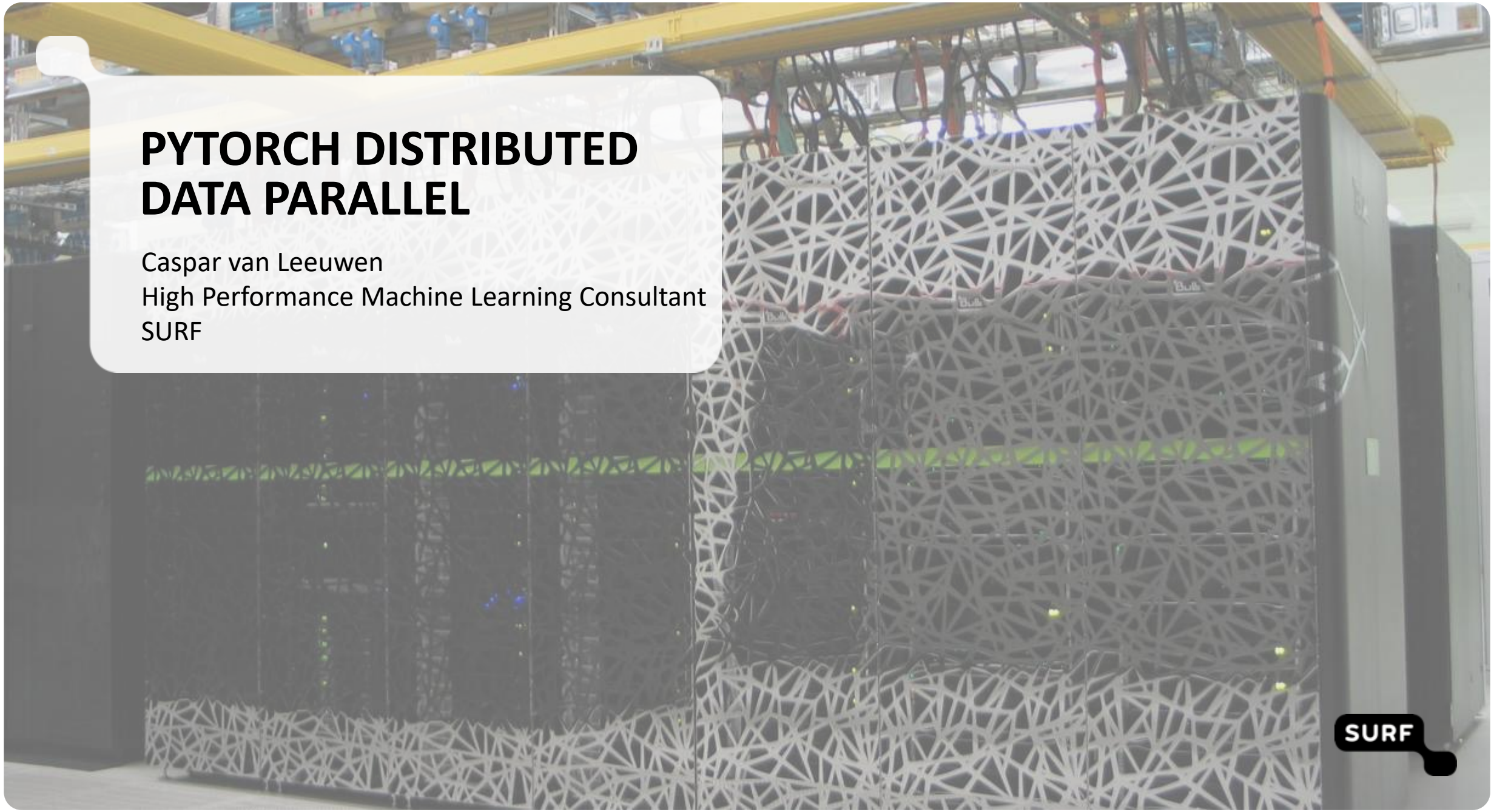


PYTORCH DISTRIBUTED DATA PARALLEL

Caspar van Leeuwen
High Performance Machine Learning Consultant
SURF

Material

The material for this session (slides & exercise) can be obtained from GitHub:

