Tuning Deep Learning Training & Evaluation Performance on the OU Supercomputer

ai2es.org

Mel Wilson Reyes, Jay Rothenberger, Andrew H. Fagg*

NSF AI Institute for Research on Trustworthy AI in Weather, Climate, and Coastal Oceanography (AI2ES)

School of Computer Science

Data Institute for Societal Challenges*

Special thanks to: Dr. Randy Chase

The UNIVERSITY of OKLAHOMA

CPU vs. GPU Processing

There are two types of computational devices we have access to:

CPU

- General purpose
- few (1-64) cores / parallel operations
- Must handle I/O tasks for data in RAM and
- Python code you write runs here

GPU

- Specialized
- Many (1000+) cores / parallel operations
- Can only operate on data in VRAM (GPU memory)
- TensorFlow code can run here

Using Python/DL with GPUs

- Tensorflow/PyTorch packages in Python provide an API for interfacing with GPUs
- By default, tensorflow-gpu will use all available memory on all GPU devices
- When multiple programs attempt to use the same GPU, they can interfere destructively with one-another
- Approach: in addition to reserving a node, also reserve one or more GPUs on this node

Reserving GPUs

Add to your batch file:

- Single GPU reservation:

  ```
  #SBATCH --gres=gpu:1
  ```

- Two GPUs:

  ```
  #SBATCH --gres=gpu:2
  ```

- During execution, your batch file environment variable $CUDA_VISIBLE_DEVICES will be set to a comma-separated string containing the integers of the physical GPUS that have been allocated
  - This is more useful for debugging that for automated program configuration

Using Multiple GPUs with Tensorflow

There are multiple options - the simple one is the Mirrored Strategy:

- Place a copy of the model onto each GPU
- Split the batch into N pieces, sending one piece to each GPU
- Each GPU performs a forward/backward pass with its batch
- The weight updates are summed & then shared back to each GPU
- Repeat

Using Multiple GPUs with Tensorflow

Using the Mirrored Strategy is Relatively Straight Forward:

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    # build the model (in the scope)
    model = network_fn(**network_args)
    # Must instantiate the loss/metrics here
    model.compile(...)
   :
   :
history = model.fit(...)
```

Note: batch size should generally be scaled by number of GPUs

Monitoring CPU, Memory, GPU, and I/O Loads

- Open a bash shell on the node your job is running on:

```
srun --jobid=JOBID --pty bash
```

In that shell:

- Monitor CPU, memory and I/O:

```
top
```

- Monitor GPU load and memory use:

```
nvidia-smi
```

top

```
top - 14:34:13 up 35 days, 18:44,  1 user,  load average: 0.94, 0.78, 0.75
Tasks: 400 total,   1 running, 399 sleeping,   0 stopped,   0 zombie
%Cpu(s):  6.3 us,  1.4 sy,  0.0 ni, 92.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 65710052 total, 52559580 free,  8746172 used,  4404300 buff/cache
KiB Swap:  8388604 total,  7595868 free,   792736 used. 53733932 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
29085 jroth     20   0   58.7g   6.6g 514280 S 187.0 10.5  68:19.11 python
 1884 telegraf  20   0 2263992  10636   3744 S   2.3  0.0 279:34.50 telegraf
29592 jroth     20   0  579224 106480  10308 S   1.3  0.2   1:58.88 jupyter-lab
30523 jroth     20   0 5235448 126764  14524 S   1.3  0.2   0:28.89 wandb-serv+
 1074 root      20   0       0      0      0 S   0.3  0.0   3:21.12 xfsaild/dm+
 1717 root      20   0       0      0      0 S   0.3  0.0   0:10.09 nv_queue
 1981 root      20   0  773304  16548   2412 S   0.3  0.0  25:05.59 salt-minion
 9951 fagg      20   0  172692   2588   1616 R   0.3  0.0   0:00.02 top
    1 root      20   0  194660   4716   2520 S   0.0  0.0   9:51.90 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:04.33 kthreadd
    4 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:+
    6 root      20   0       0      0      0 S   0.0  0.0   0:01.39 ksoftirqd/0
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.44 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0  17:54.38 rcu_sched
   10 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 lru-add-dr+
   11 root      rt   0       0      0      0 S   0.0  0.0   0:07.97 watchdog/0
```

Monitoring CPU, Memory and I/O Loads (with top)

- CPU: max use should stay within your reservation (--cpus_per_task)
  - For your process: ceiling(%CPU / 100) <= cpus_per_task
  - If load average > total number of threads available on the node, then someone is not behaving
- Memory: max use should stay within your reservation (--mem)
  - For your process: RES is the amount of RAM that your process is using
  - If free memory is low compared to total RAM, then someone is not behaving

nvidia-smi

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.57       Driver Version: 515.57       CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-PCI...  On   | 00000000:3B:00.0 Off |                    Off |
| N/A   49C    P0    74W / 250W |  39751MiB / 40960MiB |     63%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA A100-PCI...  On   | 00000000:5E:00.0 Off |                    Off |
| N/A   62C    P0   212W / 250W |  39751MiB / 40960MiB |     93%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A    236087      C   python                          39749MiB |
|    1   N/A  N/A    167230      C   python                          39749MiB |
+-----------------------------------------------------------------------------+
```

