Leibniz Supercomputing Centre

of the Bavarian Academy of Sciences and Humanities

Leibniz Supercomputing Centre

Working with Data on GPUs - I/O, Data Transfer & Generation | 09.10.25 | GCS@NHR

Overview

1. Identifying I/O and data related bottlenecks
2. Data related compute
3. Pure I/O bottlenecks
4. Data generation on GPU

Disclaimer

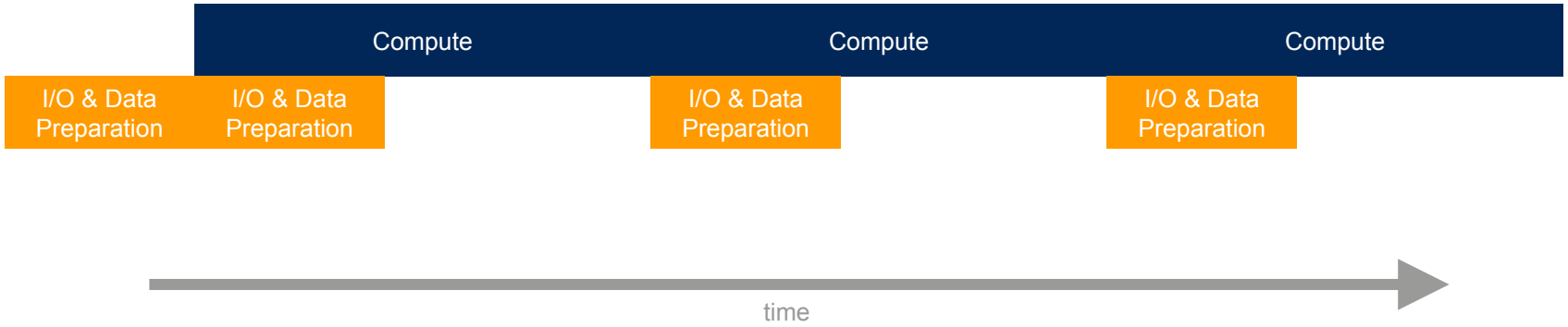
- Talk mostly interesting for beginners
- Goal:
  - Provide pointers to reduce data related bottlenecks in your ML workflows
  - Make you aware of existing solutions
  - Provide practical tips that are simple to implement in your workflows
- General considerations
  - references to PyTorch and JAX
- Consider looking at talk by Robert Schade: "My Jobs are Slow? What's wrong?"

Typical Data-Driven Machine Learning Task

Timeline:

- load data
- prepare data
- transfer to GPU
- computation on GPU
- output, e.g. values, gradient, etc. (not considered in this talk)

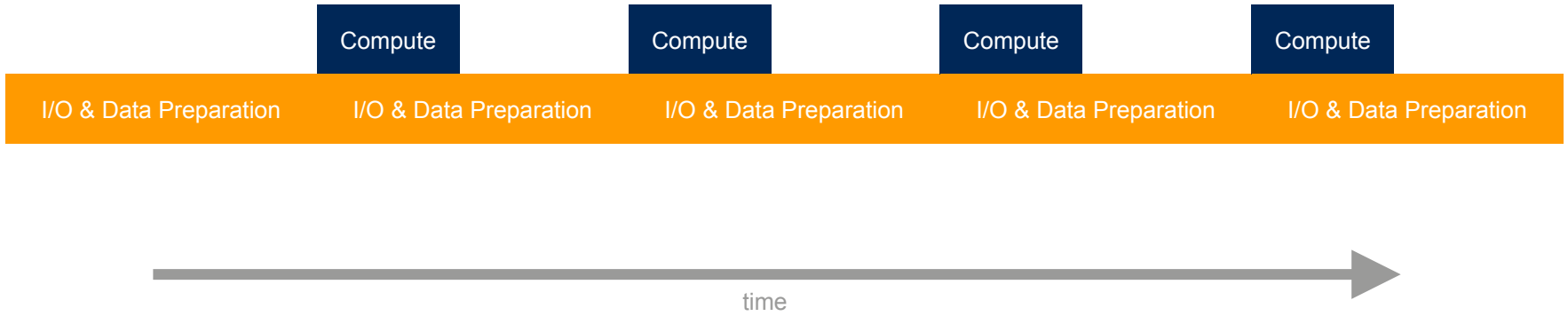
Typical Data-Driven Machine Learning Task



More realistic timeline:

- async. data handling and compute
- usually frameworks are designed to do this by default
- goal: keep GPU as busy as possible

I/O bound training



Faster compute might reveal I/O bottleneck:

- Laptop to HPC machines: powerful GPUs with high bandwidth memory
- Improved algorithms

Shared filesystem:

- High load on shared filesystem

I/O bound training

Which workflows produce such patterns?

