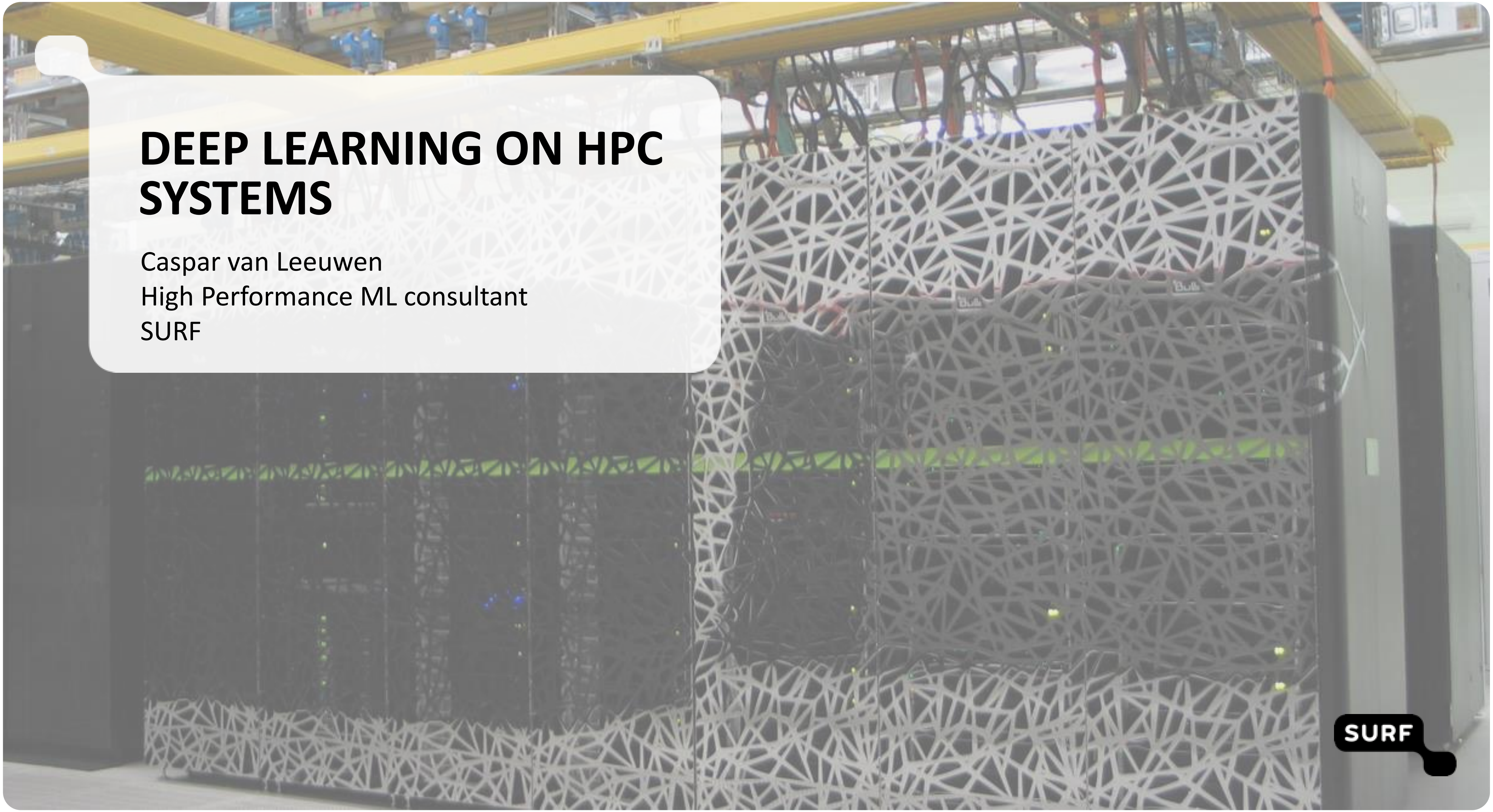


DEEP LEARNING ON HPC SYSTEMS

Caspar van Leeuwen
High Performance ML consultant
SURF

Parallel computing for Deep Learning

Topics

- How to get the most out of your HPC system?
  - Bottlenecks
  - Reduced precision / data types
  - Tensor operations
- Benefits of parallellization
- Parallellization strategies
- Data parallel, synchronous, stochastic gradient descent (SGD)
- Frameworks for distributed deep learning

How to get the most out of your HPC system?

What limits the speed of your computation on a computer?

How to get the most out of your HPC system?

Think about what is the bottleneck in your computation:

- I/O
- Memory bandwidth
- Memory size
- Communication
- Compute (floating point operations per second, FLOPS)

Some context

What?	Time	Equivalent to N CPU instructions
CPU instruction	0.25 ns (@4GHz)	1
GPU instruction	1 ns (@1 GHz)	4
Fetch from L1 CPU cache	0.5 ns	2
Fetch from L2 CPU cache	7 ns	14
Fetch from main CPU memory	100 ns	400
CPU => GPU latency (PCI)	300 ns	1200
GPU ⬄ GPU latency (nvlink)	300 ns	1200
Infiniband network latency	1 µs	4k
Ethernet network latency	10-1000 µs	40k-4M
Seek time (nvme SSD)	20 µs	80k
Seek time (SSD)	200 µs	800k
Seek time (hard disk)	2-5 ms	8-20M
Network File System (NFS) response times	10-40 ms	80-240M

I/O as a bottleneck

Particularly important on HPC systems

- Network filesystems are typically a *shared resource* => other users will dislike you if you hammer the filesystem by reading 1000's of files for 100's of epochs!
  - If your node has local disk, use it to cache your dataset before training => only have to read once from the shared network filesystem
- HPC network filesystems are optimized for bandwidth, not IOPS (I/O operations per second)
  - Accessing 100 MB spread over 1000 files is *much* slower than accessing a single, 100 MB file (also on local disks, but *much* worse on network FS)
  - Use packed file formats (zip, tar, Tfrecords, LMDB, HDF5, npy) to pack multiple samples in a single file

Memory bandwidth

Your processor (CPU/GPU) needs data to work on: it’s hard to keep it fed with data all the time!

