer Size

Batch sizes and input/output neuron counts use size 2^N.

Larger layers are more efficent to process

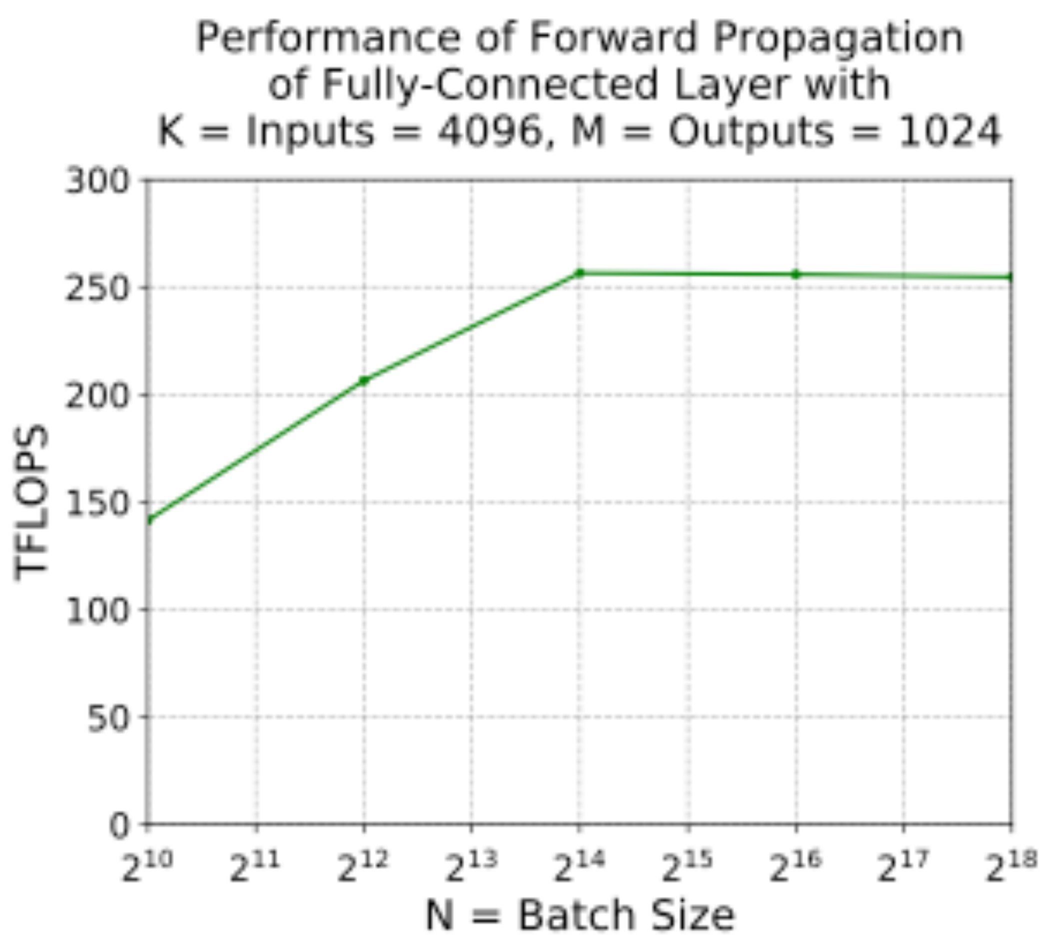


https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#input-features

Batch size & Layer Size

Batch sizes