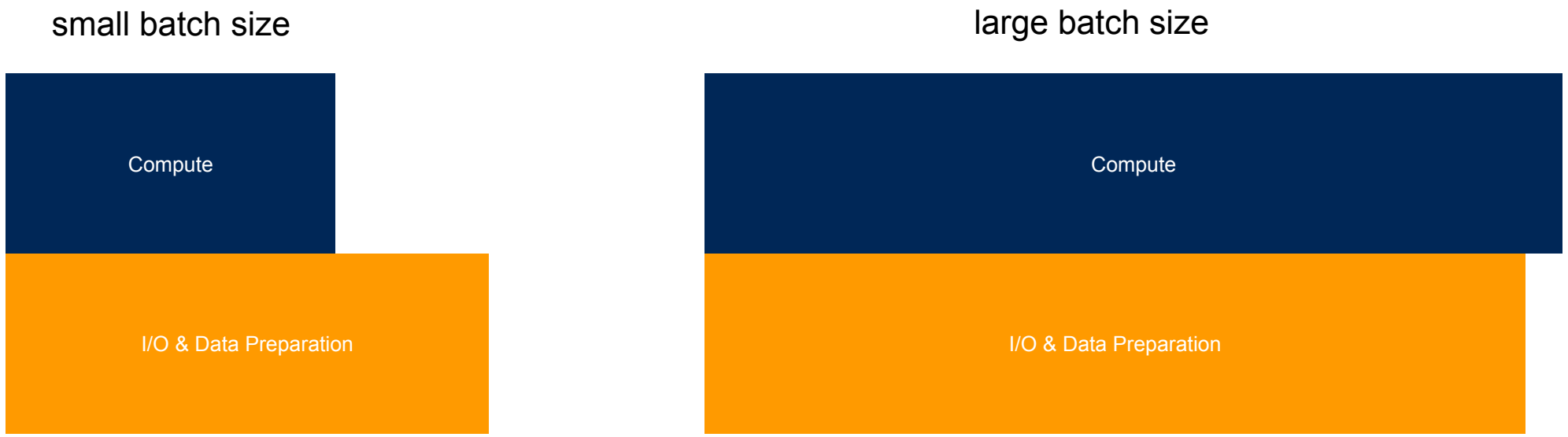
- “simple” network but “complicated” problem
- need for lots of data
- Examples:
  - high-precision training (e.g. in physics) NN output quality must be as good as possible
  - batch inference (e.g. classify a lot of pictures/video frames, run over large amounts of texts)
  - architecture search (repeated training runs)

I/O bound training

How to identify?

- Easiest way: replace data with dummy data on GPU
  - use random data to avoid caching!
  - find maximum possible improvement speedup
- Profiler (not always available on HPC systems)

I/O bound training



First thing to try: Increase batch size! → Often data handling time increases slower than compute time

Larger batch sizes can often keep GPUs busy for long enough

Careful: too large batch sizes bad for stochastic learning

How to address IO and data related bottlenecks?

This talk: How can we easily speed up data handling to be fast enough for the GPUs

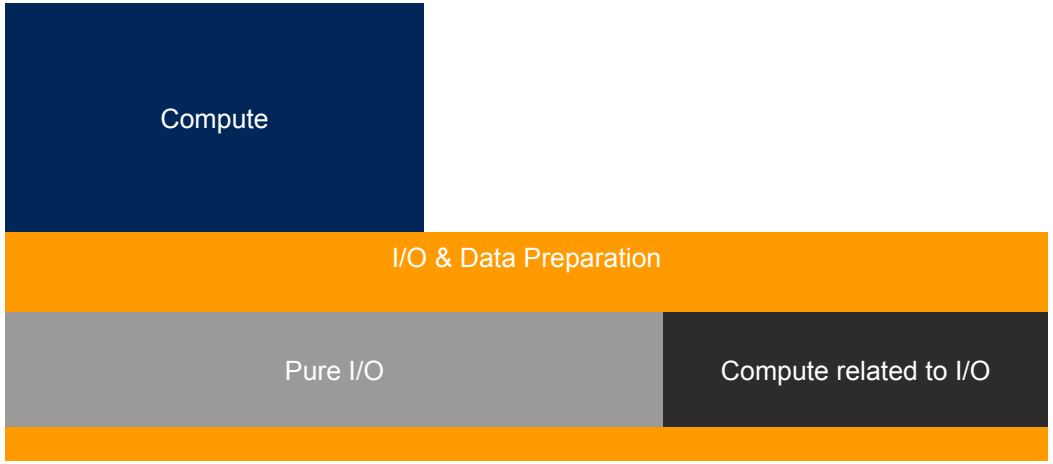
For large neural networks:

- communication of weight updates and checkpoints relevant
- typically not I/O bound
- not discussed in this talk

A detailed review of all relevant topics can be found in this paper:

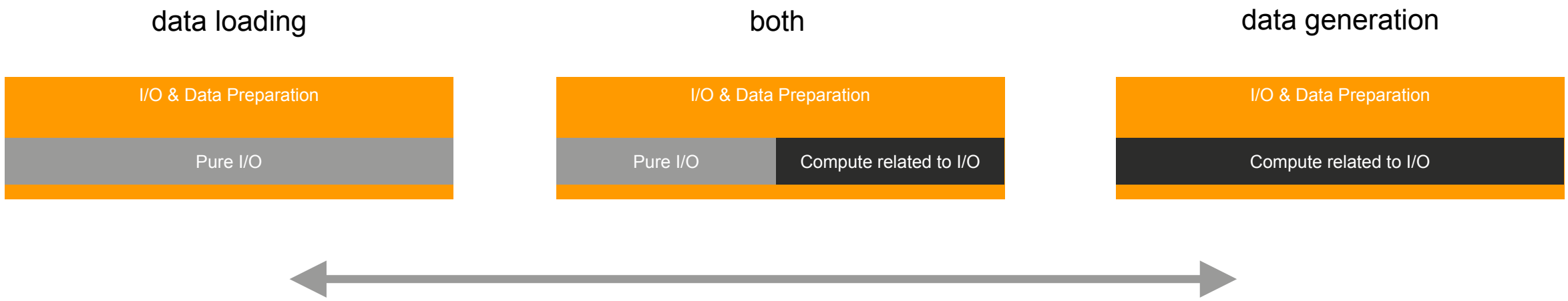
“I/O in Machine Learning Applications on HPC Systems: A 360-degree Survey”
by Noah Lewis, Jean Luca Bez, Suren Byna in ACM Computing Surveys, Volume 57, Issue 10
https://doi.org/10.1145/3722215

Spectrum of I/O and Data Preparation



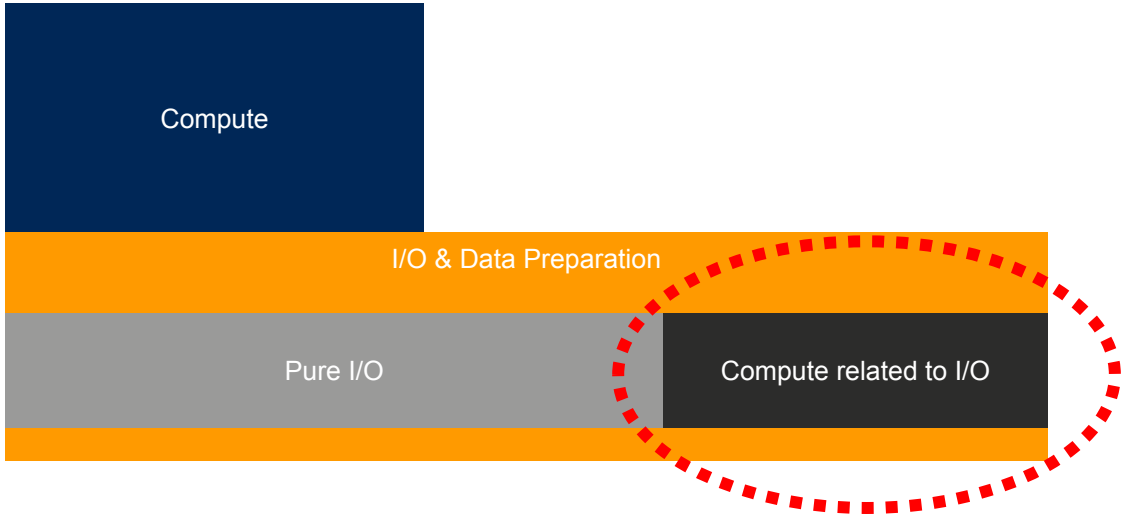
Need to speed up I/O and/or compute related to it

Spectrum of I/O and Data Preparation



Reduce Time spent on Data-Related Compute

Compute related to I/O



Compute related to data:

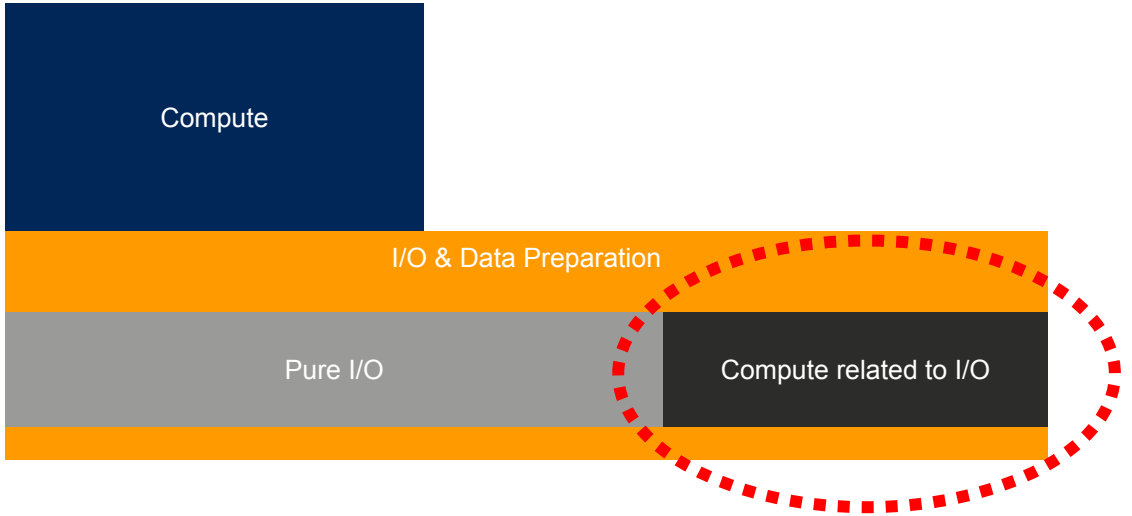
- decoding (e.g. jpeg to tensor, etc.)
- decompression (compressed text, physics data, etc.)
- augmentations
- generation

→ sometimes overlooked

Reduce data-related compute time

- use more CPU cores
  - pytroch workers: uses multiprocessing
  - MPI
  - other dataloaders
- better decoding
  - oftentimes fast libraries as drop-in replacements
- better file format
  - balance compression
- faster augmentations
  - also oftentimes drop-in (e.g. https://albumentations.ai/)

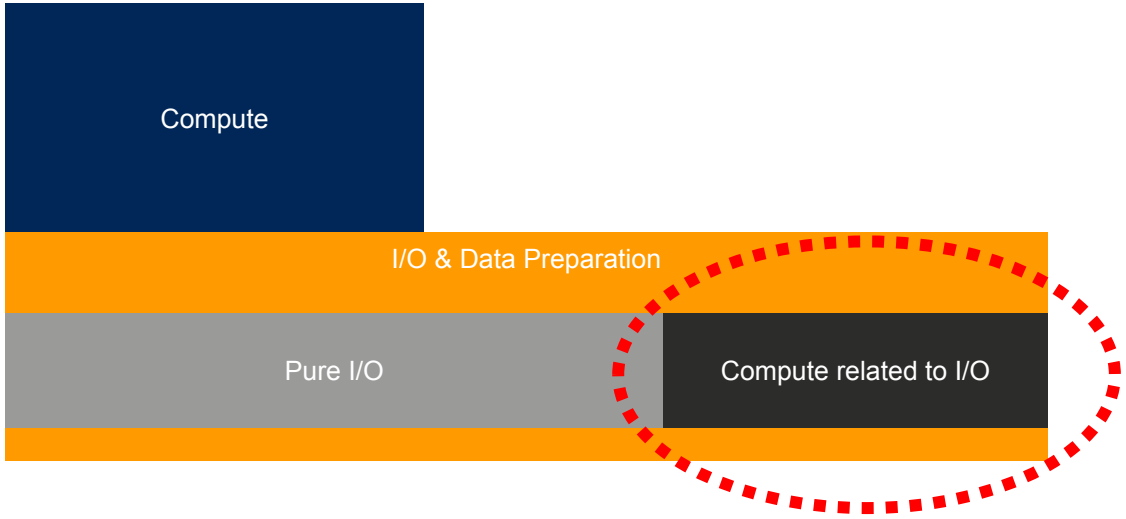
- Example for images: “Image loading benchmark : Fastest way to load large images from disk into GPU.”