Monitoring GPU Utilization

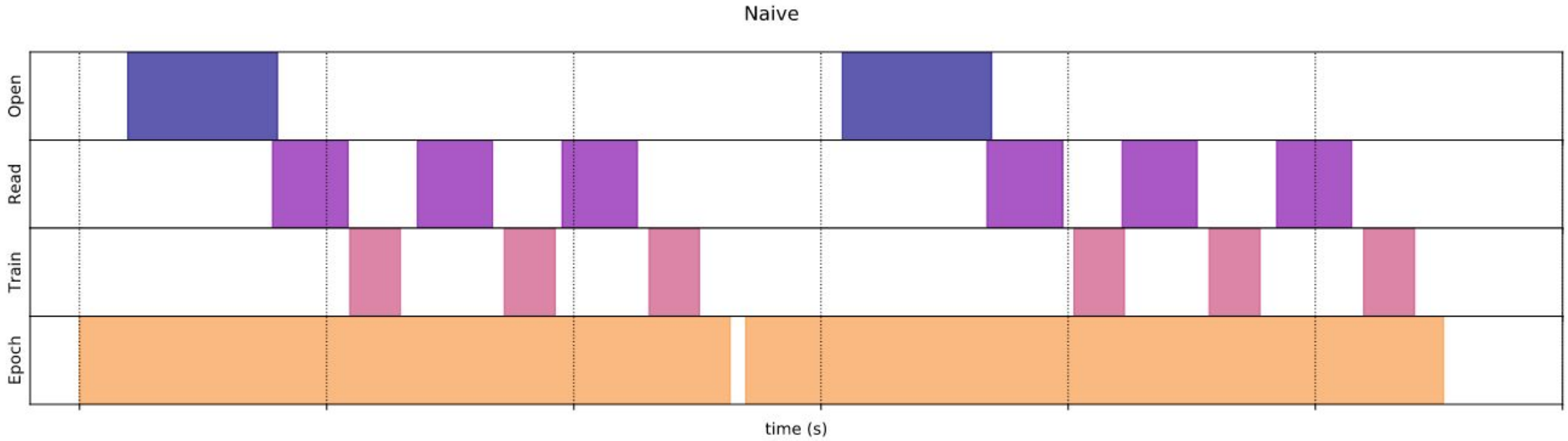
- GPU-Util: want this to be as close to 100% as possible. If it is not, you have various things you can try:
  - Increase batch size
  - Increase the number of threads available for your TF Datasets (more on this coming)
  - Cache your dataset closer to the GPU (more on this coming)