- Bad memory access patterns (if bytes that are used consecutively are not layed out consecutively in memory) can reduce your performance
  - Very hard to manage as a programmer. Fortunately, low level numerical libraries (MKL & MKL-DNN on Intel CPU, cuDNN on GPU) take care of this for us!
  - Frameworks (PyTorch/TensorFlow) have support for these libraries
  - Reduced precision datatypes *also* reduce the footprint on memory

Memory size

Memory size is a *hard* bottleneck: if your network is too large, it won’t run

- Use different hardware
  - GPU with more memory
  - CPU usually has a *ton* of memory. Use CPU + data parallel training to still obtain reasonable training speed
- Reduced precision can reduce memory footprint
  - Frameworks (PyTorch/TensorFlow) have support for these libraries
  - Reduced precision datatypes *also* reduce the footprint on memory (will get back to this)

Communication

Communication bottlenecks can be latency or bandwidth. Communcation occurs:

- CPU => GPU transfers
  - Executing many small layers / small batch size => lot’s of communication overhead (latency), not much compute to be done
  - Try larger batch sizes
  - For the last few %: layer fusion (see e.g. torch.quantization.fuse_modules) => causes e.g. a conv layer, batch norm, and relu to be executed as *one* GPU kernel, instead of *three* (overhead only once!)

Communication

Communication bottlenecks can be latency or bandwidth. Communcation occurs:

- GPU ⇔ GPU transfers
    - Within a node: fast if it uses direct connection between GPUs (e.g. NVlink), slow if it has to go through PCI.
    - Between nodes: typically requires a low-latency, high bandwidth network (e.g. Infiniband), otherwise scaleability is poor
    - GPU nodes in Lisa: 4 GPUs (1080Ti, TitanRTX) per node, no Nvlink, no Infiniband. Good for single GPU training. Acceptable for data-parallel training on up to 4 GPUs (i.e. 1 node).
    - GPU nodes in Snellius: 4 GPUs (A100) per node. NVlink, all-to-all connected, infiniband between nodes. Good single node GPU training, but *also* for data-parallel training requiring 10's of GPUs!

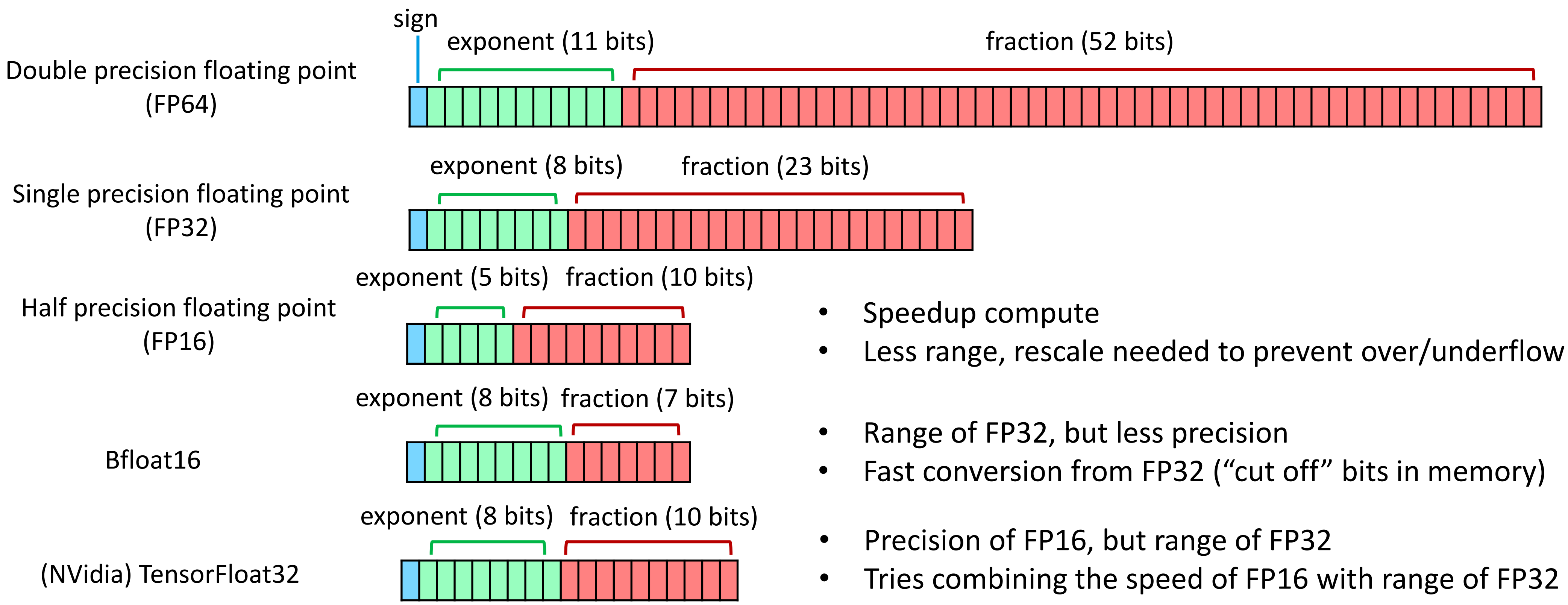
Compute

Congratulations, if compute is your limitation, you did really well!