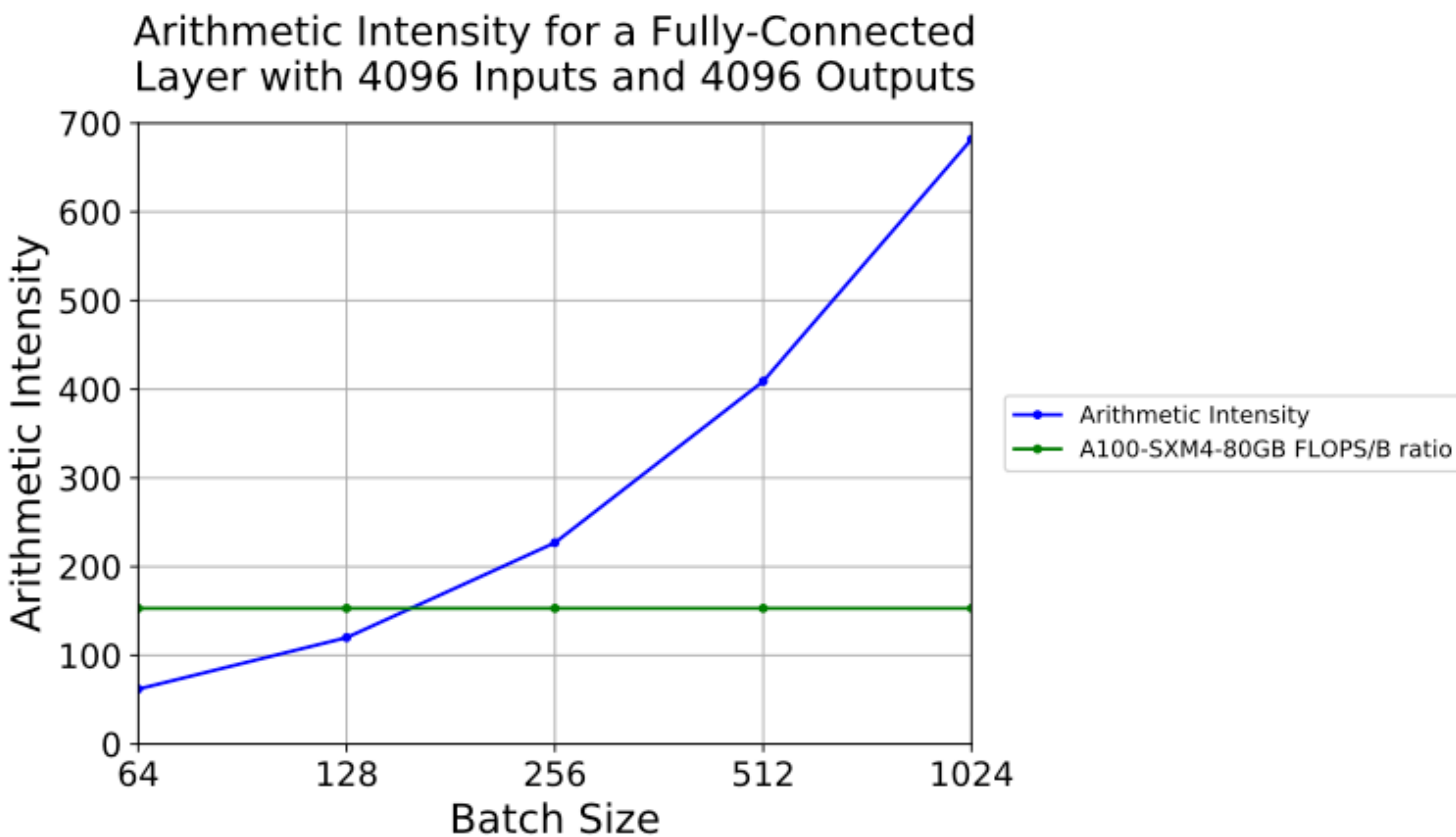
- Larger size more efficient
- Requires more memory



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

Batch size & Layer Size

Batch sizes 128 and below are bandwidth limited on NVIDIA A100 accelerators.



