Optimizing Your PyTorch Code: Unlocking Performance and Efficiency

A Guide to Boosting Speed and Resource Usage

Hovhannes Tamoyan

DataLoader

Multi-process Data Loading

- DataLoader uses a single-process by default.
- The GIL prevents true fully parallelizing Python code across threads -> blocking computation code with data loading.
- To perform multi-process data loading set the argument num_workers=R_+.

More: https://pytorch.org/docs/stable/data.html

DataLoader

Multi-process Data Loading

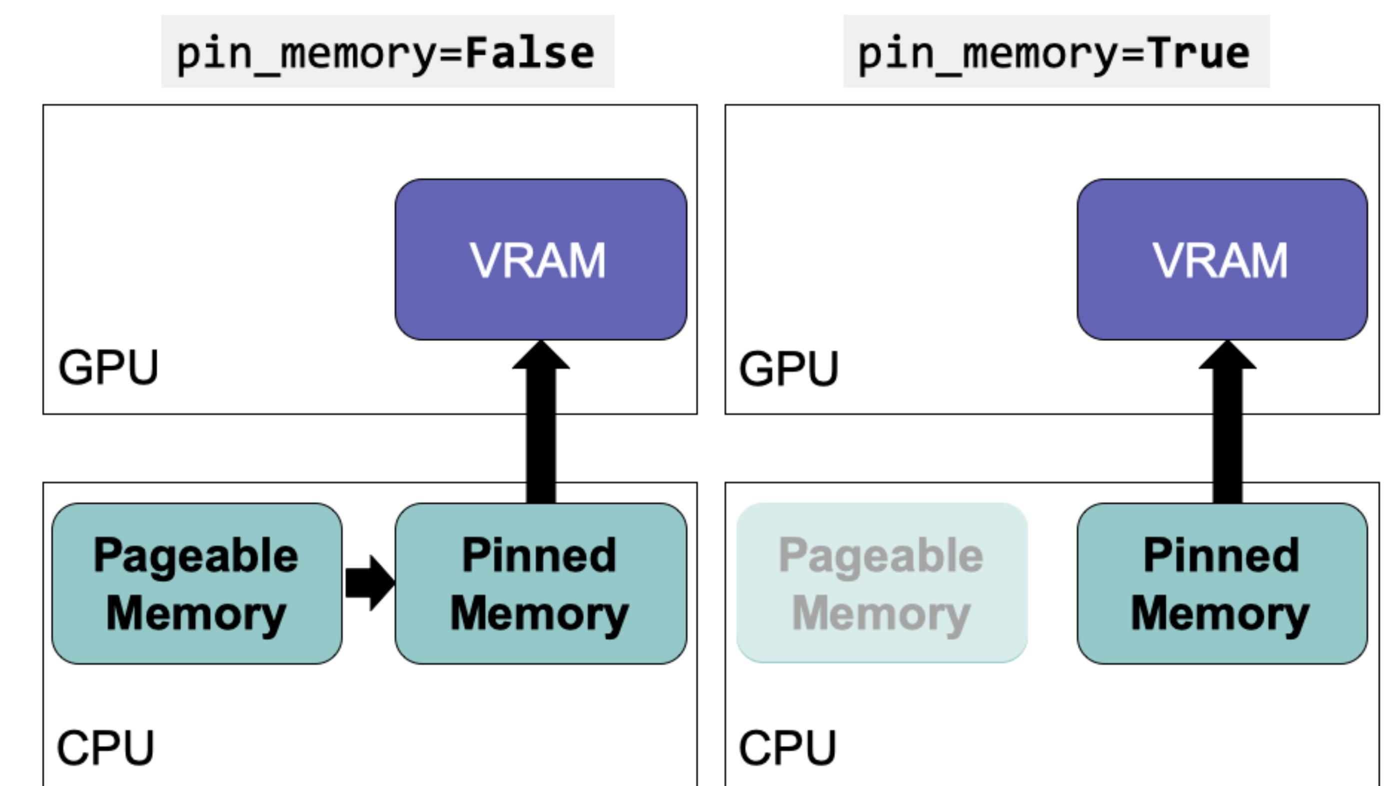
- After several iterations, the loader worker processes will consume the same amount of CPU memory as the parent process for all Python objects in the parent process which are accessed from the worker processes.
- This can be problematic if the Dataset contains a lot of data (e.g., you are loading a very large list of filenames at Dataset construction time) and/or you are using a lot of workers (overall memory usage is number of workers * size of parent process).
- The simplest workaround is to replace Python objects with non-refcounted representations such as Pandas, Numpy or PyArrow objects.