- Go to high-FLOP devices (GPUs)
- Use reduced precision
- Make sure to use TensorCores (on Nvidia GPUs)

Solutions: reduced precision

Floating point: $(-1)^{sign} \cdot 2^{(exp-127)} * 1.fraction$



- Speedup compute
- Less range, rescale needed to prevent over/underflow

- Range of FP32, but less precision
- Fast conversion from FP32 (“cut off” bits in memory)

- Precision of FP16, but range of FP32
- Tries combining the speed of FP16 with range of FP32

See https://servicedesk.surf.nl/wiki/display/WIKI/Deep+Learning+on+A100+GPUs

Compute

- (signed) N-bit integer: $(-1)^{b_0}(b_1 \cdot 2^n + b_2 \cdot 2^{n-1} + \cdots + b_n \cdot 2^0)$
- 1-bit: bool
- 4-bit: semiocted
- 8-bit: int8, uint8, byte
- 16-bit: uint16, int16, word, short
- 32-bit: uint32, int32, double word, long
- 64-bit: uint64, int64, quadword, long long
- Not so common in machine learning

Solutions: tensor operations

Tensor operations

- Nvidia: tensor cores
- Intel: Advanced Matrix Extensions (AMX) instruction
- Is a *single* CPU/GPU instruction, executed in one clock cycle
- $D = A \times B + C$ for matrices of limited sizes

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{matrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{matrix}$$

See https://servicedesk.surf.nl/wiki/display/WIKI/Deep+Learning+on+A100+GPUs

Solutions: tensor operations

Theoretical performance on A100 GPU

- $D = A \times B + C$
  - Input: datatype of A & B
  - Accumulator: datatype of C & D
  - Warning: Tensor Cores do *not* support using FP32 for A & B
  - NVIDIA libraries (cuDNN, CUBLAS) will automatically cast FP32 to TF32

Input (A & B)	FP64	TF32	FP16	FP16	Bfloat16	INT8	INT4	Binary
Accumulator	FP64	FP32	FP32	FP16	FP32	INT32	INT32	INT32
Performance (TOPS)	19.5	156	312	312	312	624	1248	4992

See https://servicedesk.surf.nl/wiki/display/WIKI/Deep+Learning+on+A100+GPUs

Potential for speedup

Reduced precision helps

- Communication bottleneck
- Memory bottleneck
- Computation bottleneck (if vector or tensor instuctions are used)

How to figure out your bottleneck?

Profiling!

- Profilers show the *execution pattern* of your application. They will tell you how long you are spending in certain functions.
- A profiler can provide insight into whether you’re waiting for compute, data transfer, etc
- Reading/understanding profiling results can take a bit of skill!

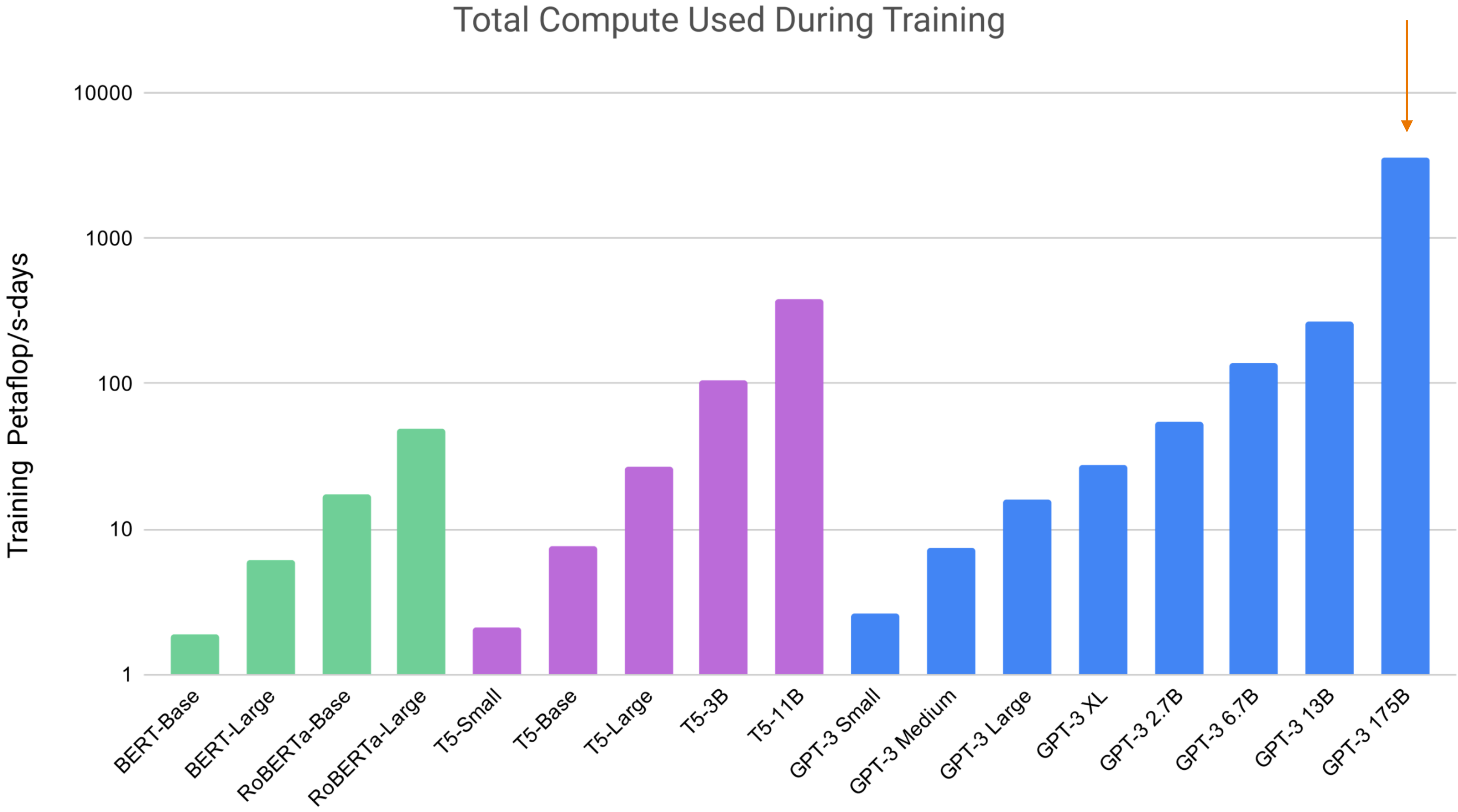
What’s in SURF systems?

