PARIO COURSE – PYTHON ML/DL (2/2)

2024-11-05 | Jan Ebert | `ja.ebert@fz-juelich.de` | Helmholtz AI Consultant | Jülich Supercomputing Centre

JÜLICH Forschungszentrum

Part I: Outline

Outline

Data loading pipelines

Checkpointing

Logging

Hands-on exercise tomorrow

Exercises

Part II: Data loading pipelines

Libraries and data loaders (refresher)

- **PyTorch**: `torch.utils.data.DataLoader`
- **TensorFlow/JAX**: `tf.data`

Worth mentioning: HuggingFace `datasets` as cross-framework abstraction.

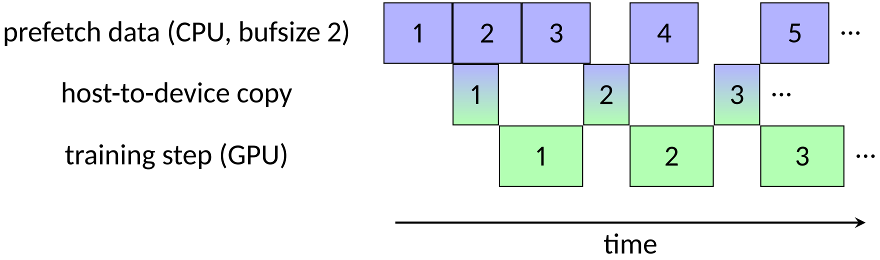
Data loader overview

	PyTorch	TensorFlow/JAX
API	`torch.utils.data.DataLoader`	`tf.data.Dataset`
multiprocessing	`num_workers=4`	`num_parallel_calls=4`
prefetching	`prefetch_factor=2`	`dataset.prefetch(buffer_size)`
caching	-	`dataset.cache()`
memory pinning	`pin_memory=True`	-
sharding	`[...].DistributedSampler(dataset)`	`dataset.shard(world_size, rank)`
batching	`batch_size=1`	`dataset.batch(batch_size)`
shuffling	`shuffle=True`	`dataset.shuffle(buffer_size)`

Some `tf.data` function arguments allow `tf.data.AUTOTUNE` for dynamic value assignment. For JAX: `jax.tree_util.tree_map(lambda t: t._numpy(), batch)` and `flax.jax_utils.prefetch_to_device` may prove useful.

Prefetching

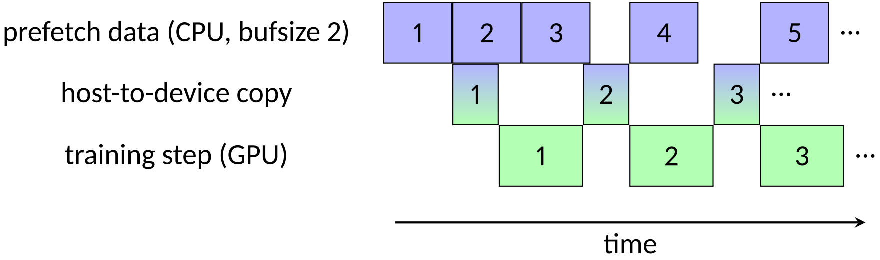
Asynchronously process future required data in a separate thread. Store it in a buffer for later.



What would be a good general choice for the buffer size to guarantee that training and data prefetching are always overlapped?

Prefetching

Asynchronously process future required data in a separate thread. Store it in a buffer for later.



What would be a good general choice for the buffer size to guarantee that training and data prefetching are always overlapped?

Answer: 2.

Shuffling

When using `torch.utils.data.DistributedSampler`, it is also used for shuffling.

Requires calling `dataloader.sampler.set_epoch(epoch)` before the data loader is iterated each epoch; otherwise same shuffling is re-used.

Shuffling iterable-style data requires a buffer to store and sample data from. Works similar to prefetching.

Interlude: Memory management

Reminder: Kernel uses virtual memory to optimize accesses to high-bandwidth physical storage.
This memory management means data is swapped in and out as required: memory is pageable.

Assign memory section as page-locked/pinned to disallow management unit from swapping it out.

Host-to-device transfer

Here: host = CPU, device = GPU.

- Device cannot access data from host's pageable memory.
  $\rightarrow$ Need to explicitly assign memory section as pinned to avoid swap-out.
- For a host-to-device transfer, this means:
  1. allocate page-locked buffer
  2. copy host's paged data to page-locked buffer
  3. transfer data from page-locked buffer to device
  4. free page-locked buffer.