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

Gradient Accumulation

Calculate gradients in smaller increments due to memory constraints

```
training_args = TrainingArguments(
                    per_device_train_batch_size=1,
                    gradient_accumulation_steps=4, **default_args)
```

Gradient Checkpointing

Activations from the forward pass consume a lot of memory

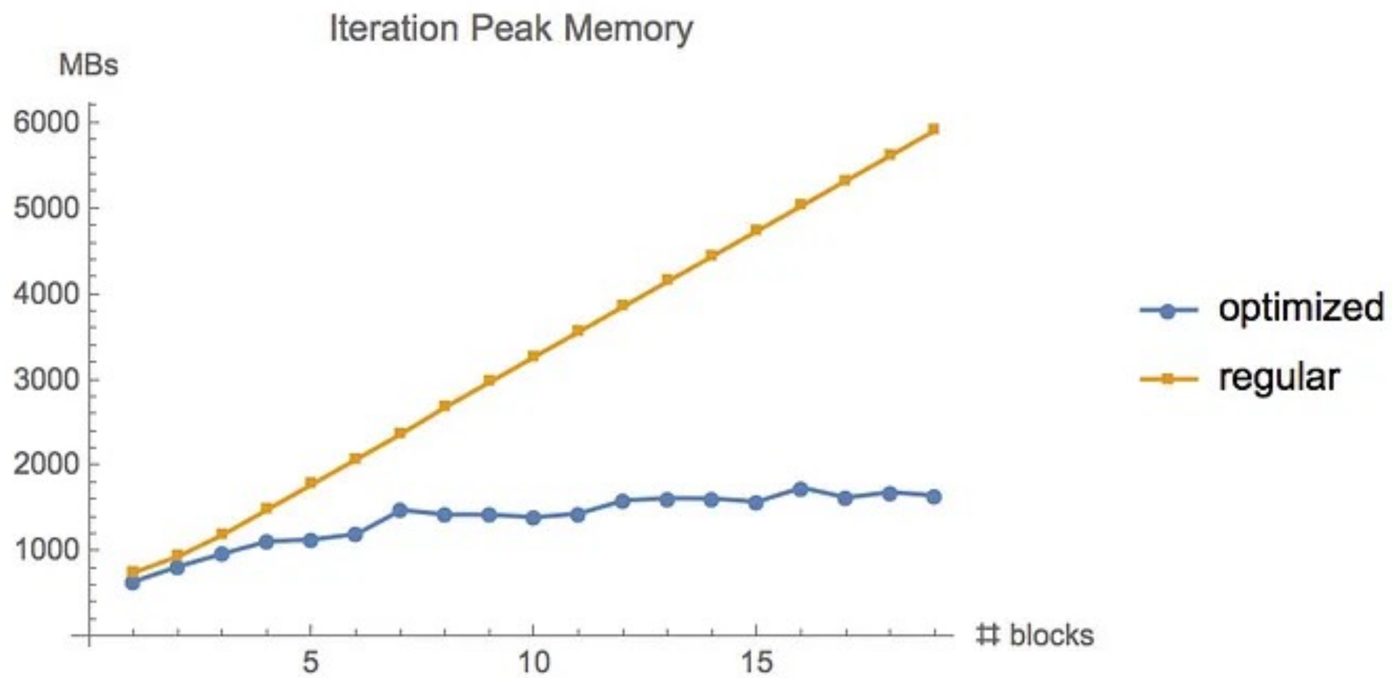
Deleting them and recomputing in the backward pass
- Saves memory but slows down backward pass