- Snellius ‘gpu’ partition: 4 x A100, 40 GB GPU memory each
  - 4 x A100, 40 GB GPU memory
  - 156 TFLOPS @ TF32, 312 TFLOPS ‘mixed precision’ (A, B @ FP16, C & D @ FP32)
- Lisa ‘gpu’ partition:
  - 4 x 1080Ti, 11 GB GPU memory
  - 11.34 TFLOPS @ FP32
- Lisa ‘gpu_titanrtx’ partition:
  - 4 x TitanRTX, 24 GB GPU memory
  - 16.31 TFLOPS @ FP32, 130 TFLOPS ‘mixed precision’ (A, B @ FP16, C & D @ FP32)

https://servicedesk.surf.nl/wiki/display/WIKI/Snellius+hardware+and+file+systems
https://servicedesk.surf.nl/wiki/display/WIKI/Lisa+hardware+and+file+systems

Parallelization: why?

5 years @ 1 A100 GPU



Parallelization: why?

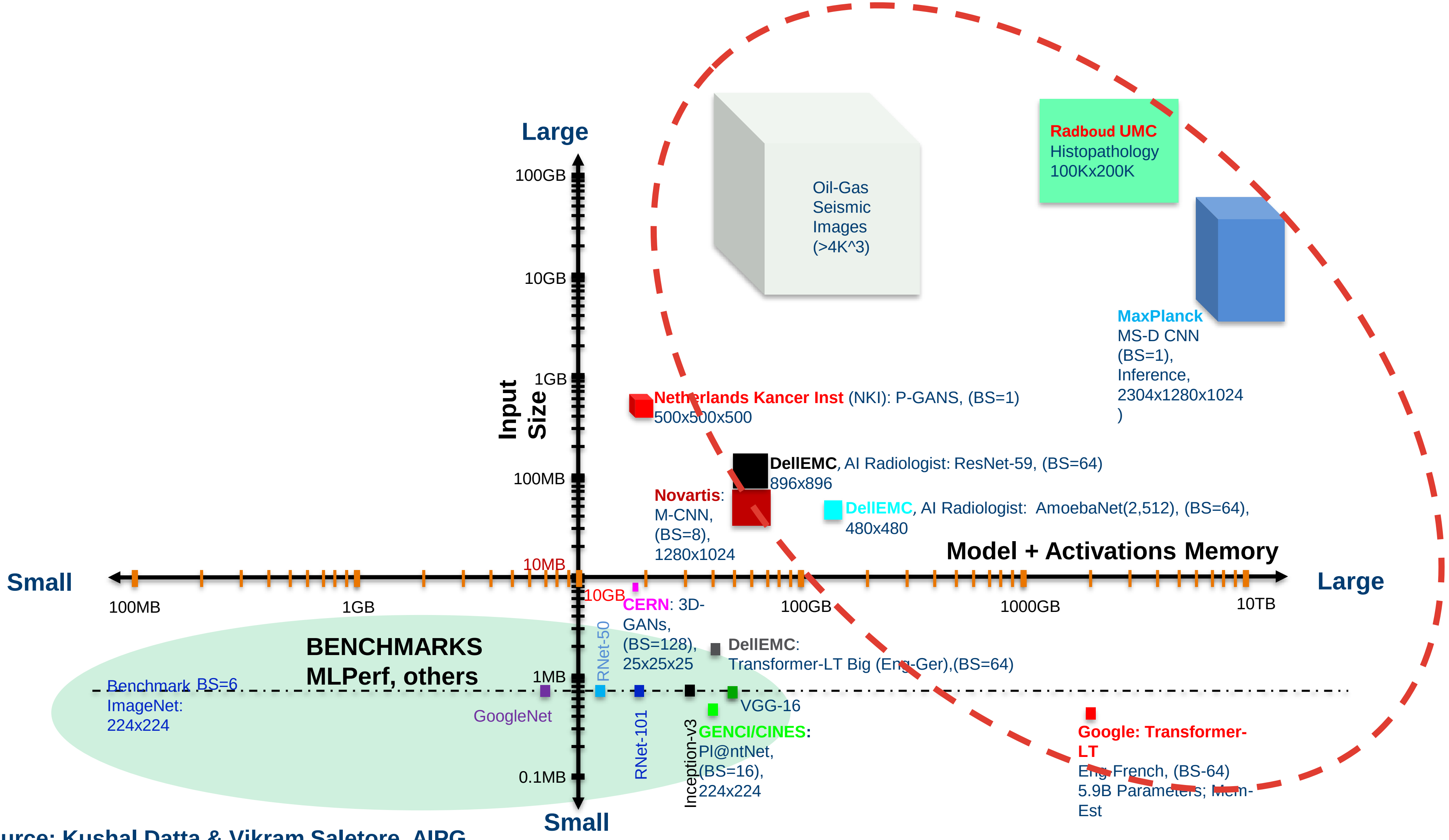
Faster trainings …

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

Bigger models (high memory requirement) …

- Enables larger, more complex models

Parallelization: when?