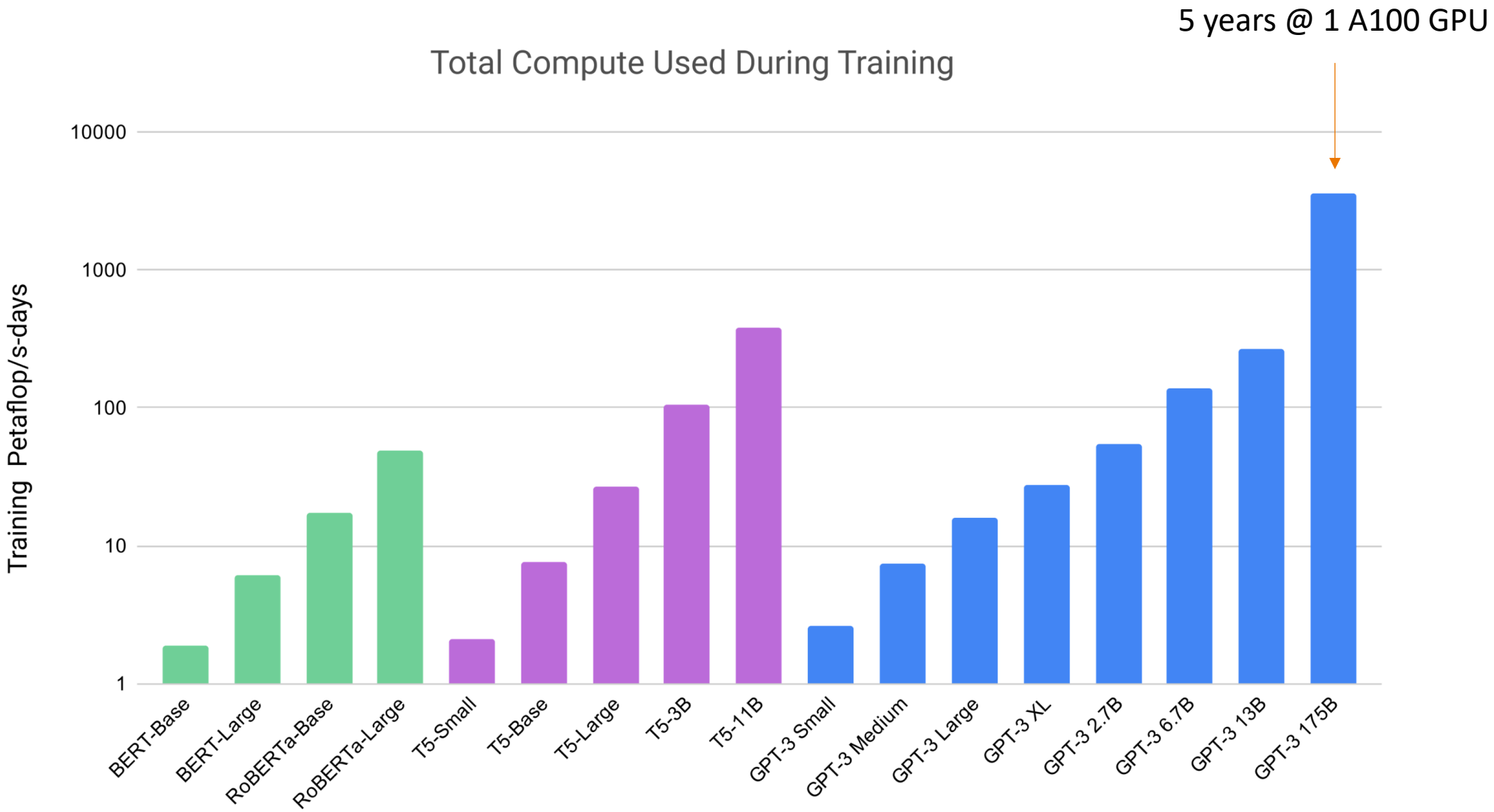
git clone https://github.com/casparvl/PyTorch_DDP.git

Goal

Understand…

- What data-parallel training *is*
- How to use PyTorch Distributed Data Parallel
- (A little of) what PyTorch DDP does ‘under the hood’

Parallelization: why?



Parallelization: why?

Faster trainings …

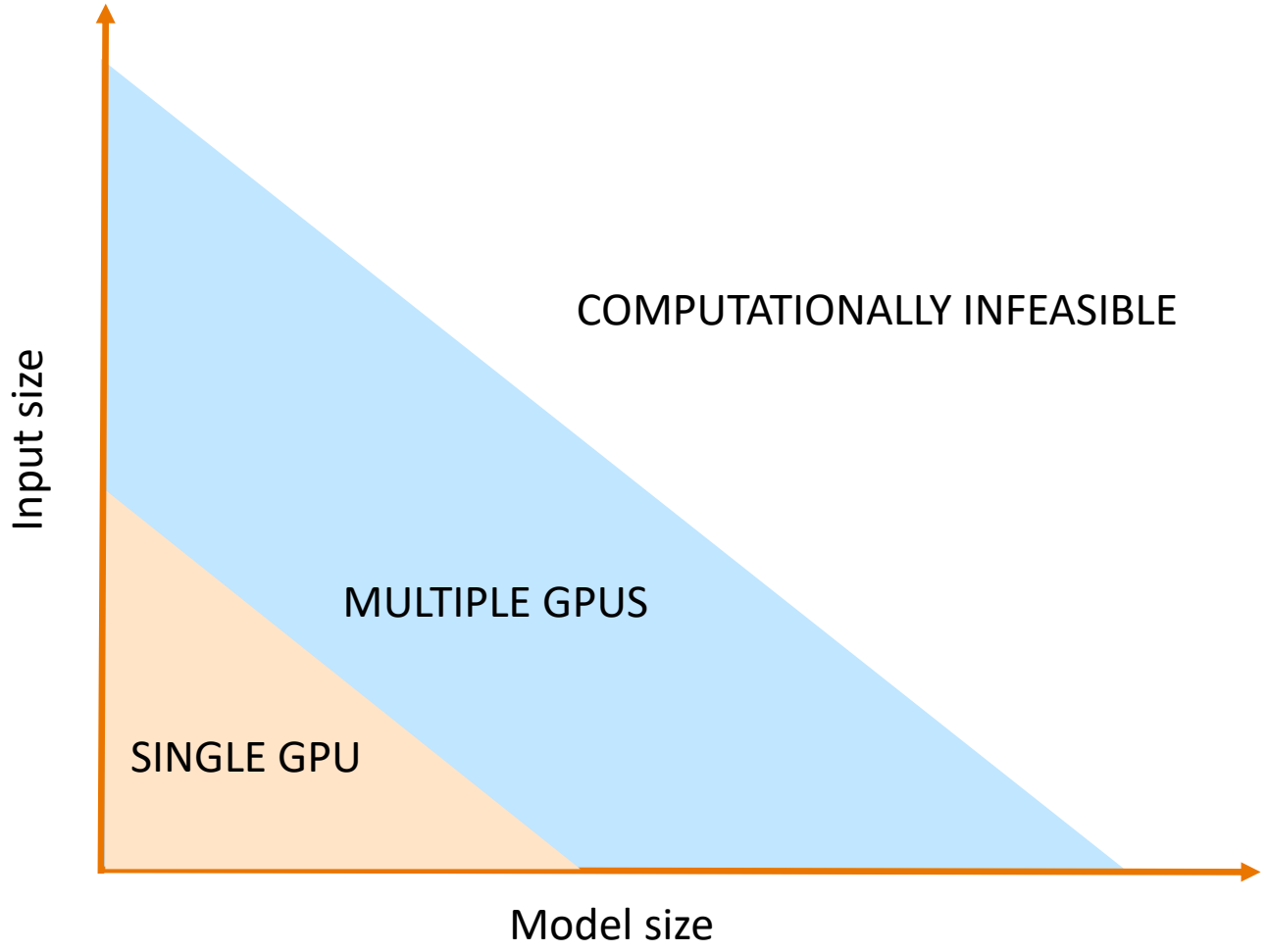
- Enables learning on larger datasets
- Enables improved accuracy through better hyperparemeter tuning
- Enables larger, more complex models

But before we go parallel…

Optimize your serial code!