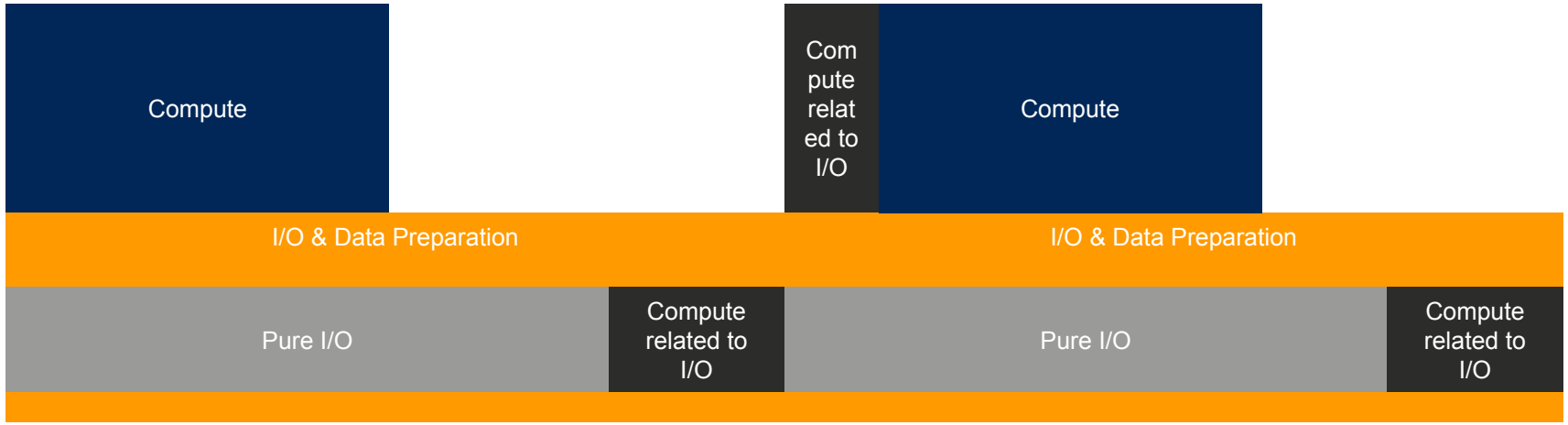
Reduce data-related compute time

Examples:

- High-Energy Physics:
  - CERN's ROOT
  - Compression settings can introduce significant CPU load
  - uproot can improve data loading
- Jpeg decoding:
  - different libraries
  - direct GPU decoding (example: nvJPEG for Nvidia GPUs)
  - used by decode_jpeg in torch
- Offload decompression to GPU (example: nvCOMP)



Make use of GPU



Make use of GPU resources:

- augmentations on GPU
- decoding (especially images, video)

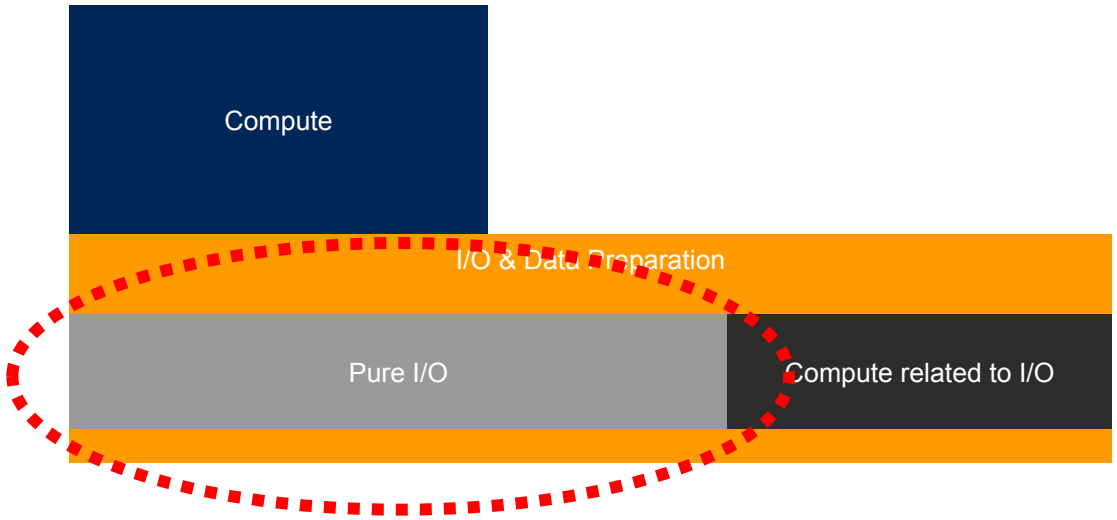
Data Augmentation on GPU

Examples:

- torchvision: apply augmentations on GPU tensors
- using JAX:
  - PIX library from images runs on GPU
  - Custom numpy-style augmentations (beyond images)
  - Just-in-time compiled
  - interoperability with torch via: dlpack (zero-copy)

Speed Up Data Loading

Data Transfer



Transfer data from filesystem to GPU

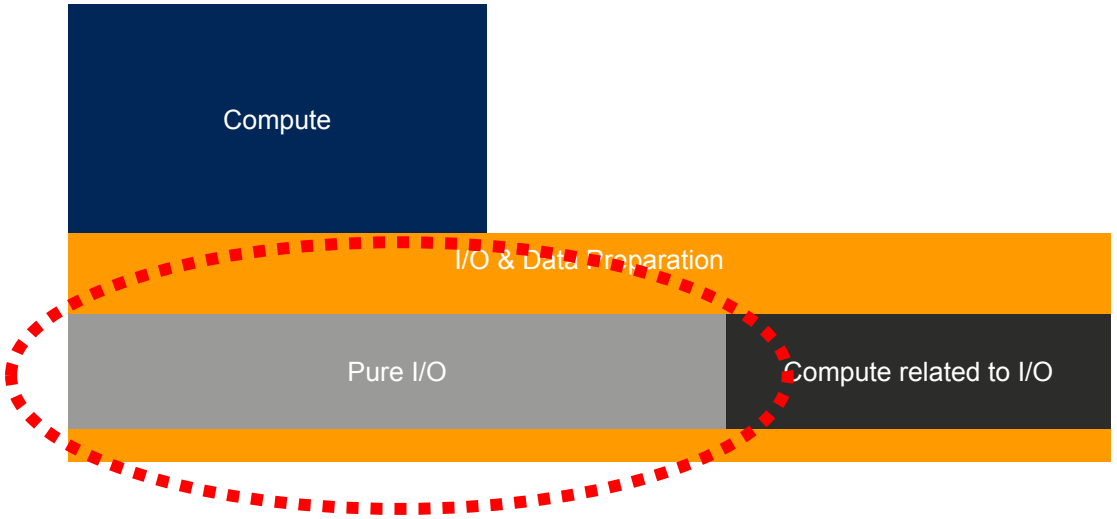
Typical Data-Driven Machine Learning Task