Source: Kushal Datta & Vikram Saletore, AIPG, Intel

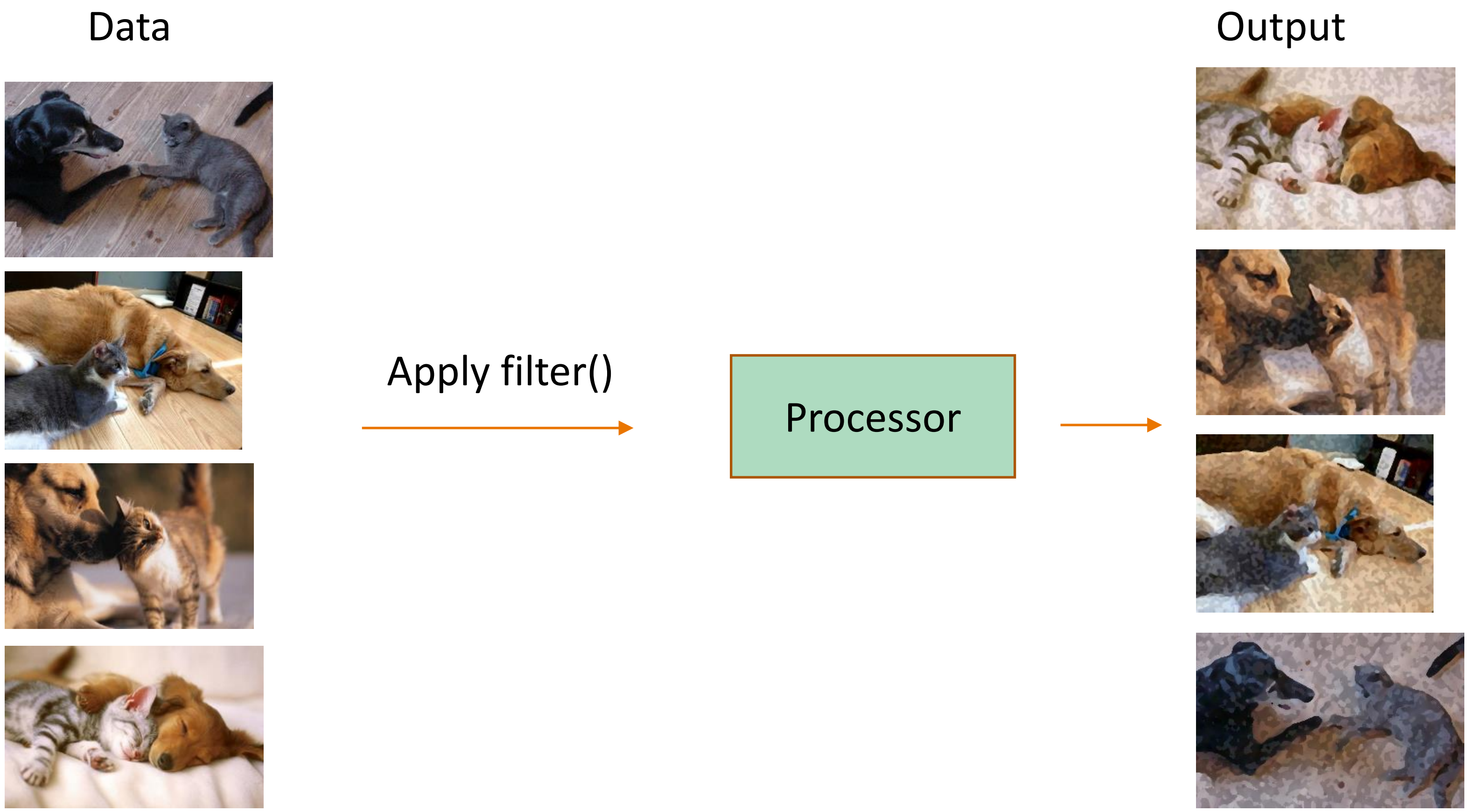
Parallelization: the basics

What is parallel computing?