- Make sure you exploit the hardware features of your GPU (e.g. reduced precision, tensor cores, etc)
- Compile your models with torch.compile
- Avoid unnecessary CPU-GPU transfers
- Profile your code to identify bottlenecks (e.g. I/O)
- Check out https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

Parallelization: when?



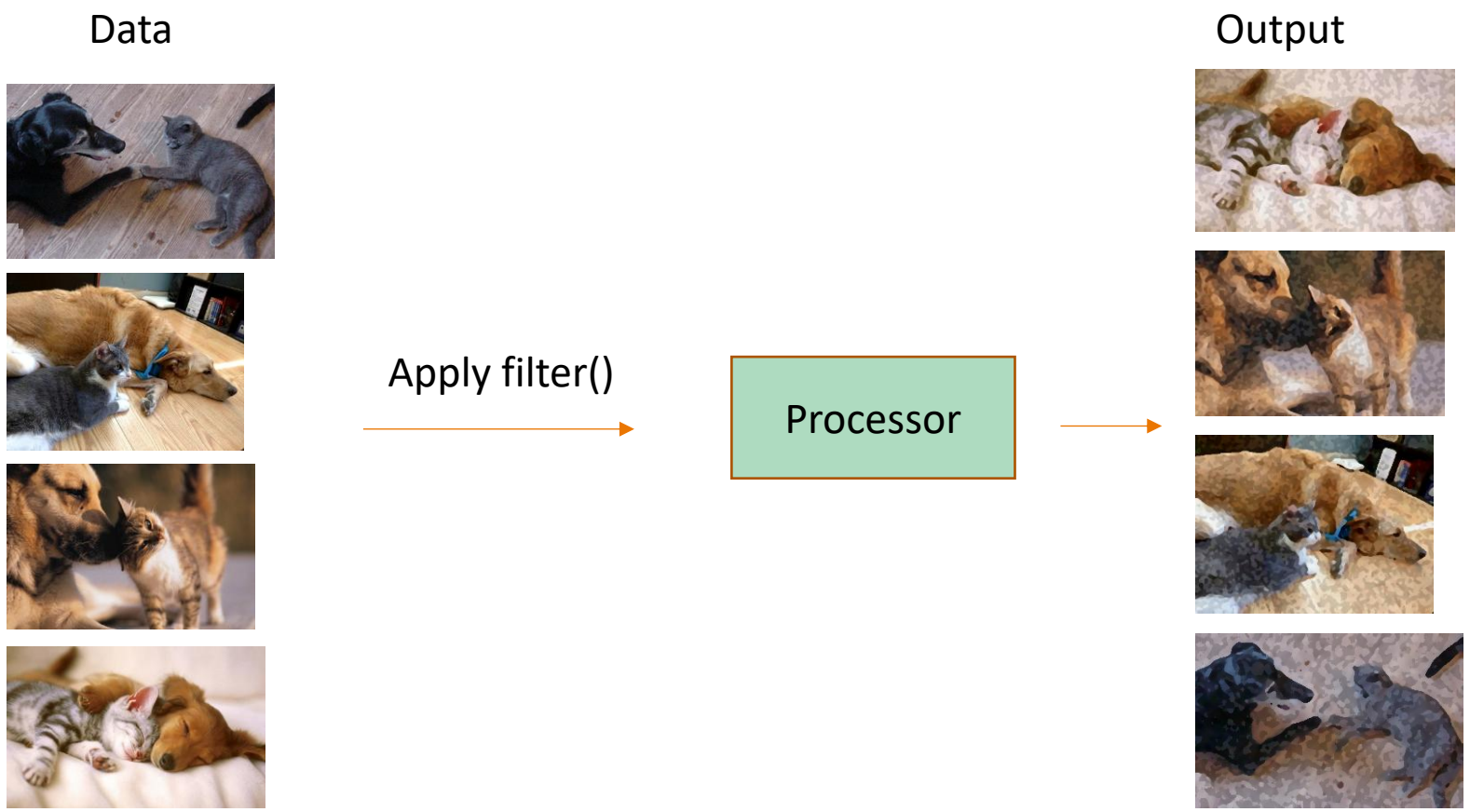
Parallelization: the basics

What is parallel computing?