More: https://github.com/pytorch/pytorch/issues/13246

DataLoader

Memory Pinning

- To speed up the host the Dataset transfer from CPU to GPU enable pin_memory.
- This lets the DataLoader allocate the samples in page-locked/ pinned memory, which speeds-up the transfer to GPU.
- To put the fetched data tensors in pinned memory set pinn_memory=True



More: https://pytorch.org/docs/stable/data.html

DistributedDataParallel

DistributedDataParallel

- DistributedDataParallel uses multiprocessing where a process is created for each GPU
- DataParallel uses multithreading.
- During multiprocessing, each GPU has its dedicated process, this avoids the performance overhead caused by GIL.
- Recommended to use DistributedDataParallel, instead of DataParallel to do multi-GPU training, even if there is only a single node.
- Use torch.distributed.launch utility to launch your program utilizing DistributedDataParallel.

More: https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html

More: https://pytorch.org/docs/stable/distributed.html

Gradients

Inference Mode

- Inference code run under this mode gets better performance by disabling view tracking and version counter bumps
- Make sure your operations will have no interactions with autograd
- Note that unlike some other mechanisms that locally enable or disable grad, entering inference_mode also disables forward-mode AD.

More: https://pytorch.org/docs/stable/generated/torch.inference_mode.html

Gradients

No grad Mode

```
with torch.no_grad():
    x = torch.randn(1)
    y = x + 1

y.requires_grad = True
z = y + 1
print(z.grad_fn)
```

> <AddBackward0 object at 0x7fe9c6eafdf0>

Inference Mode

```
with torch.inference_mode():
    x = torch.randn(1)
    y = x + 1

y.requires_grad = True
```

> RuntimeError: Setting requires_grad=True on inference tensor outside InferenceMode is not allowed.

Gradients

Set grad to None

- Pass an additional argument set_to_none=True when calling optimizer.zero_grad() to set the grade to None and not 0.
- Leads to a lower memory footprint and modestly faster performance.
- Caveats apply:
  - When the user tries to access a gradient and perform manual ops on it, a None attribute or a Tensor full of 0s will behave differently.
  - The operation followed by a backward pass, guarantees the .grads to be None for params that did not receive a gradient.
  - torch.optim optimizers have a different behavior if the gradient is 0 or None (in one case it does the step with a gradient of 0 and in the other it skips the step altogether).

More: https://pytorch.org/docs/stable/generated/torch.optim.Optimizer.zero_grad.html

Operator Fusing

Pointwise operations

- Pointwise operations are memory-bound, for each operation PyTorch launches a separate kernel.
- Use the torch.jit to fuse pointwise operators into a single operator (kernel call).
- Fused operator launches only one kernel for multiple fused pointwise ops.

More: https://pytorch.org/docs/stable/jit.html#

Operator Fusing

Optimizers

- PyTorch has 3 major categories of oprimizers: for-loop, foreach (multi-tensor), and fused.
- Think of foreach implementations as fusing horizontally and fused implementations as fusing vertically on top of that.
- fused > foreach > for-loop.
- supported optimizers: FuseAdam, FuseLAMBD, FusedNovoGrad, FusedSGD.