- Memory Usage:
  - Keep batch size small enough so that you are not maxing out available VRAM (get close, but don't exceed)
  - Exceeding -> memory allocation error, Out of Memory (OOM)

Large Data Sets

- Our data sets are often small enough to fit into RAM/GPU RAM
- For interesting data sets (e.g., where we have a large number of images), these data don't fit! Only a subset of data will fit in RAM at once
  - For Mel Wilson Reyes' Visibility data set, we have ~1.8M images
- We will swap parts of data set into RAM as they are needed … we call these batches
- Pipeline the process of loading, preparing, and computing gradients for different batches simultaneously
  - Perform I/O and Training at the same time to avoid bottleneck

Large Data Sets: Naive Approach

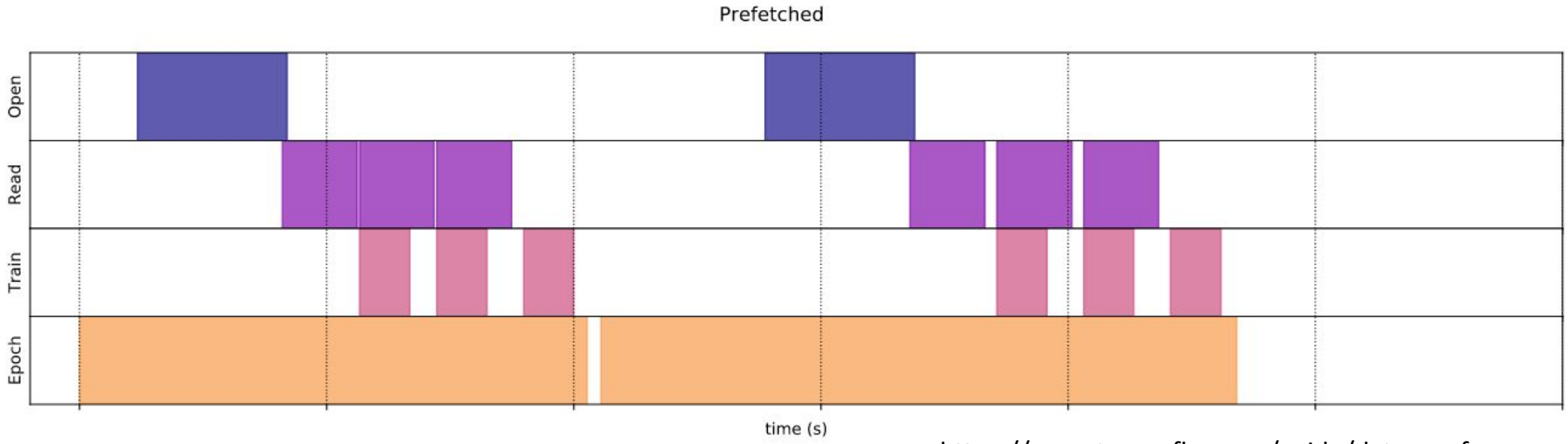


- Blue: Initializing fetch of data from spinning disk
- Purple: Loading/preparing data
- Pink: Training with the GPU