- Multiple processors or computers working on a single computational problem

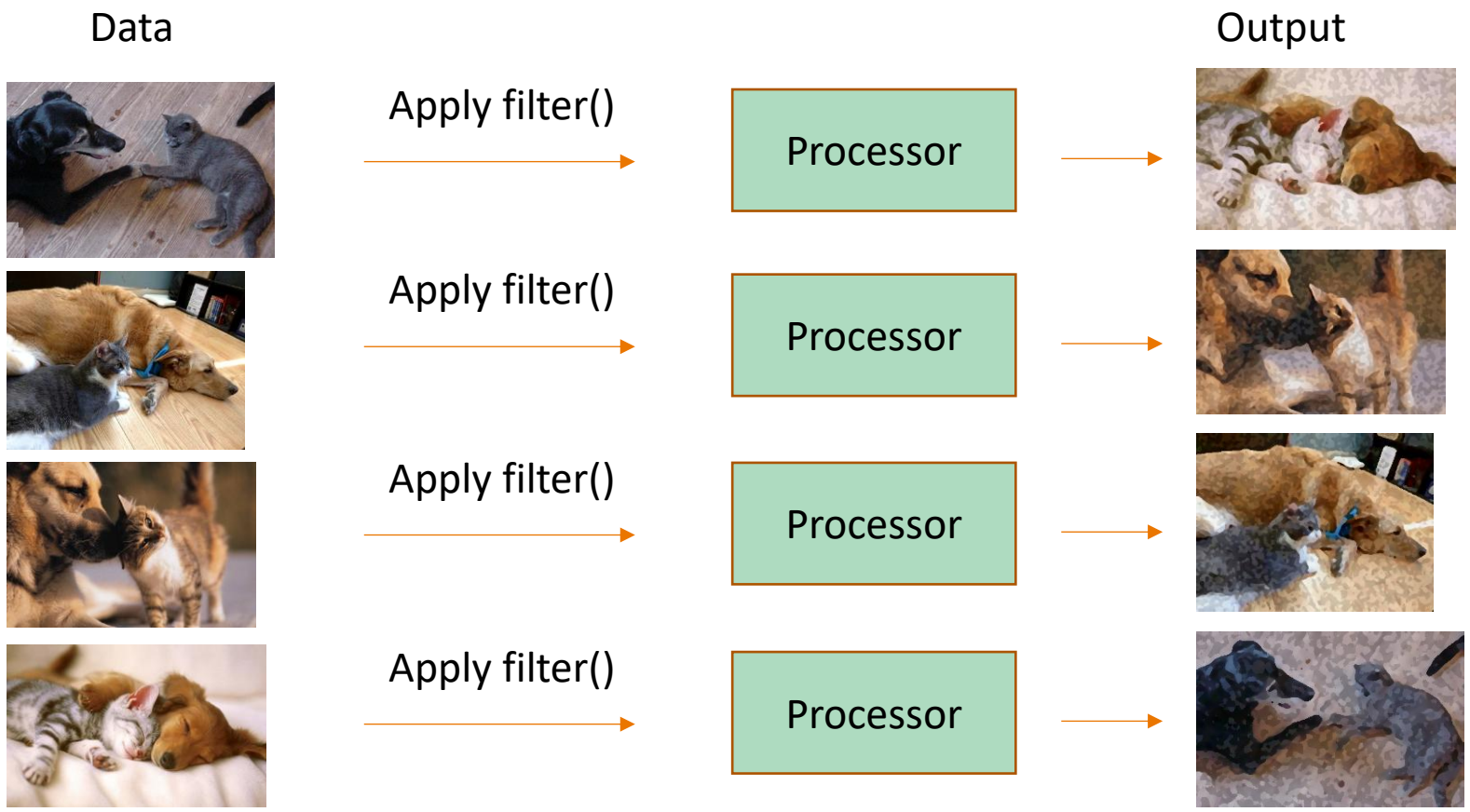
Parallelization: the basics

Serial computing



Parallelization: the basics

Parallel computing



Parallelization: the basics

Benefits:

- Solve computationally intensive problems (speedup)
- Solve problems that don’t fit a single memory (multiple computers)

Requirements:

- Problem should be divisible in smaller tasks

Considerations for problem decomposition

- How can I limit the need for communication? (smaller overhead = bigger speedup)
- How does my partitioning affect memory consumption?
- How does partitioning affect my algorithm, e.g. convergence behavior?

Types of parallelization

What types of parallelization exist?

- Instruction level parallelism → Single instruction operating on e.g. entire vectors / matrices
- Embarrassingly parallel → E.g. hyperparameter grid search
- Data parallel → Topic of this session ☺
- Model parallel
- Hybrid data/model parallelism
- Tensor parallel
- Pipeline parallelism

Typically: PyTorch, TensorFlow, …, takes care of this

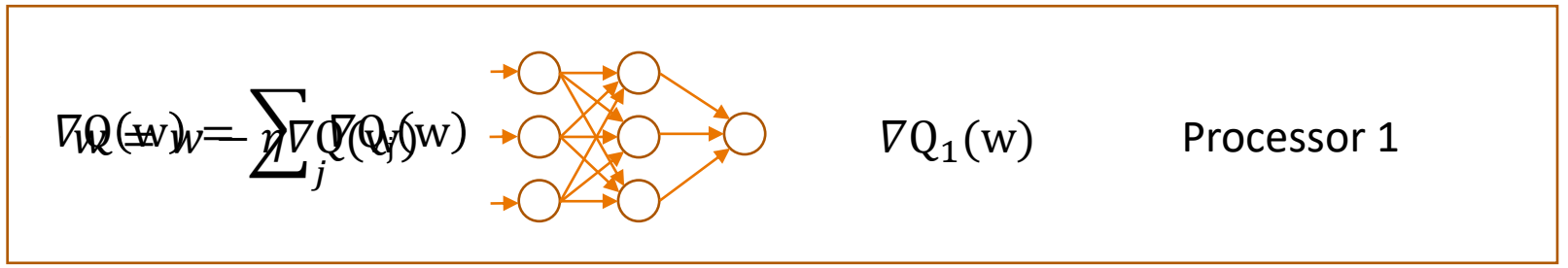
Covered in upcoming sessions in this course

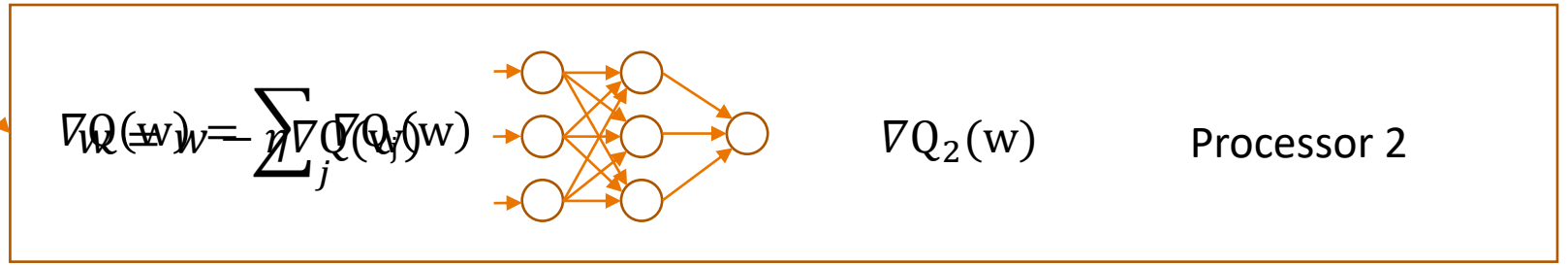
Data Parallelism

Train a single model, single set of hyperparameters, but ***faster***

- Split the data over multiple processors (CPUs/GPUs)
- Each processor holds an identical copy of the model
- Forward pass: calculated by each of the workers
- Backward pass: gradients computed (per worker)
- Communicate and aggregate gradients
- Model update