- Multiple processors or computers working on a single computational problem

Parallelization: the basics

Serial computing



Data

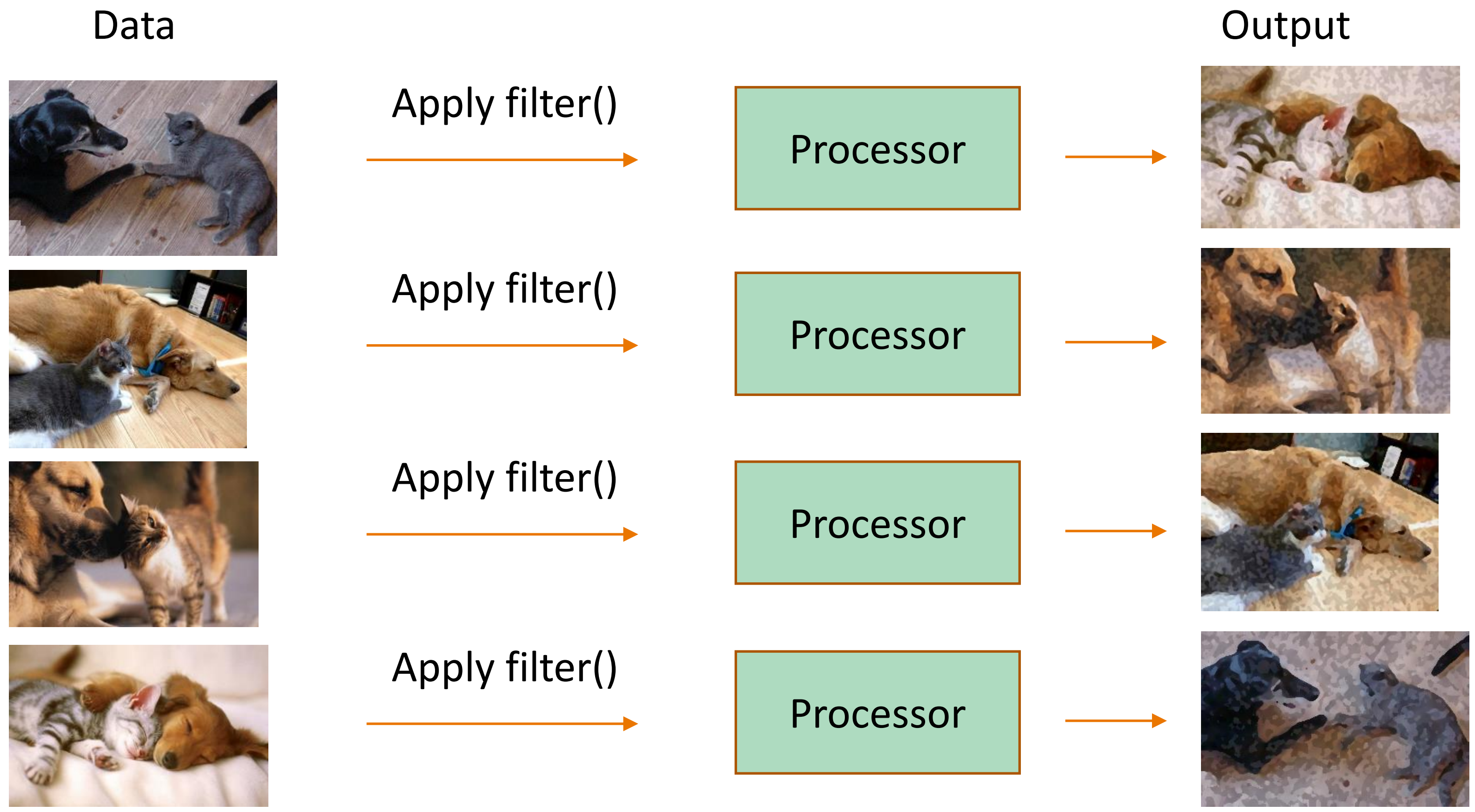
Apply filter()

Processor

Output

Parallelization: the basics

Parallel computing



Data

Output

Apply filter()

Processor

Apply filter()

Processor

Apply filter()

Processor

Apply filter()

Processor

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

Parallelization for deep learning

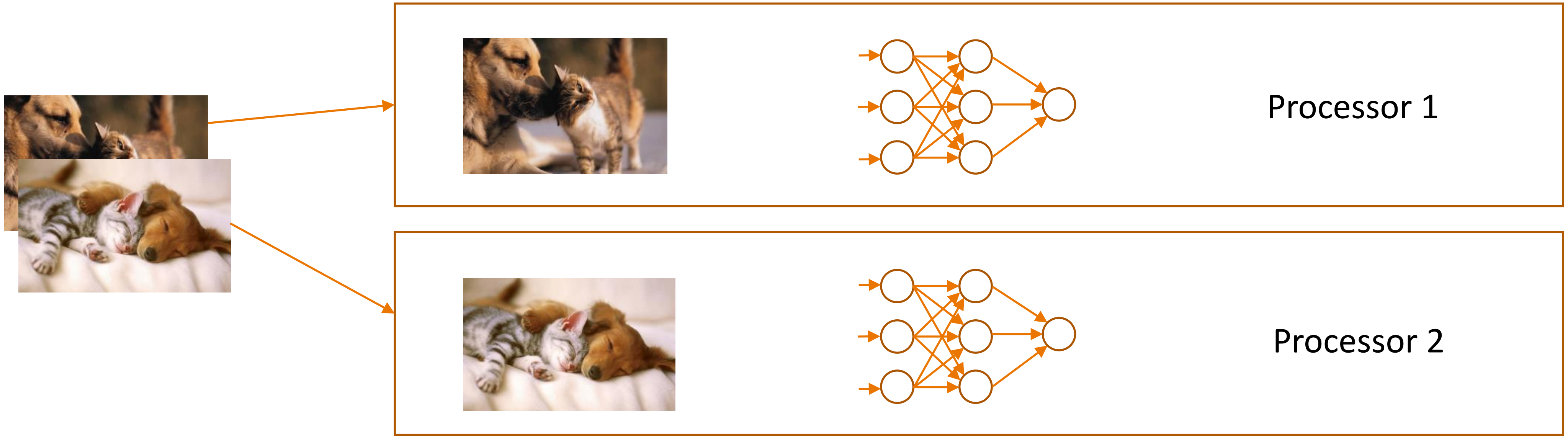
How does one 'divide' DL tasks?