https://www.tensorflow.org/guide/data_performance

Large Data Sets: Prefetching + Parallel Execution

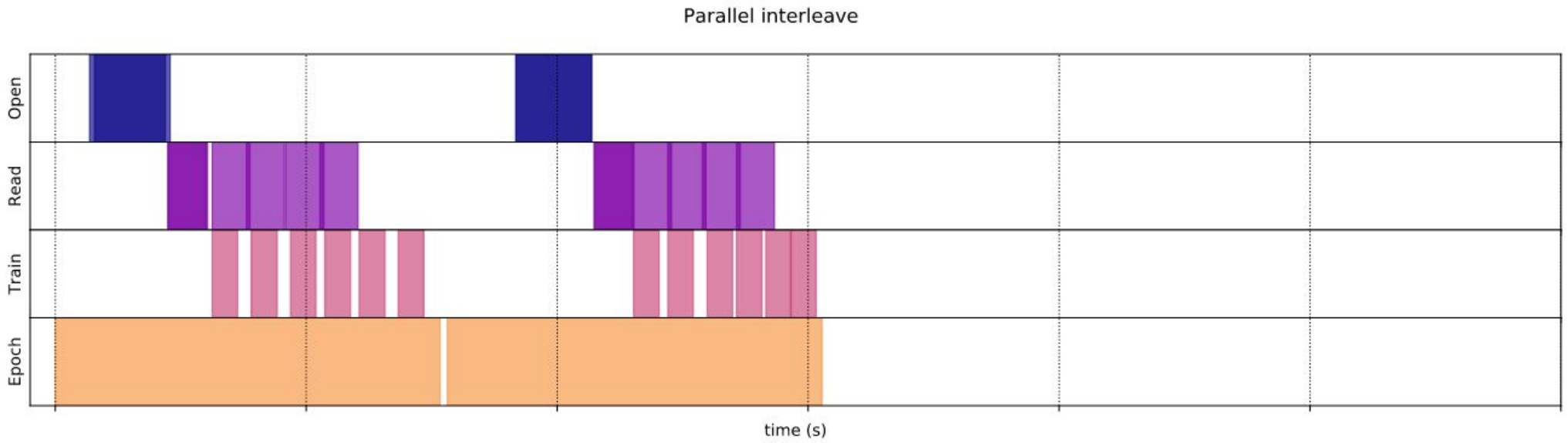
- Fetching new batch before training with the current batch completes
- Better utilization of the GPU



https://www.tensorflow.org/guide/data_performance

Large Data Sets: Prefetching Multiple Batches at Once

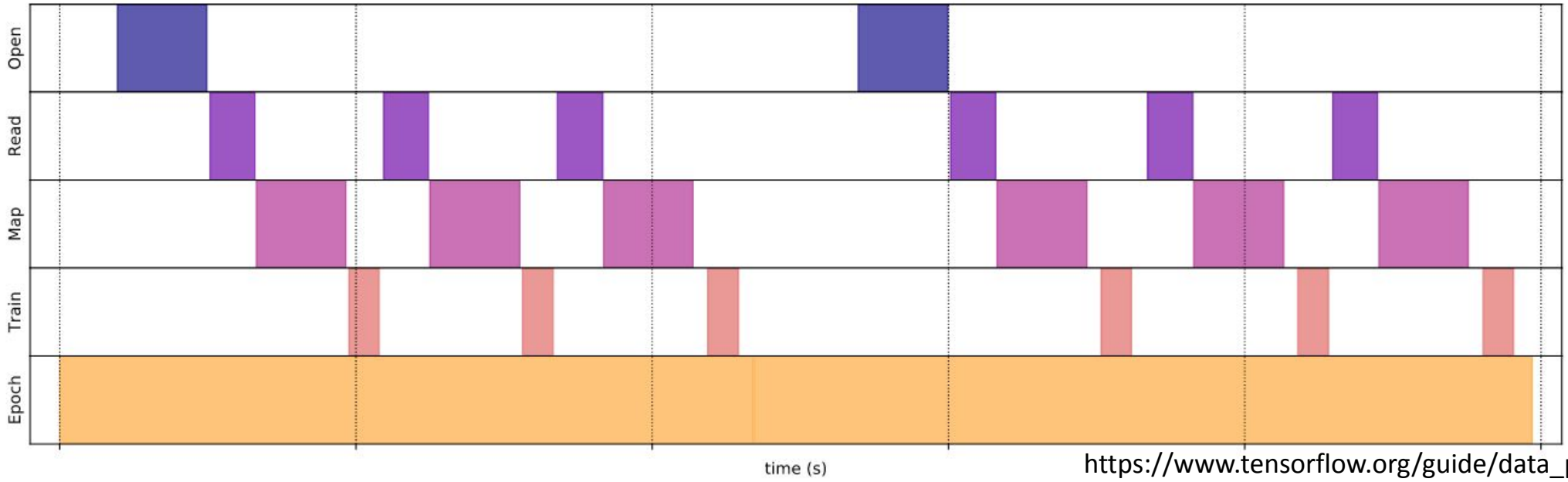
- Each batch is fetched using one or more threads