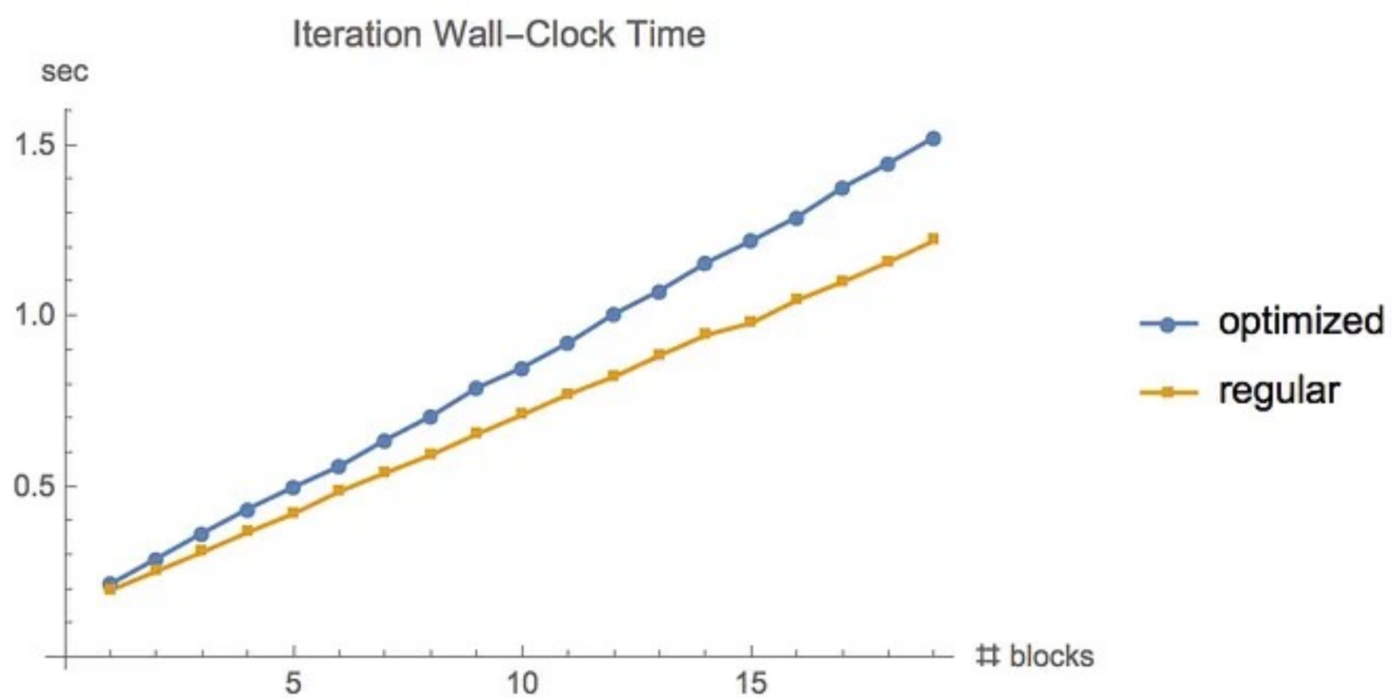
Gradient checkpointing
- Saves strategically selected activations
- Only a fraction of the activations need to be re-computed for the gradients.

```
training_args = TrainingArguments(
                    per_device_train_batch_size=1,
                    gradient_accumulation_steps=4,
                    gradient_checkpointing=True,
                    **default_args
)
```