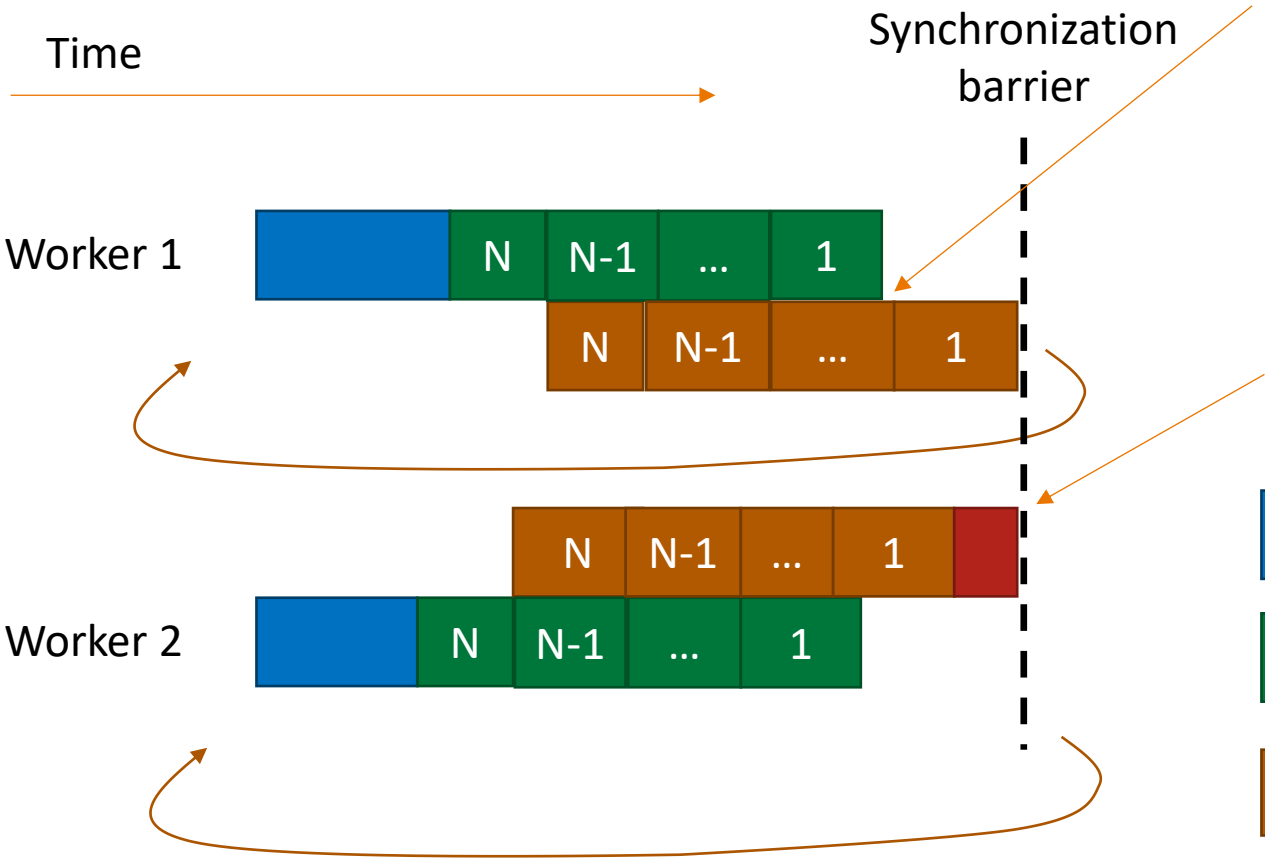
NB: model is still identical, since initial model *and* update were identical!





Distributed Data Parallel

A different view…



Pro tip:
- Overlap communication and computation (don't waste compute cycles waiting for communication!)
- *Most* (distributed) DL frameworks already take care of this for you ☺ => so does PyTorch DDP!

Pro tip 2:
- Make sure all workers are equally fast!

Forward pass

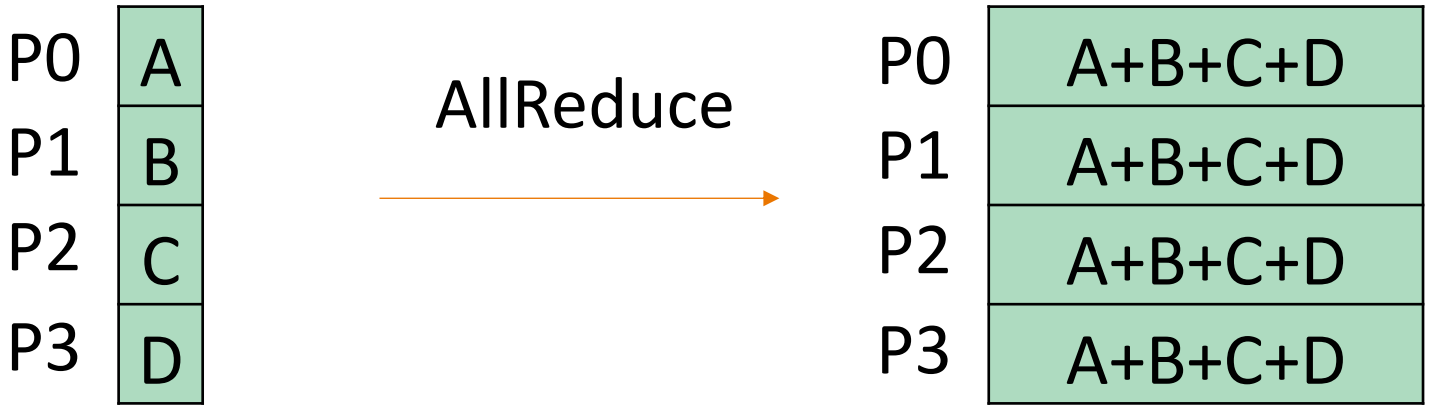
N — Backward pass, compute gradient of layer N

N — Backward pass, communicate gradient N

Wait

How does PyTorch aggregate gradients?