https://www.tensorflow.org/guide/data_performance

Large Data Sets: Data Transformation

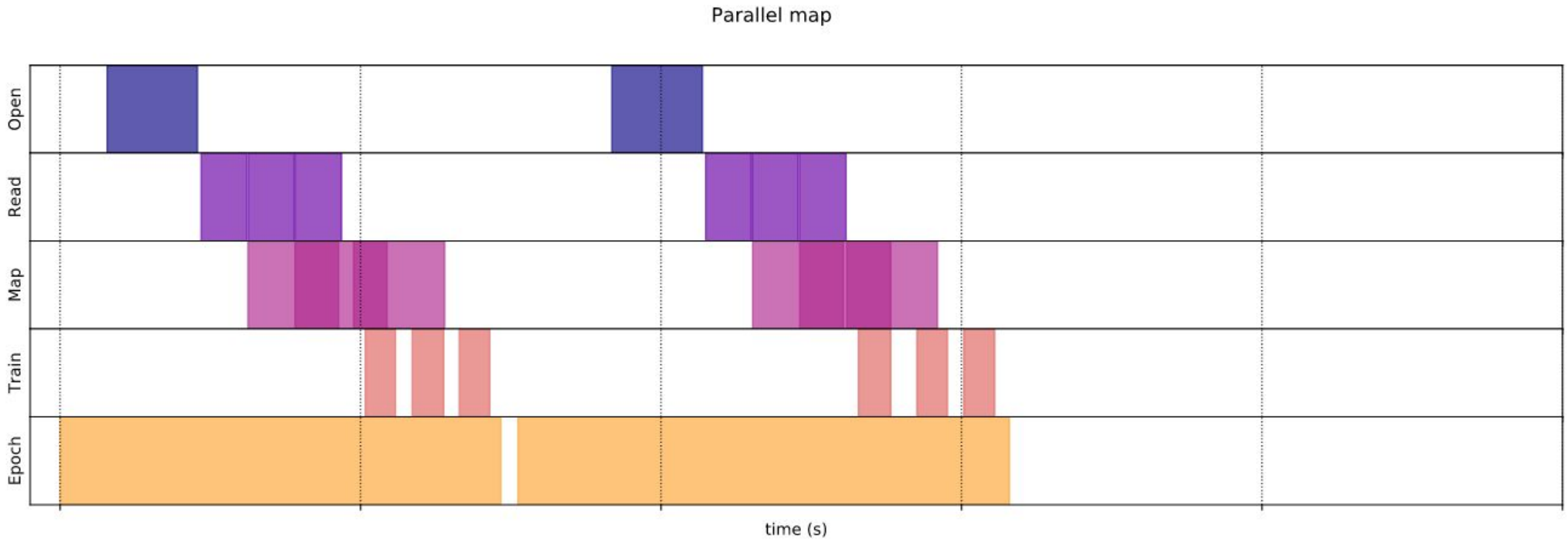
- Typically storage format on the spinning disk is different than what we need for training
  - Disk: PNG format: pixel color is captured with 3 x 1-byte integers
  - Training: TF Tensor: pixel color is 3 x float32s or float16s
- Transformation process is referred to as “mapping”