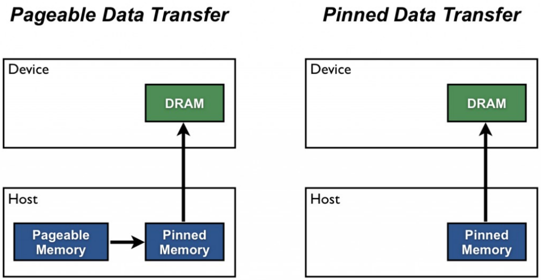


Figure: From: `https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/`

Memory pinning in practice

RAM section is pinned, i.e., configured so that it will not be swapped out.
$\longrightarrow$ Guaranteed fast access at the cost of RAM.

Can be used by PyTorch for data loading:

1. Place data on pinned CPU memory.
2. Move data to GPU immediately.

Part III: Checkpointing

PyTorch serialization

- Saving and loading across PyTorch versions not guaranteed.
- Cross-platform portability: unsure because of Pickle usage, but looked fine from implementation side.
- Tensor data is transferred to CPU for saving.

PyTorch serialization

- Saving and loading across PyTorch versions not guaranteed.
- Cross-platform portability: unsure because of Pickle usage, but looked fine from implementation side.
- Tensor data is transferred to CPU for saving.

More info in the serialization documentation:
`https://pytorch.org/docs/stable/notes/serialization.html#serialized-file-format-for-torch-save`

Naive checkpointing

Training checkpointing = storing state (model weights, optimizer state, …) persistently to allow resumption of training.

Naively:

1. (If model is sharded:) gather all weights on rank 0.
2. Save checkpoint file only on rank 0.

JÜLICH Forschungszentrum

Naive checkpointing

Training checkpointing = storing state (model weights, optimizer state, …) persistently to allow resumption of training.

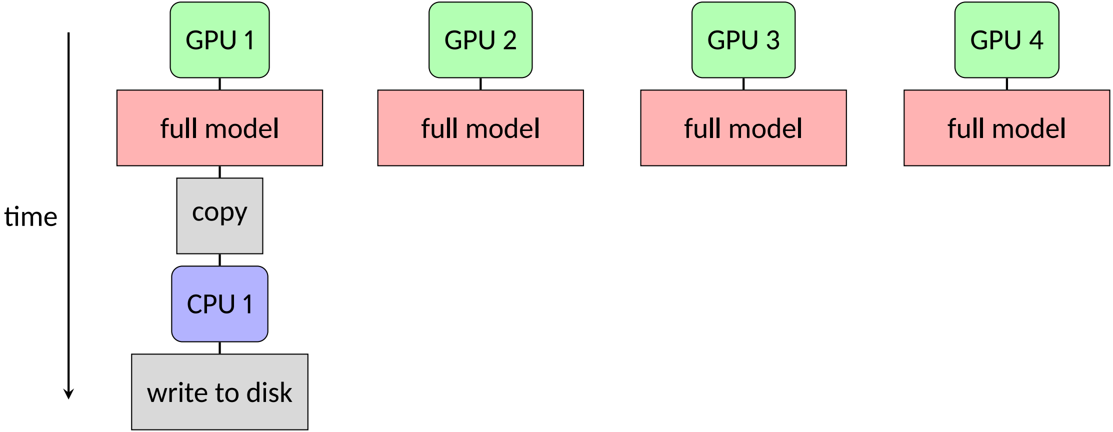
Naively:

1. (If model is sharded:) gather all weights on rank 0.
2. Save checkpoint file only on rank 0.

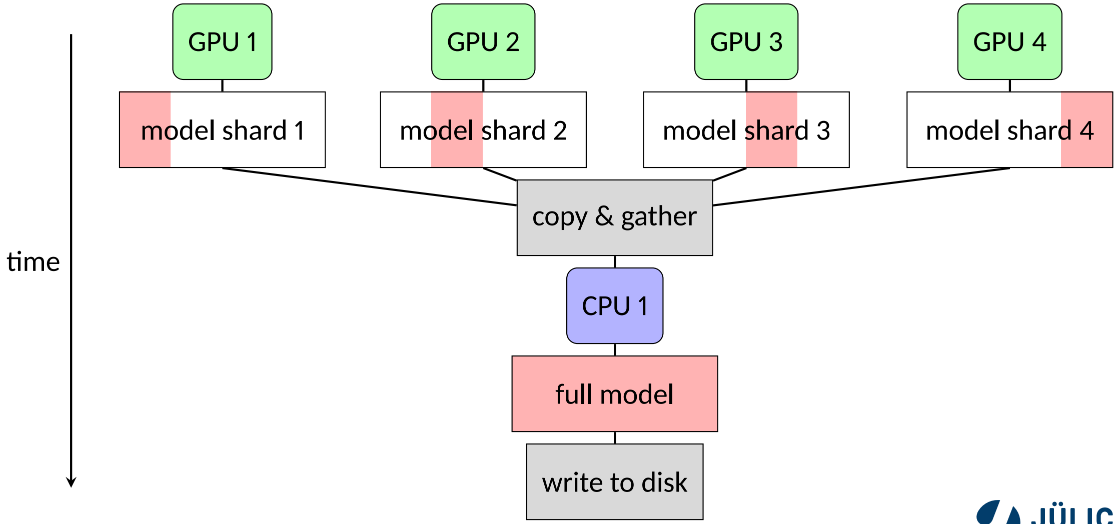
Effects:

1. Can only save models that fit in GPU/CPU memory.
2. With sharded model: Lots of communication.
3. I/O is not distributed.
4. Training is blocked during checkpointing.