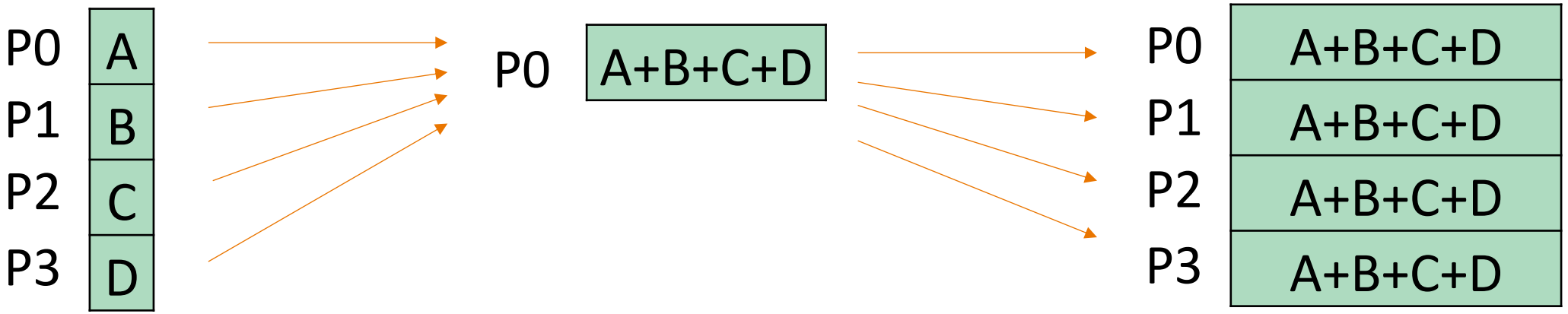
The AllReduce operation (with ‘sum’ as reduction operator) is defined as:



But this does not define how the operation should be implemented!

An AllReduce algorithm

- Example: a (typically inefficient) AllReduce operation could implemented like this



So: what algorithm does PyTorch use?

That is up to the communication backend! (https://discuss.pytorch.org/t/multiple-node-multiple-worker-allreduce/98869)

There are three communication backends to PyTorch:

- Gloo => Ring AllReduce
- NCCL => Tree and Ring Allreduce implementions, automatically selected
- MPI => Up to the MPI library

So: what communication backend should I use?

Rule of thumb (https://pytorch.org/docs/stable/distributed.html#backends)

- (Nvidia) GPU host: NCCL
- CPU host with infiniband + IP over IB enabled: Gloo
- CPU host with infiniband; no IP over IB: MPI (have to build PT from source!)
- CPU host, no infiniband: Gloo

Why NCCL is the best backend for Nvidia GPUs

- NCCL implements multiple algorithms for AllReduce
- NCCL *knows* the connectivity between the GPUs involved (e.g. are the connected by NVLink, are they in the same node, etc)
  - Try `nvidia-smi topo -m`
- NCCL makes an *informed choice* on the best algorithm based on the connectivity

Intermezzo: why the PyTorch dispatcher is a blessing (and a curse)

Torch call stack is *very* complex! E.g.

- You call Torch function
- Which calls other Python (Torch) functions
- Which call C++ (Torch) functions
- Which call low level libraries, often hardware specific (MKL/MKL-DNN, CuBLAS/cuDNN, NCCL, …)

The blessing

- The low level libraries usually do things in a *very* well-optimized way!

http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/

SURF

Intermezzo: why the PyTorch dispatcher is a blessing (and a curse)

The curse

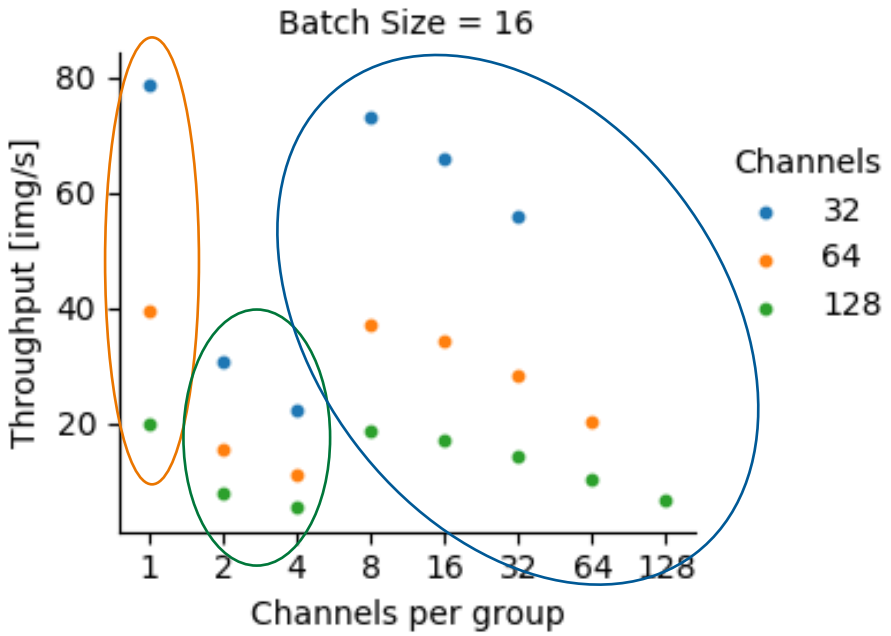
- High degree of abstraction makes it hard to predict for the programmer what will happen
- Your choices matter! (but it is sometimes *very hard* to see how!)