https://www.tensorflow.org/guide/data_performance

Large Data Sets: Parallel Fetching, Mapping and Training

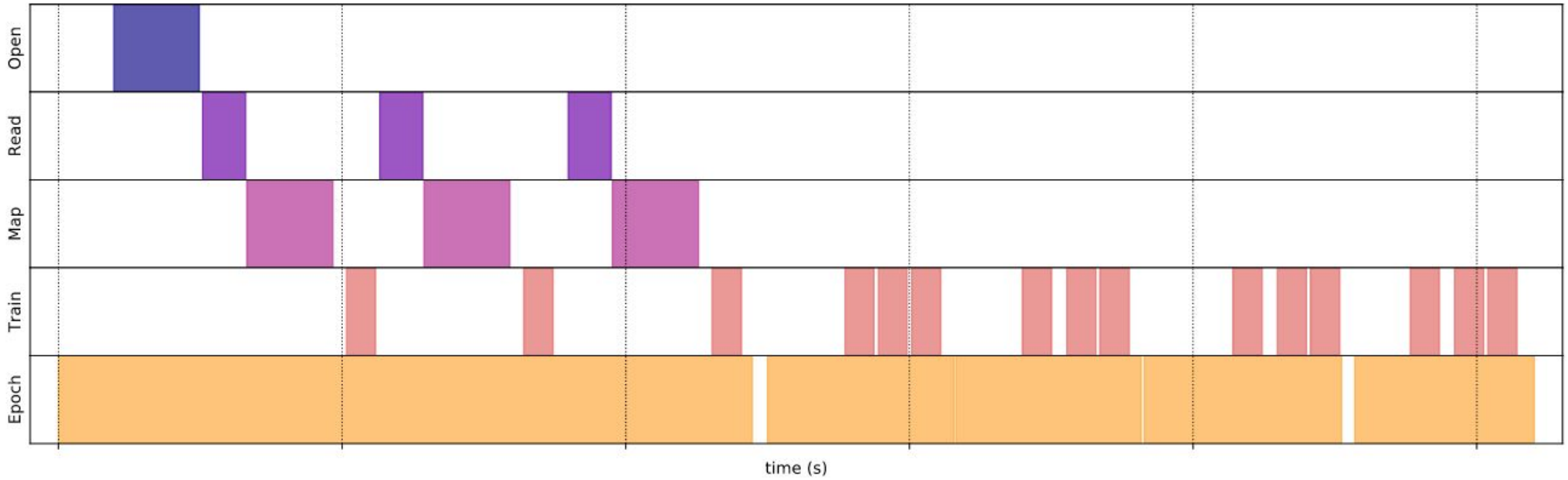
- One or more threads dedicated to fetching and mapping



https://www.tensorflow.org/guide/data_performance

Large Data Sets: Caching

- After loading/mapping data, store in a cache so subsequent accesses are much faster
- In TensorFlow, can cache to RAM or to Disk (we use local SSD)



Class: tensorflow.data.Dataset

- TF Datasets act like generators:
  - Implement a ‘next’ type method that produces the ‘next’ sequential element
  - Will signal if you have reached the end of the data set
  - model.fit() will iterate over each element of a Dataset for training purposes
    - model.evaluate(), model.predict(), too
  - Input side: some other sequence of items (often another Dataset)

- Different TF Dataset methods for:
  - Mapping data
  - Buffering
  - Shuffling
  - Caching
  - Batching

Representing Metadata with Pandas Dataframes

Dataframe: 2D table

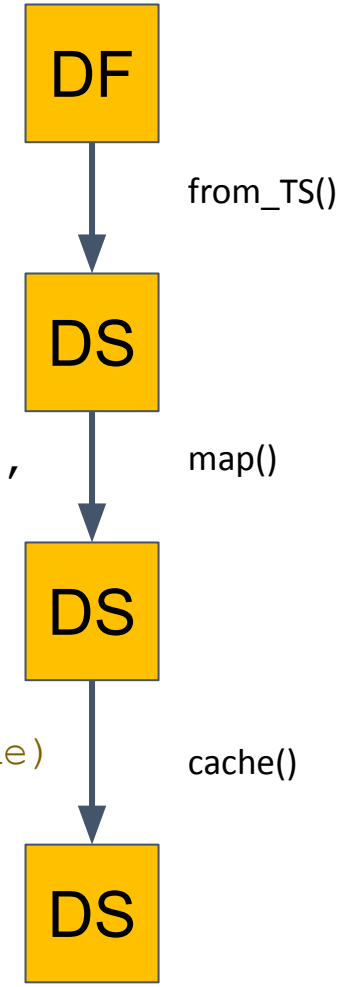
- Rows: single examples
- Columns: different properties for the examples
  - Image file path
  - Class
  - Other information (e.g., timestamp, location)
- Pandas Dataframe implements a lot of database-like operations that make it easy to organize data in many different ways
  - Select all daytime rows
  - Select all rows for a given class, example type…
  - Shuffle the rows