Gradient Checkpointing

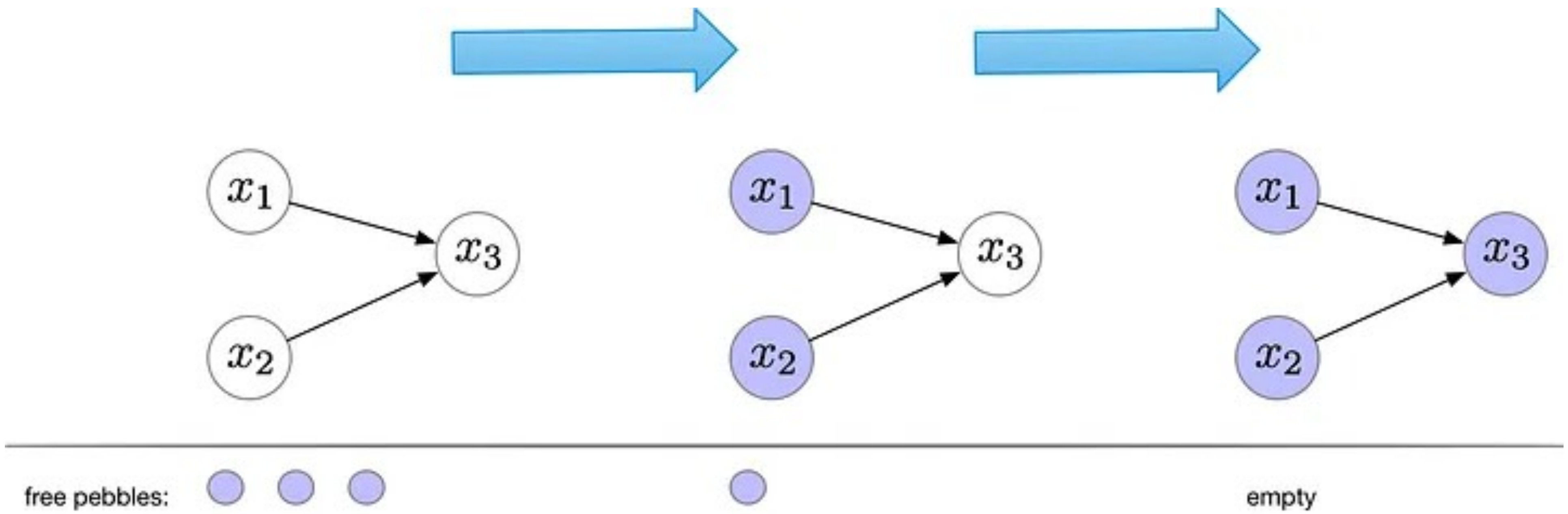
batch size = 1280



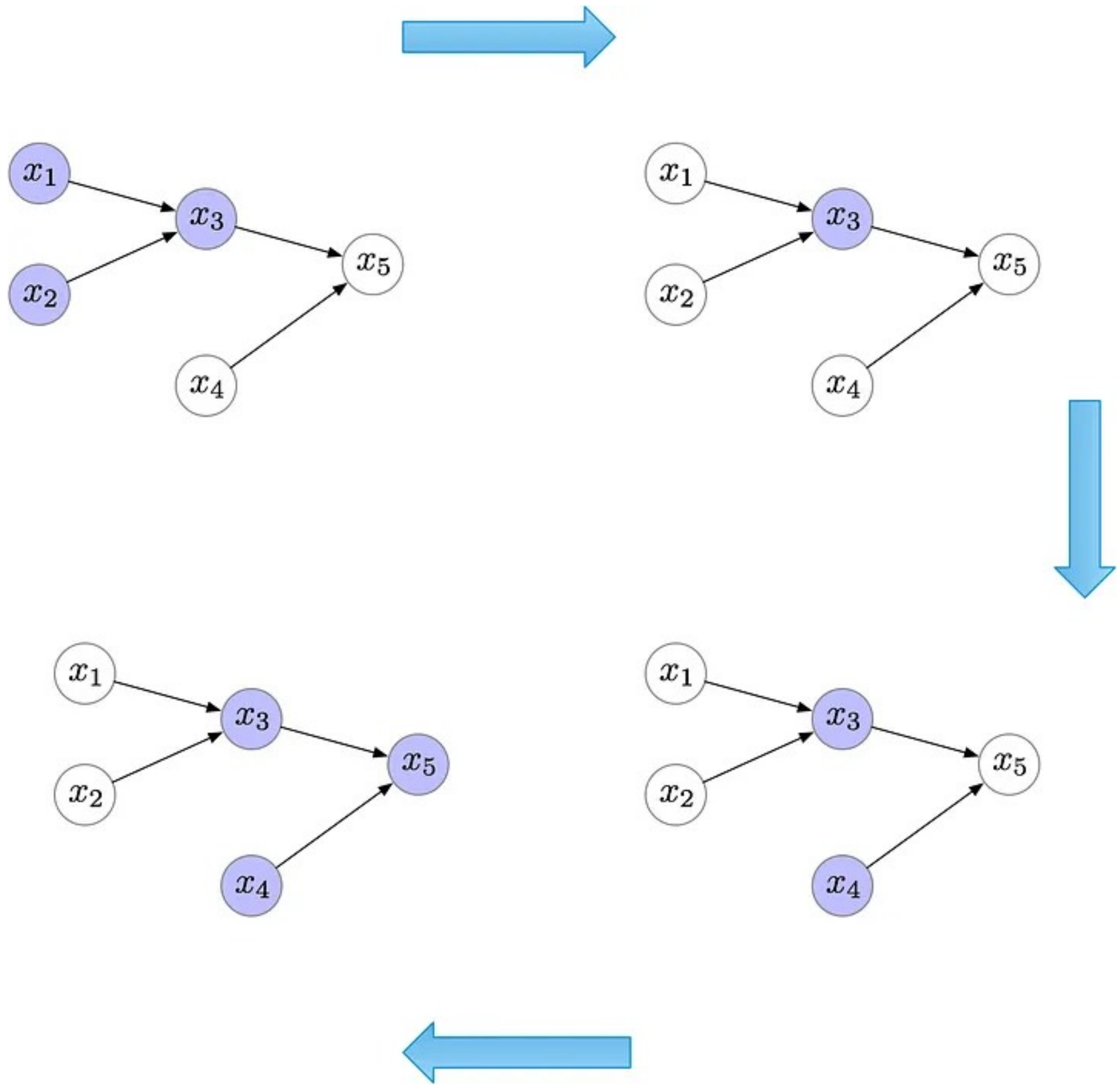


https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9

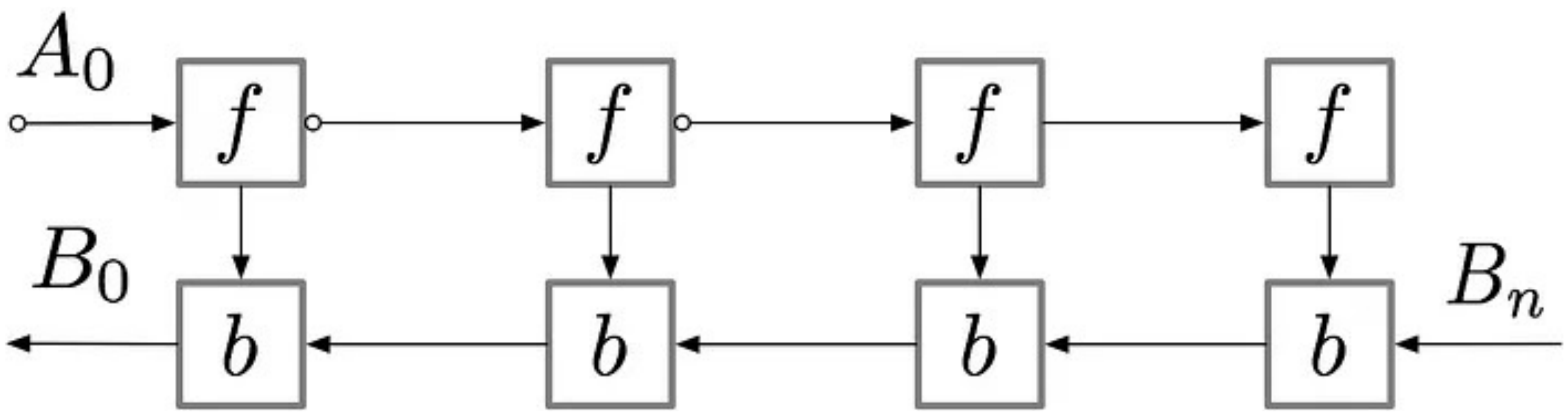
Pebble Analogy



Pebble Analogy



Gradient Computation



Checkpoints every sqrt(n) steps

Memory Requirement	$O(\sqrt{n})$
Compute Requirement	$O(n)$
Forward calcs per node	1 to 2

https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9

FlashAttention-2

Additionally parallelizing the attention computation over sequence length

Partitioning the work between GPU threads to reduce communication and shared memory reads/writes

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, LlamaForCausalLM

model_id = "tiiuae/falcon-7b"
tokenizer = AutoTokenizer.from_pretrained(model_id)

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    torch_dtype=torch.bfloat16,
    attn_implementation="flash_attention_2",
)
```

model's dtype must be fp16 or bf16