Grouped 2D convolution on Intel CPUs with MKL

Uses optimized AVX512-based algorithm

Uses generic matrix-multiply algorithm

Uses optimized AVX2/AVX512-based algorithm



https://medium.com/@casparvanleeuwen/why-efficientnet-is-not-very-computationally-efficient-505fc08087b9

Say something about DataParallel vs DistributedDataParallel

- DataParallel
  - Single process, multi-threaded
  - Limited to single machine
  - Typically slower than DistributedDataParallel due to Python GIL (Global Interpreter Lock)
  - Cannot be combined with model parallel
- DistributedDataParallel
  - Multiple (possibly multi-threaded) processes
  - Supports multiple machines
  - Can be combined with model parallel

See also: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html#comparison-between-dataparallel-and-distributeddataparallel

How to go from regular PyTorch code to PyTorch with DDP

- Step 1: Constructing the process group
- Step 2: Constructing the DDP model
- Step 3: Distributing input data
- Step 4: Send data to correct GPU
- Step 5: Alter saving model checkpoint (optional)
- Step 6: Running the code: spawning multiple processes

See also: https://pytorch.org/tutorials/beginner/ddp_series_multigpu.html

Assumptions in this tutorial

- We assume you are running on a SLURM cluster, and pass the `--cpus-per-task` and `--ntasks-per-node` options when scheduling your job
- Each process in the DDP training is uniquely identified by its *rank*
- We do pure DDP (i.e. we don't combine with model parallelism). Thus, each rank uses 1 GPU
- We will use the concept of local rank, a unique identifier of the process *on that node.* E.g. if you have global ranks 0-7 and launch 4 processes per node: global ranks 0, 1, 2, 3, 4, 5, 6, 7 will have local ranks 0, 1, 2, 3, 0, 1, 2, 3, respectively.
- We want each local rank to use the GPU with the same ID as the local rank, i.e. local rank 0 will use `gpu:0`

Step 1a: Constructing the process group

- Why: parallel launchers can launch multiple processes, but these processes need to know about each others existence!
- How: by calling `init_process_group`, but we need to know the `world_size` and `rank` first. We can get those easily in a SLURM allocation!

```
def ddp_setup():
    world_size = int(os.environ["SLURM_NTASKS"])
    rank = int(os.environ["SLURM_PROCID"])

    dist.init_process_group("nccl", rank=rank, world_size=world_size)

    return rank
```

- We've chosen NCCL as backend, as per recommendation on https://pytorch.org/docs/stable/distributed.html#backends
- N.B. There are multiple initialization methods for `init_process_group`, see https://pytorch.org/docs/stable/distributed.html#torch.distributed.init_process_group

SURF

Step 1b: Constructing the process group

- Call `set_device` before `init_process_group` to avoid hangs and/or multiple processes ending up on the default GPU (`GPU:0`)
- Note that your code *may* work correctly without calling `set_device`, but it is recommended to avoid potential issues (see https://pytorch.org/tutorials/beginner/ddp_series_multigpu.html#constructing-the-process-group)
- We need a `local_rank` to determine which GPU to set as device

```
def ddp_setup():
    world_size = int(os.environ["SLURM_NTASKS"])
    rank = int(os.environ["SLURM_PROCID"])
     local_rank = int(os.environ["SLURM_LOCALID"])

    torch.cuda.set_device(local_rank)
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

    return rank, local_rank
```

Step 2: Constructing the DDP model

- Why: after the backward pass has been done on a layer, the gradients should be aggregated between all ranks
- How: by wrapping our existing (serial) model in the DDP class and telling it on which GPU this model should be kept. We can reuse the `local_rank` variable from earlier.

```
ddp_model = DDP(model, device_ids=local_rank)
```

Step 3a: Distributing input data

- Why: we don't want our N processes to do *the same work* N times, we want to *divide* our work into N smaller pieces – and have each process do one piece of work
- How: by using the DistributedSampler

```
train_loader = torch.utils.data.DataLoader(
    dataset=…,
    batch_size=…,
    shuffle=False,  # We don't shuffle
    sampler=DistributedSampler(train_dataset), # Use the Distributed Sampler here.
)
```

- See https://pytorch.org/docs/stable/data.html?highlight=distributedsampler#torch.utils.data.distributed.DistributedSampler

Step 3b: Distributing input data

- Every time we start new epoch, we have set a new seed for the distributed sampler (otherwise it would go through the samples in the same order every epoch)

```
for epoch in range(0, max_epochs):
    train_loader.sampler.set_epoch(epoch)
    train(…)  # Call actual training function that loops over batches
```

- See https://pytorch.org/docs/stable/data.html?highlight=distributedsampler#torch.utils.data.distributed.DistributedSampler

Step 4: Send data to correct GPU

- Why: our input data needs to be sent to the GPU that is assigned to our (local) rank
- How: by using `.to(local_rank)`

```
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(local_rank), target.to(local_rank)
    …
```

Step 5: Alter saving model checkpoint (optional)

- Why: the DDP model's `state_dict` contains the original model as a module. Thus, all the original model's parameters are prefixed with `module.`
  - *If* you want to load this into another DDP-wrapped version of your model, that's fine
  - *If* you want to load this into an unwrapped version of your module, you'd have to strip the `module.` prefixes.
- How: if you want to load into another DDP-wrapped version of your model, keep:

```
torch.save(model.state_dict(), "mnist_cnn.pt")
```

If you want to load into a serial version of your model, instead use:

```
torch.save(model.module.state_dict(), "mnist_cnn.pt")
```

Step 6a: Running the code: spawning multiple processes

- Why: DDP is intended for multiple processes working together – so we need to start the code multiple times
- How: using srun as a parallel launcher after setting `MASTER_ADDR` and `MASTER_PORT`

```
#!/bin/bash
#SBATCH --job-name=mnist-ddp --partition=gpu --time=10:00
#SBATCH --ntasks=8 --ntasks-per-node=4 --gpus-per-node=4 --cpus-per-task=18

# Pick a quasi-random port, and use the first node in the allocation as master node
export MASTER_PORT=$(expr 10000 + $(echo -n $SLURM_JOBID | tail -c 4))
master_addr=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -n 1)
export MASTER_ADDR=$master_addr

srun python mnist_classify_ddp.py <script_arguments>
```