- Data parallelism
- Model parallelism
- Hybrid data/model parallelism
- Pipeline parallelism

Increasing complexity

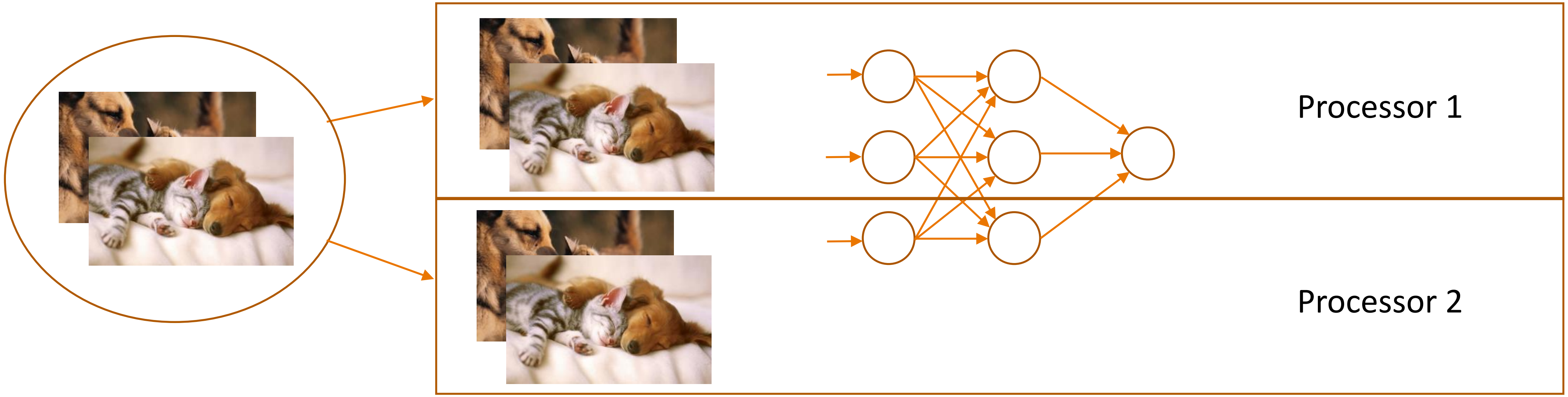
Data Parallelism

- Split the batch over multiple processors (CPUs/GPUs)
- Each processor holds a copy of the model
- Forward pass: calculated by each of the workers
- Backward pass: gradients computed (per worker), communicated and aggregated

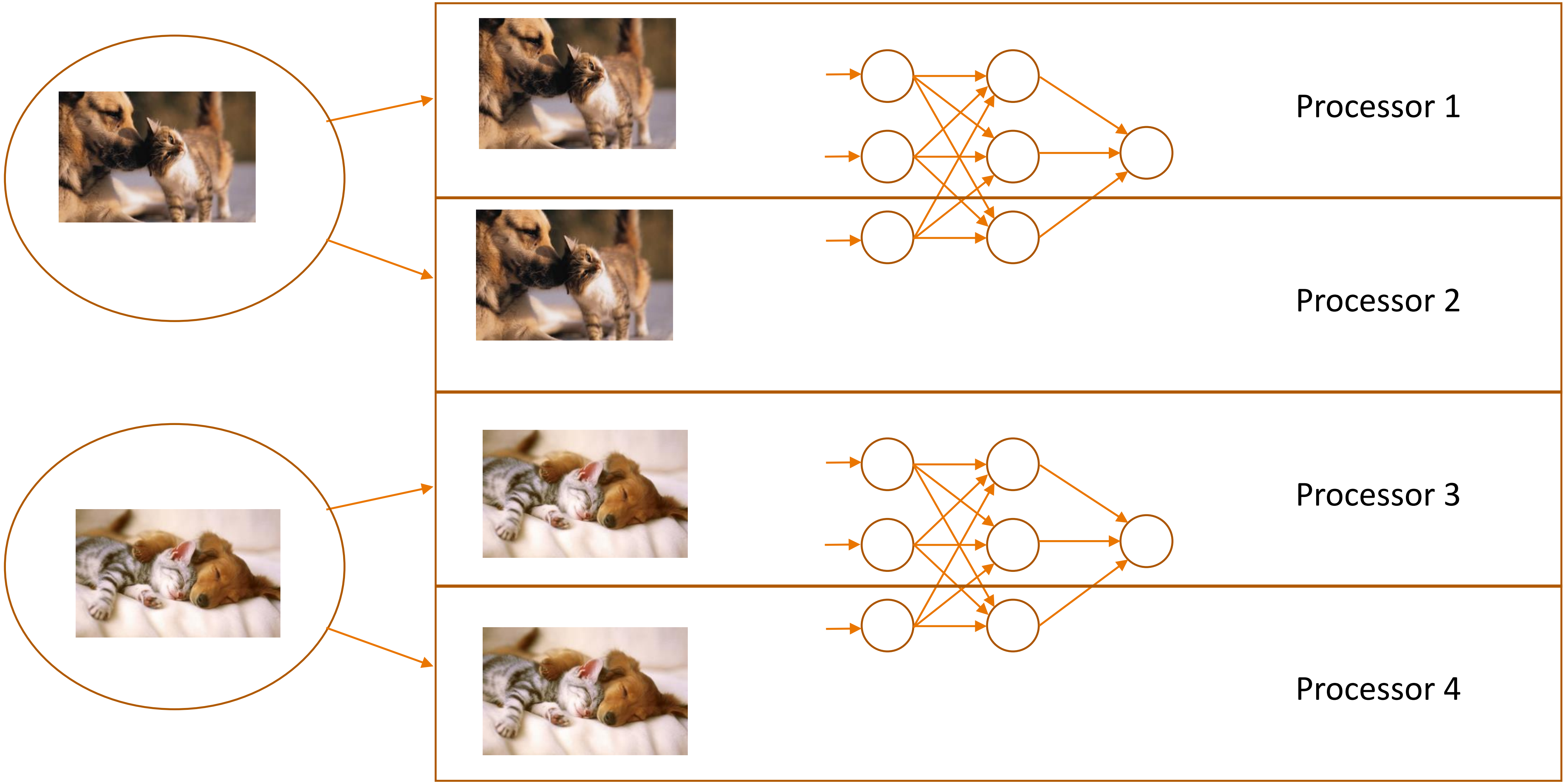


Model parallelism

- Split the model over multiple processors (CPUs/GPUs)
- Each processor sees all the data
- Communication needed both during forward and backward pass!