Example: DF Describes Images -> Dataset

```
# Convert DF with a file name and a class label to a dataset
ds = tf.data.Dataset.from_tensor_slices(df[["filename", "class"]].to_numpy())

# For each DF row, create a TF Tensor pair: rows x cols x 3  AND class
ds = ds.map(lambda x: tf.py_function(func=prepare_single_example, inp=[base_dir, x],
                                     Tout=(tf.float32, tf.int8)),
                                num_parallel_calls=4)

# Cache the data set (cache_location = path to local SSD; dataset_name = unique name)
ds = ds.cache('%s/cache_%s'%(cache_location, dataset_name))
```



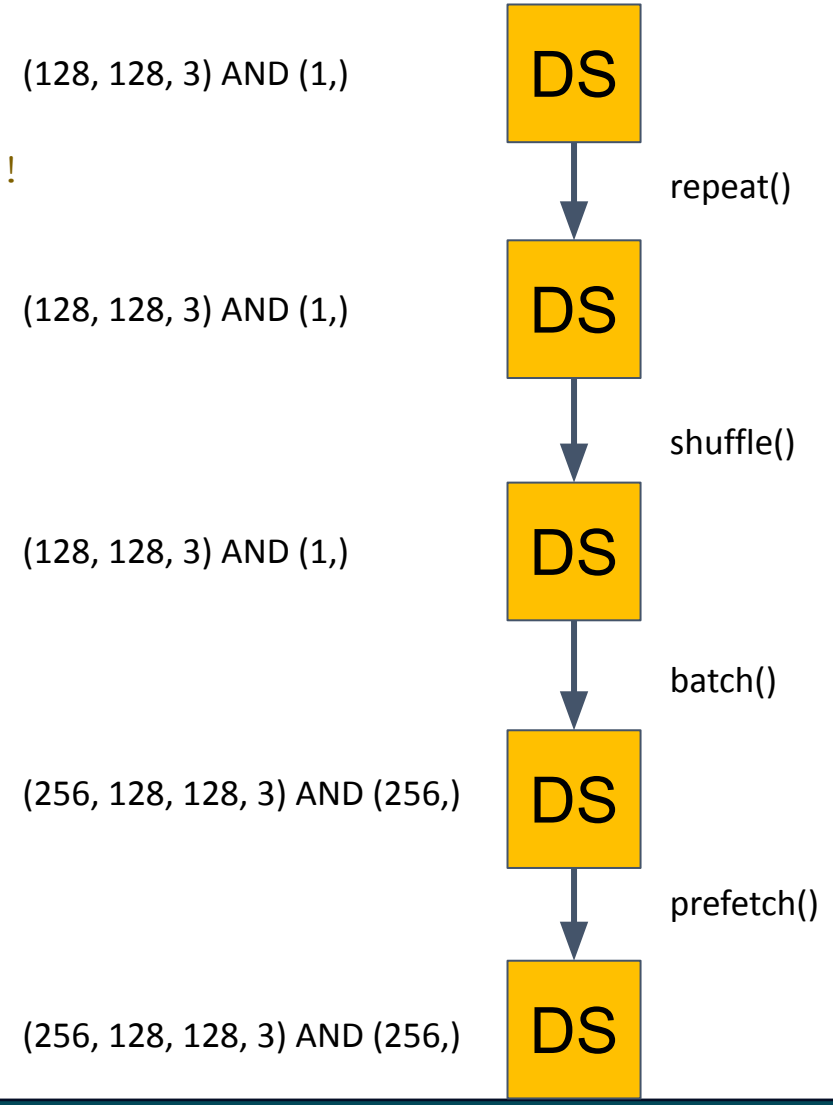
Example Continued

```
# Optionally repeat the data set indefinitely.  Use with caution!
if repeat:
    ds = ds.repeat()

# Pseudo shuffle the dataset (buffer size = 100)
ds = ds.shuffle(100)

# Batch individual examples into groups of 256
ds = ds.batch(256)

# Prefetch batches so we can be ready for requests
ds = ds.prefetch(2)
```