Machine Learning data:

- diverse (text, photos, videos, etc.)
- oftentimes many small files
- random access

HPC filesystems:

- different filesystems available
  - local (on node)
  - global
- different technology
- typically shared → fluctuations in latency and bandwidth possible



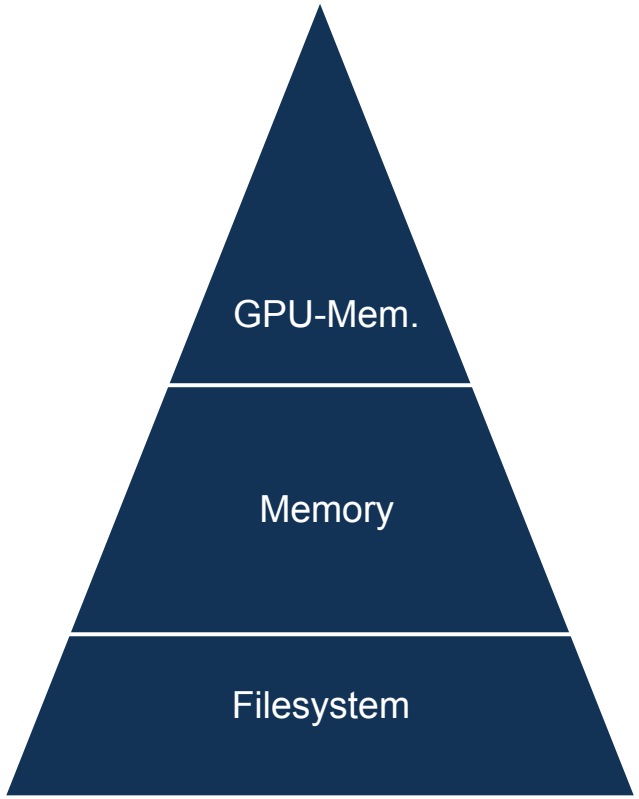
Chose Filesystem

First find best filesystem on HPC system:

- Often home directory not suitable! (typically too small anyway)
- Different paths may use different storage technology (influences bandwidth and latency)
  - Caching behavior
  - Fully SSD-based backend
- Example for SuperMUC-NG at LRZ:
  - General filesystem information: https://doku.lrz.de/file-systems-of-supermuc-ng-10746566.html
  - Parallel filesystem: https://doku.lrz.de/files/10746566/10746567/11/1755197969103/Best-Practice-Guide-Parallel-IO.pdf
  - Make use of parallel I/O with libraries such as hdf5
- Node local filesystems to avoid fluctuations

Caching:

Size:

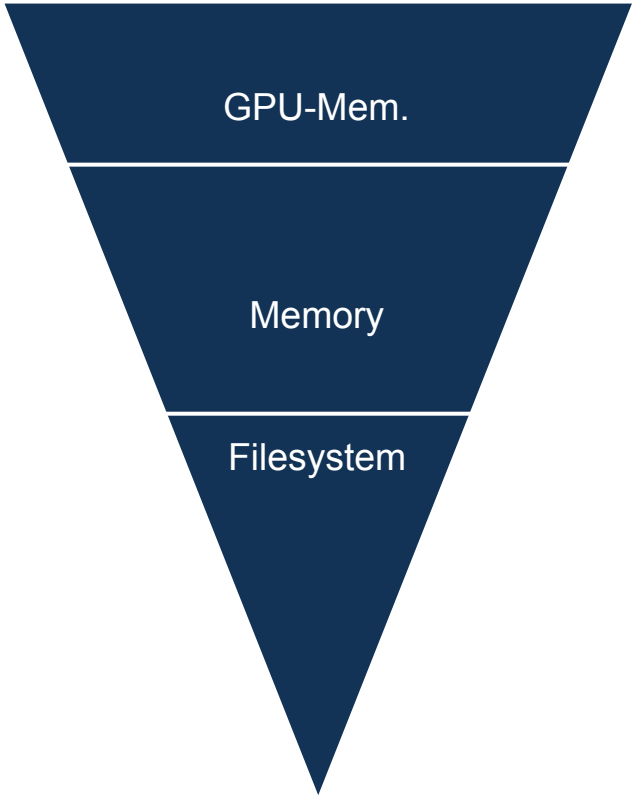


Move data to faster memory

↔

Size limitation

Speed:



Cache Size

How to deal with “small” available cache?