Naive checkpointing (DDP)



Naive checkpointing (FSDP)



Distributed checkpointing (1/3)

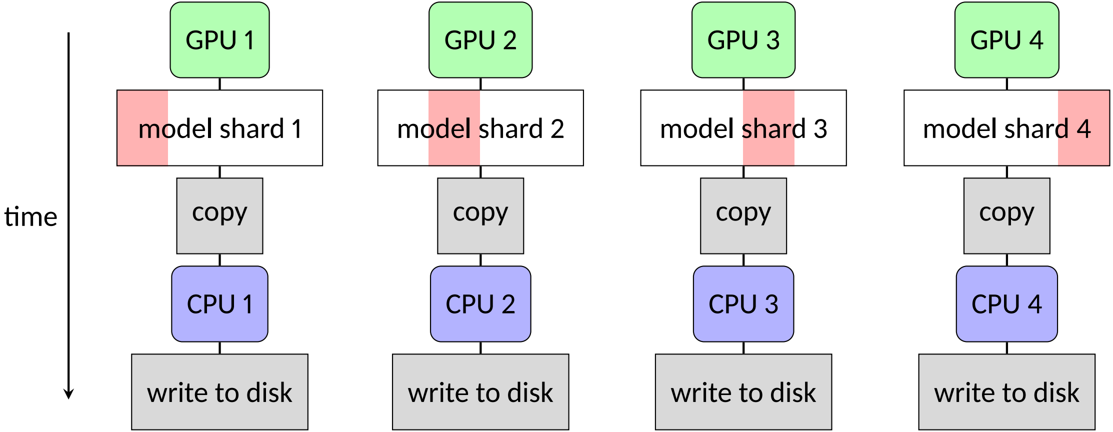
Setting: we have a sharded model, i.e., each GPU contains a part of the model.

Naively, we gather the model onto one rank and then save it there.

Instead, with distributed checkpointing, just save the sharded model on each GPU. Only applicable for bigger models that we actually shard.
$\rightarrow$ All GPU processes write in parallel, communication is avoided.

JÜLICH Forschungszentrum

Distributed checkpointing (2/3)



Distributed checkpointing (3/3)

- APIs also work transparently for non-distributed checkpointing.
- Models can be loaded with fewer or more ranks!
- No backward compatibility guarantees!

Distributed checkpointing (3/3)

- APIs also work transparently for non-distributed checkpointing.
- Models can be loaded with fewer or more ranks!
- No backward compatibility guarantees!

Effects:

1. ~~Can only save models that fit in GPU/CPU memory.~~
2. ~~With sharded model: Lots of communication.~~
3. ~~I/O is not distributed.~~
4. Training is blocked during checkpointing.

JÜLICH Forschungszentrum

Optimizing distributed checkpointing

Default implementation writes to a different file on each rank.
$\rightarrow$ Metadata accesses!

- Can implement new `StorageWriter` to, e.g., only write to a single file.
- With fixed-size state, don't need to communicate for each checkpoint.

Asynchronous checkpointing (1/2)

In PyTorch: Only works with distributed checkpointing.

After copying the checkpoint data to CPU, training resumes. Data is saved to disk in a separate thread.
$\rightarrow$ Training can continue in the meantime.

Asynchronous checkpointing (1/2)

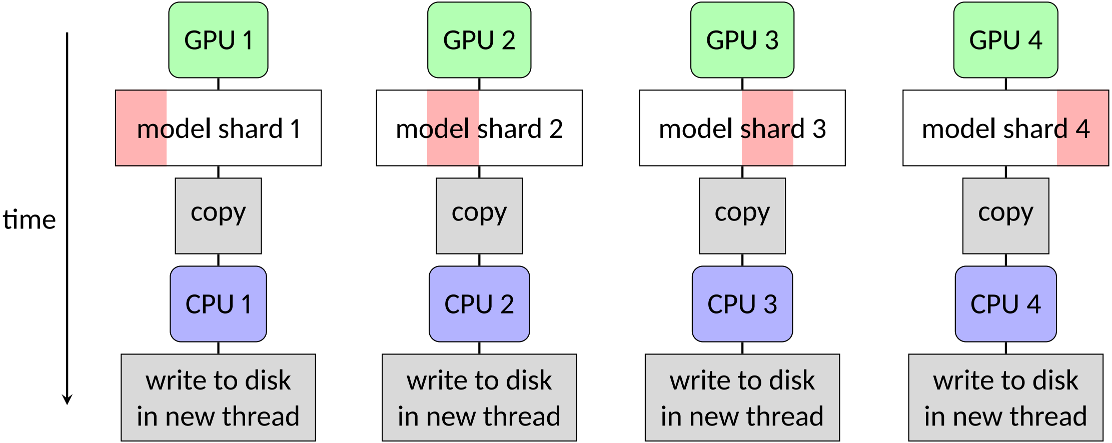
In PyTorch: Only works with distributed checkpointing.