More: https://pytorch.org/docs/stable/optim.html

Checkpoint intermediate buffers

Intermediate layer storing

- For the backward pass, store the inputs of a few layers and recompute others during the backward pass.
- This reduces the memory requirements, and enables increasing the batch size

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

Avoid CPU-GPU Synchronizations

Avoid operations that requires synchronization such as:

- print(cuda_tensor)
- cuda_tensor.item()
- cuda_tensor.cpu()
- python control flow which depends on the results of operations performed on cuda tensors e.g. if (cuda_tensor ≠ 0).all()

More: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html#avoid-unnecessary-cpu-gpu-synchronization

Use Mixed Precision and AMP

Mixed Precision

- Some operations use the torch.float32 data type and other operations use torch.float16.
- Some operations, such as linear layers and convolutions are much faster in float16.
- Other operations like reductions often require the dynamic range of float32.
- AMP tries to match each op to its appropriate data type.

More: https://pytorch.org/docs/stable/amp.html

bfloat16 Data Type

bfloat16

- Neural networks are more sensitive to the size of the exponent than the size of the mantissa.
- Provides identical behavior for underflows, overflows, and NaNs.
- bfloat16 is a drop-in replacement for float32 when training and running deep neural networks.

More: https://cloud.google.com/tpu/docs/bfloat16

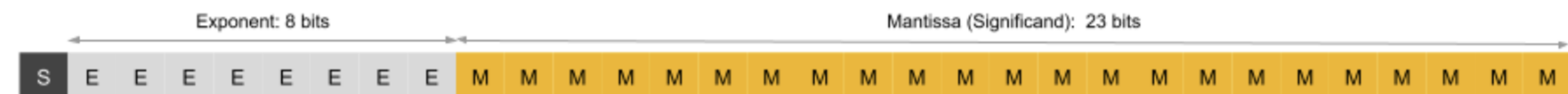
Floating Point Formats

bfloat16: Brain Floating Point Format

Range: ~1e^{-38} to ~3e^{38}

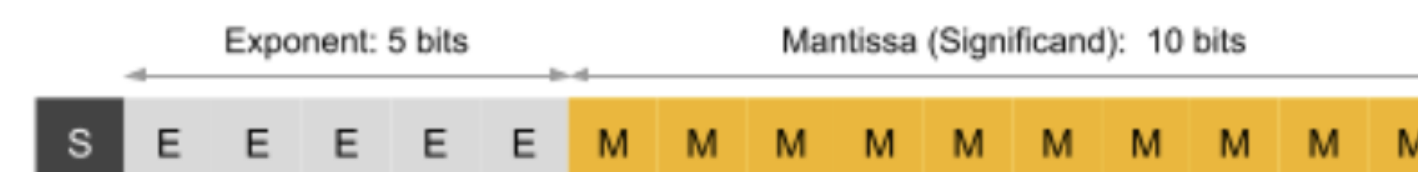
fp32: Single-precision IEEE Floating Point Format

Range: ~1e^{-38} to ~3e^{38}



fp16: Half-precision IEEE Floating Point Format

Range: ~5.96e^{-8} to 65504



Profilers

A Guide to Boosting Speed and Resource Usage

Hovhannes Tamoyan

PyTorch Profiler

torch.profiler

- PyTorch Profiler is a tool that allows the collection of performance metrics during training and inference.
- Profiler's context manager API can be used to better understand what model operators are the most expensive, examine their input shapes and stack traces, study device kernel activity and visualize the execution trace.

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

Trace Viewers

Tools

- chrome://tracing
- https://www.speedscope.app/
- https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras

Extras

A Guide to Boosting Speed and Resource Usage

Hovhannes Tamoyan

Debugging Parallel Ops

You can force synchronous computation by setting environment variable CUDA_LAUNCH_BLOCKING=1.

Annotate Tensor Shapes

```
class NN:
    embedding: "(V, E)" = torch.zeros(V, E)

assert(str(tuple(var.shape)) ==
NN.__annotations__["embedding"])
```

More: https://github.com/ofnote/tsalib

cuDNN Benchmark

A bool that, if True, causes cuDNN to enable the inbuilt auto-tuner to find the best algorithm to use for your hardware:

torch.backends.cudnn.benchmark

Thank You

Hovhannes Tamoyan