Step 6b: Running the code: spawning multiple processes

- Why: DDP is intended for multiple processes working together – so we need to start the code multiple times
- How: using torchrun as a parallel launcher

```
#!/bin/bash
#SBATCH --job-name=mnist-ddp --partition=gpu --time=10:00
#SBATCH --ntasks=2 --ntasks-per-node=1 --gpus-per-node=4 --cpus-per-task=18

head_node_ip=$(hostname --ip-address)
OMP_NUM_THREADS=18 srun torchrun --nproc_per_node=4 --nnodes 2 --rdzv_id 1234 --
rdzv_backend c10d --rdzv_endpoint $head_node_ip:29500 mnist_classify_ddp_torchrun.py
<script_arguments>
```

- Where `mnist_classify_ddp_torchrun.py` has some small modifications compared to the original script…
- Note the 2 tasks & 1 task per node: `srun` launches `torchrun` once per node, and `torchrun` itself then launches 4 processes

Step 6b: Running the code: spawning multiple processes

- Where `mnist_classify_ddp_torchrun.py` has some small modifications compared to the original script:

```
def ddp_setup()
    # These get set by torchrun, let's get them from the environment:
    local_rank = int(os.environ["LOCAL_RANK"])
    rank = int(os.environ["RANK"])

    # no need to pass them explicitly anymore to the init_process_group
    dist.init_process_group(backend="nccl")
    return rank, local_rank
```

- N.B. we still retrieve `local_rank` and `rank` from the environment because we use them elsewhere in the code, e.g. in our `.to(local_rank)` statements

Step 6: Running the code: spawning multiple processes

- You could even use things like mpirun, or other ways of spawning multiple processes, *as long as you adapt your code accordingly* so that it is able to determine the rank and local rank for the current process
- So… which should I use? Depends on personal preference, needs, etc…
- `srun`: less abstraction. Probably easier for your HPC site to help you. More clear and explicit what is happening.
- `torchrun`: more abstraction, but also more functionality. Can automatically restart if single ranks fail (with srun, you'd have to manually submit a new job & restart from checkpoint). Probably easier for the Torch community to help you. See https://pytorch.org/tutorials/beginner/ddp_series_fault_tolerance.html
- Word-of-warning: I saw a 5% (single node) – 25% (multinode) performance decrease when using `torchrun` on our mnist example code, for which I haven't found a cause yet => one of the downsides of more abstraction…

Material

The material for this session (slides & exercise) can be obtained from GitHub:

git clone https://github.com/casparvl/PyTorch_DDP.git

Exercise: implement DDP on serial MNIST classification code

- You'll find the serial code in `mnist_classify.py`
- First, create an interactive allocation with 2 GPUs: `salloc --partition=boost_usr_prod --account=tra25_castiel2 --time=10:00 --nodes=1 --ntasks-per-node=2 --gpus-per-node=2 --cpus-per-task=8 --reservation=s_tra_castiel2`
- Load the software environment: `module load python/3.11.6--gcc--8.5.0` and `source $WORK/PyTorchDDP/pt_ddp_venv/bin/activate`
- Try to run the serial code: `srun -n 1 python3 mnist_classify.py --batch-size 128 --epochs 5 --data-dir $WORK/PyTorchDDP`
- Once you have a successful run, try to go through the 6 steps to alter the code, and your job script, to make things run in parallel. You can run your modified script on 2 GPUs using: `srun -n 2 python3 <my_DDP_script.py> --batch-size 128 --epochs 5 --data-dir $WORK/PyTorchDDP`
- Hint: to accurately time the code, add a `dist.barrier()` *before* every `time.time()` invocation. WARNING: you'd never to this in production runs, as unnecessary barriers may increase your overall computation time – this is just so we can time *this section* of the code accurately!
- Do you see any speedup? How much? Is it enough to justify using 2 times more resources?

Bonus Exercise 1: Try a scaling run on 1, 2, 4, and 8 GPUs

- First, release your 2-GPU allocation from the previous exercise by running `exit`
- Then, create a job script for your 2-GPU run. List the same arguments to SLURM in your #SBATCH statements, then load the module environment, then launch your 2-GPU run with `srun`.
- Submit your job script and verify that it completes successfully
- Now, scale up: alter your job script so that it runs on 4 and 8 GPUs. Note: we have 40 nodes (120 GPUs) reserved for 50 students. The more students release their 2-GPU interactive allocation, the faster these jobs will be scheduled…
- Determine the strong scaling speedup and efficiency on 1, 2, 4 and 8 GPUs and create a scaling plot. See https://hpc-wiki.info/hpc/Scaling#Strong_Scaling and https://hpc-wiki.info/hpc/Scaling#Strong_Scaling_2

Bonus Exercises 2: Try a scaling run on 1, 2, 4, and 8 GPUs

- Try different reduction algorithms by setting `NCCL_ALGO`, see https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-algo . Do you see any performance difference? Why (not)?

Bonus Exercises 3: Try to run with torchrun

- Check https://pytorch.org/docs/stable/elastic/run.html for the arguments to torchrun
- You can use `head_node_ip=$(hostname --ip-address)` to get the IP of the head node of your allocation, and use that as `rdzv-endpoint`
- You will need to set `OMP_NUM_THREADS`, otherwise `torchrun` uses only 1 thread for each `torchrun` process
- To start `torchrun` on multiple nodes, you'd need to wrap it in an `srun` statement. Hint: think about how many tasks per node you'd want `srun` to start and make sure to set that number through `#SBATCH --ntasks-per-node`
- How much performance do you get? How does that compare to what you started with `srun`?

Recap

- Step 1: Constructing the process group => Make sure processes can work together
- Step 2: Constructing the DDP model => Make sure PyTorch automatically aggregates gradients before model update
- Step 3: Distributing input data => Make sure each worker sees a different part of the data (i.e. does *different* work)
- Step 4: Send data to correct GPU => Make sure our data goes to the same GPU as our model
- Step 5: Alter saving model checkpoint (optional)
- Step 6: Running the code: spawning multiple processes => If we want multiple processes to work together, we need to start multiple processes

Recap

Understand…

- What data-parallel training *is*
- How to use PyTorch Distributed Data Parallel
- (A little of) what PyTorch DDP does ‘under the hood’

Further reading

- How to train LLMs at (very) large scale: https://huggingface.co/spaces/nanotron/ultrascale-playbook