After copying the checkpoint data to CPU, training resumes. Data is saved to disk in a separate thread.
$\rightarrow$ Training can continue in the meantime.

Effects:

1. ~~Can only save models that fit in GPU/CPU memory.~~
2. ~~With sharded model: Lots of communication.~~
3. ~~I/O is not distributed.~~
4. Training is blocked during ~~checkpointing~~ device-to-host copy of shard.

Asynchronous checkpointing (2/2)



Part IV: Logging

Logging

Logging various metrics is key to debugging and improving training.

PyTorch, TensorFlow, and JAX “native” way: TensorBoard
PyTorch and JAX require explicit installation of the `tensorboard` package.

PyTorch API (requires `tensorboard` package):
`torch.utils.tensorboard.SummaryWriter`
TensorFlow/JAX API: `tf.summary.create_file_writer`

TensorBoard in PyTorch

	PyTorch
API	`torch.utils.tensorboard.SummaryWriter(log_dir)`
enabling	-
logging	`summary_writer.add_scalar(tag, scalar_value, global_step)`
flushing	`summary_writer.flush()`
closing	`summary_writer.close()`

Requires explicit installation of the `tensorboard` package.

For faster loading, use `summary_writer.add_scalar([...], new_style=True)`.

TensorBoard in TensorFlow/JAX

	TensorFlow/JAX
API	`tf.summary.create_file_writer(logdir)`
enabling	`with summary_writer.as_default():`
logging	`    tf.summary.scalar(tag, value, step)`
flushing	`summary_writer.flush()`
closing	`summary_writer.close()`

JAX requires explicit installation of the `tensorboard` package.

Logging I/O improvements

Logging frameworks usually keep a buffer that they write to before committing the logged values to a file.

- PyTorch:
  `SummaryWriter([...], max_queue=10, flush_secs=120)`
- TensorFlow/JAX:
  `create_file_writer([...], max_queue=10, flush_millis=120_000)`

Part V: Hands-on exercise tomorrow

Hands-on exercise tomorrow

You’re given a naive, distributed Vision Transformer training loop on fake image data; your goal is to use your newfound knowledge to make it as fast as possible!

JÜLICH Forschungszentrum

Part VI: Exercises

Setting up

1. Log into the supercomputer JUSUF (`ssh <user>@jusuf.fz-juelich.de`).
2. `cd /p/project1/training2403/ParIO_course_material/exercises/Python_ML_DL`
3. `nice bash set_up.sh` and wait until done. In the meantime, feel free to look into exercise 1.1 in the same directory!
4. Every time you (re-)connect to the machine and want to do the Python ML/DL exercises, execute the following to activate the software environment:
   `cd /p/project1/training2403/ParIO_course_material/exercises/Python_ML_DL`
   `source activate.sh`
5. Exercises should be executed like `sbatch <file>.sbatch [args...]`.

JÜLICH Forschungszentrum

Exercise 2.1

We took a look at how data loaders can be optimized. Let's combine this with our findings from exercise 1.1 to put data loading optimizations into practice! The code actually loads the data into GPUs in a distributed fashion and simulates a model training (using `time.sleep`), so make sure that GPU transfer is optimized.

There are a few new arguments compared to exercise 1.1, mostly concerning the data loader. Please use `python 2.1_data_loading.py --help` to get a list of all arguments. See also the data loader overview table.

You can also let the code run on more than the default 2 nodes.

Exercise 2.2

The script for this exercise saves checkpoints. We already implemented naive, distributed, and asynchronous checkpointing. However, the asynchronous implementation suffers from race conditions. Important arguments: `--save-root`, `--dist-cp`, `--dist-single`, `--async-cp`.

1. Compare the script's runtime when using the various arguments.
2. Fix the asynchronous checkpointing race conditions and compare its runtime.

If you don't encounter race conditions with asynchronous checkpointing before fixing them, or if you don't have enough memory available, increase or decrease `hidden_dim` in the `build_model` function by a factor of a power of two.

You can also let the code run on more than the default 2 nodes.

JÜLICH Forschungszentrum