- Libraries often offer simple caching (e.g. torch dataloader)
- Possible pitfalls:
  - caches decoded/decompressed data → huge increase e.g. for jpeg
  - multiworker dataloaders often replicate data

Solutions?

- Cache compressed data
- Use shared memory

Simple way to do both: /dev/shm or sometimes /tmp as tmpfs

- check if allowed to use and available!
- clean up at the end of the job!
- no code changes needed!

Cache Size

/dev/shm or /tmp as tmpfs not available/allowed on every system:

- Manually cache files in memory
- Copy raw bytes into memory and open file from memory
- Can be used in combination with shared memory:
  - multiprocessing for torch (example by ptrblck here)
  - MPI windows

Cache Size

Example filling cache:

```
77          file_sizes = []
78          for path in tqdm(self.paths, desc="load image info from filesystem", unit_scale=1, unit="images",):
79              with open(path, 'rb') as imgfile:
80                  size = len(imgfile.read())
81              file_sizes.append(size)
82          file_sizes = np.array(file_sizes)
83
84          self.offsets = np.cumsum(file_sizes)
85
86          self.cache = np.empty(shape=(self.offsets[-1],), dtype=np.uint8)
87
88          for idx, path in enumerate(tqdm(self.paths, desc="load images from filesystem", unit_scale=1, unit="images",)):
89              with open(path, 'rb') as imgfile:
90                  data = np.frombuffer(imgfile.read(), dtype=np.uint8)
91              offset_upper = self.offsets[idx]
92              offset_lower = 0 if idx == 0 else self.offsets[idx-1]
93              np.copyto(self.cache[offset_lower:offset_upper], data)
94
```

Cache Size

Example decoding cache:

```
117         offset_upper = self.offsets[idx]
118         offset_lower = 0 if idx == 0 else self.offsets[idx-1]
119
120         data =  torch.from_numpy(self.cache[offset_lower:offset_upper])
121
122         img = decode_jpeg(data, device=self.device)
123         if self.transform:
124             img = self.transform(img)
125         return img
```

Cache Size

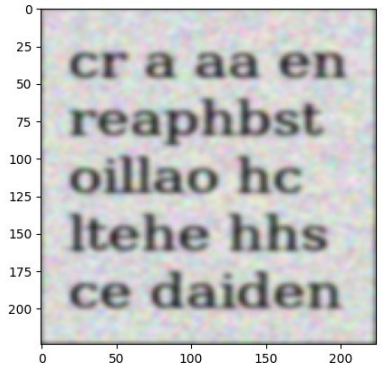
Further considerations (more involved):

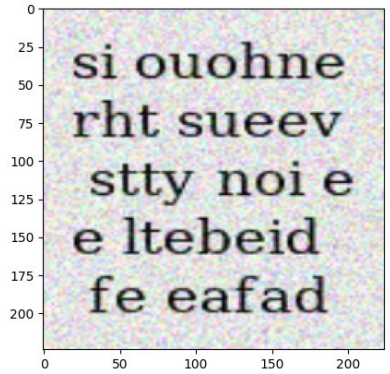
- Cache only partially (example ‘stochastic caching’: https://github.com/Charl-AI/stochastic-caching)
- Distribute across nodes: shuffle locally first and globally only sometimes

Example:

Generated 64k jpegs 224x224x3

→ ca. 1 GB of files → decompressed as tensors ca. 10 GB

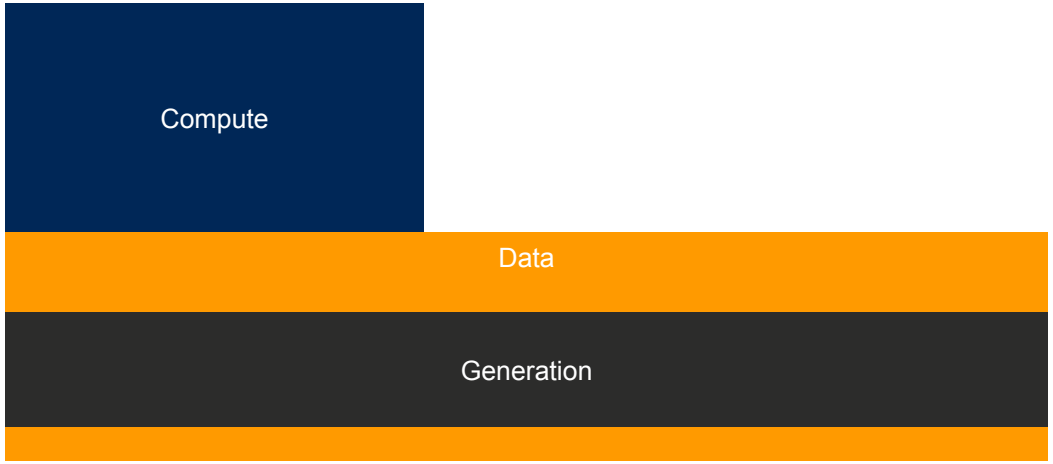




Test	Decoding	Fill Cache [images/s]	Load to GPU [images/s]
images from filesystem	CPU	-	308
images from cache	CPU	~ 12k	388
images from filesystem	GPU	-	1.45k
images from cache	GPU	~12k	2.10k
image files raw shm	-	~40k	-
images from shm	GPU	-	1.92k
images from shm	CPU		370

Data Generation on GPU

Data Generation



Sometimes training data is artificial data:

For example from simulations, from LLMs, etc.

Artificial Data

Consider generating the data directly on the GPU → removes data transfer completely

Not always possible:

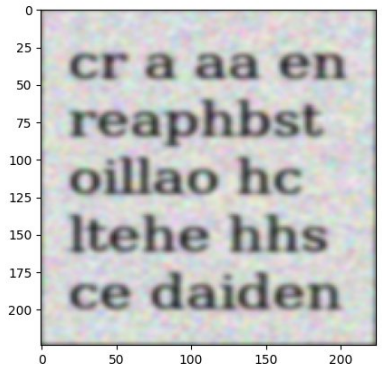
- existing simulation code might be complicated and old
  - HEP is a good example for this
- generate simplified data and apply transfer learning
- can also help to use sparse data more efficiently
  - first generated ‘good enough’ artificial data
  - finish training with dataset

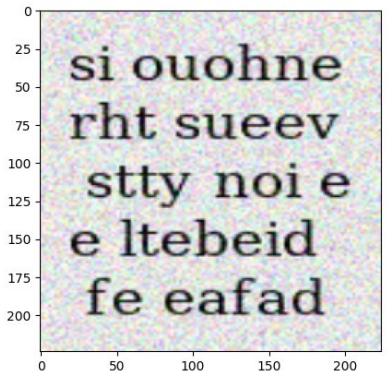
JAX makes it easy to write simulations on GPU!

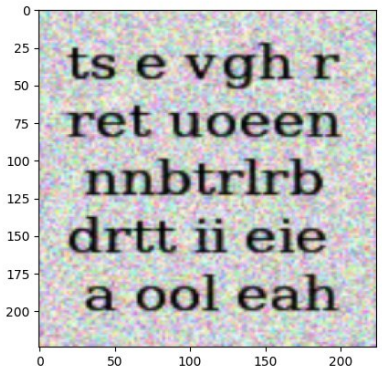
Example (continued)

Images from before: generated on GPU!

- generation approx. 40k images/s!
- when using 100% JAX/Flax code: compile generation and training loop
- training 100% on GPU
- training with approx. 8k images/s → faster than loading with decoding







Example (continued)

```python
@jax.jit  # regular JAX
def jax_train_step(graphdef, state, key):

    # define loss function
    def loss_fn(model, x, found):
        predictions = model(x)[:,0]
        return jnp.mean(optax.losses.sigmoid_binary_cross_entropy(predictions, found)), predictions

    # value and gradient of loss
    vgr = nnx.value_and_grad(loss_fn, has_aux=True)

    # microstep
    def body_fn(i, state_and_key):

        state, key = state_and_key

        model, optimizer, metrics = nnx.merge(graphdef, state)

        # get random key and generate a batch
        key, key_batch = random.split(key)
        x, idcs = gen_text_img_batch(key_batch, n_batch=n_batch)

        # check if string in generated image
        idcs_reshaped = idcs.reshape((idcs.shape[0], -1))
        found = letter_indices_in_line_batch(test_idcs, idcs_reshaped).astype(jnp.int32)

        # calc. grads; update state and metrics
        (loss, logits), grads = vgr(model, x, found)
        optimizer.update(model, grads)
        metrics.update(loss=loss, logits=logits, labels=found)

        state = nnx.state((model, optimizer, metrics))

        return state, key

    # run several microsteps
    state, key = jax.lax.fori_loop(0, n_microsteps, body_fn, (state, key), unroll=5)

    return state
```

Here: use Flax NNX and JAX to write fully compile training loop!

See: https://flax.readthedocs.io/en/v0.6.11/getting_started.html

Summary

Summary

- Oftentimes simple handles to improve data-related bottlenecks
- Familiarize with system
  - use optimal filesystem
  - use caching
  - consider input format
- Look for drop-in replacements of augmentation libraries
- Combine techniques
- Consider data generation on GPU