Mapping Function Example

```
def prepare_single_example(base_dir: str, example: np.array) -> [tf.Tensor]:
    # example[0]: string file name
    # example[1]: string class index: “d” where d is a digit
    fname = example[0]

    # Extract Class number
    cl = example[1]
    cl = tf.strings.to_number(cl, out_type=tf.int8)

    # Load image from file system
    img = load_single_png_image(base_dir, fname)

    return img, cl
```

Mapping Function Example

```
def load_single_image(base_dir: str, fname: str) -> tf.Tensor:
    # Implementation uses all TF operators -> can be mapped to GPU

    # Load raw data from file
    image_string = tf.io.read_file(base_dir + "/" + fname)

    # Interpret it as a PNG file
    image = tf.image.decode_png(image_string, channels=3)

    # Convert to standard TF Tensor format
    image = tf.image.convert_image_dtype(image, tf.float32)

    return image
```

Advanced TF Datasets

- Combining multiple Datasets
  - sample_from_datasets(): sample based on a probability distribution from the child Datasets
    - Can use to oversample classes with a small number of examples
  - choose_from_datasets(): iterate through each child Dataset, taking one sample

- Models that take as input multiple images:
  - batch() or choose_from_datasets() to put together the K images into a single example input

- Repeating Datasets: tread carefully here
  - model.fit(): must set steps_per_epoch to something other than None (the default)

Caching the Cache

The Dataset.cache() object:

- As the underlying data are read in and converted, the data are all stored in a single cache file (+ an index file) - on the `$LSCRATCH` SSD
- What you get:
  - Your first pass through your entire dataset still requires all of the data to be fetched from spinning disk (and across network)
  - In subsequent passes through the data set, the data will be taken from the cache file on the SSD instead of over the network

Notes on Caching

- `$LSCRATCH` is allocated to your job – just for its lifetime
- Each node has a different size SSD
- The space available on your `$LSCRATCH` is proportional to the fraction of threads that you reserve on the node
  - #SBATCH --cpus-per-task=20
  - Different nodes also have different numbers of threads available

Summary: TF Datasets

- Not static objects
- Instead:
  - Constantly producing “next” items
  - Backfilling with their input Dataset (or other sequence of items)
- Serve as inputs directly into Keras Model objects:
  - model.fit()
  - model.predict()
  - model.evaluate()

Tuning to Maximize GPU Utilization

- Batch size tuning
  - Increase the size of the batches until you fill the GPU memory
- Caching
  - Cache to the directory given by SLURM environment variable `$LSCRATCH`
- Prefetching
- tune number of threads for operations with tf.data.AUTOTUNE:
  - .prefetch(tf.data.AUTOTUNE)
  - .map(..., num_parallel_calls=tf.data.AUTOTUNE)
  - .batch(..., num_parallel_calls=tf.data.AUTOTUNE)
  - Tread carefully with AUTOTUNE - there are some bugs…

Why Do All These Things?

One example: image classification task

- 160K images: shape: 128x128x3
- 10 Classes
- CPU vs tuned multi-GPU implementation
  - CPU-only requires 50-100x more wall clock time