Hybrid model/data parallelism



Parallelization for deep learning

Very generic statements (there are exceptions) on model vs data parallelism

	Data parallel	Model parallel
Throughput	Increases	Decreases
Implementation	Relatively straigtforward	More difficult
Communication	Low to moderate	High
Typical use case	Speedup training	Models with large memory requirement

Use data parallel whenever you can. Use model parallel if you *really* need to. Even then, consider alternatives:

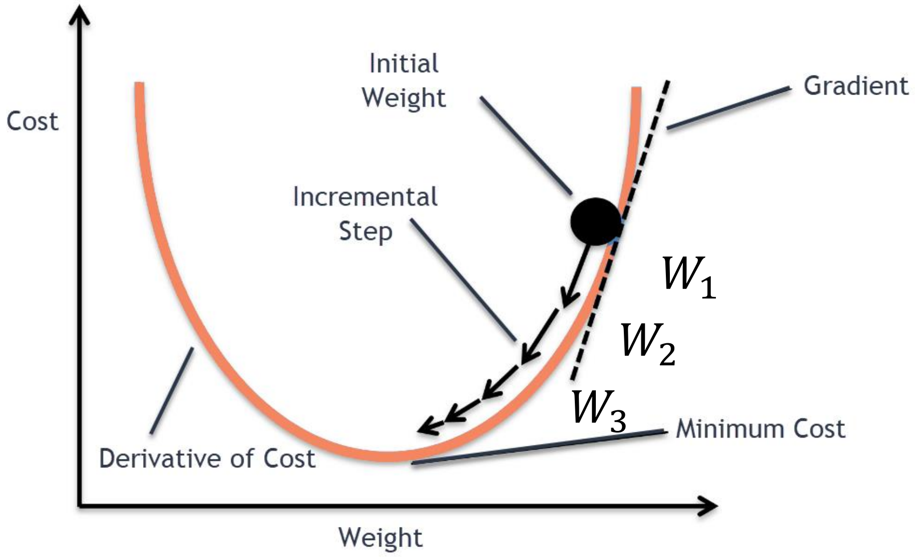
- Model pruning
- Use different hardware architecture (e.g. data parallel @ CPU)
- Use pipeline parallelism

Stochastic gradient descent (SGD)

SGD: find optimum by following the slope

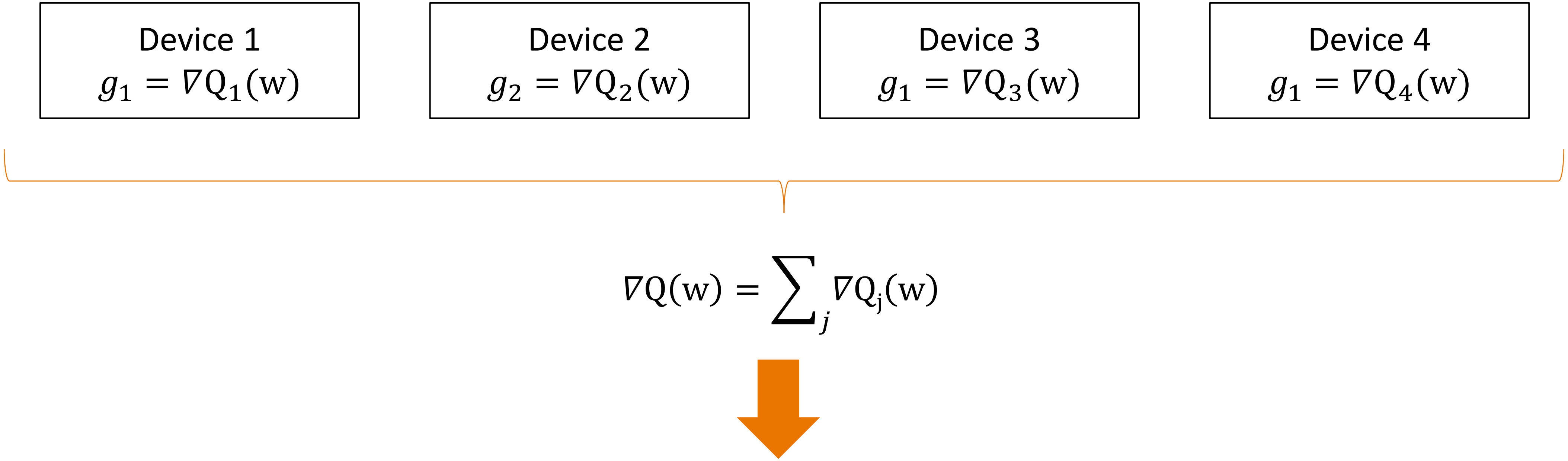
$$w = w - \eta \nabla \mathrm{Q}(\mathrm{w})$$

w = weights, η = learning rate, $\nabla \mathrm{Q}(\mathrm{w})$ = gradient for current batch.



Data parallel synchronous SGD

- Each device (j) computes the gradients ($\nabla \mathrm{Q_j(w)}$) based on its own batch!
- Needs to be aggregated before updating weights



Device 1
$g_1 = \nabla \mathrm{Q_1(w)}$

Device 2
$g_2 = \nabla \mathrm{Q_2(w)}$

Device 3
$g_1 = \nabla \mathrm{Q_3(w)}$

Device 4
$g_1 = \nabla \mathrm{Q_4(w)}$

$$\nabla \mathrm{Q(w)} = \sum_j \nabla \mathrm{Q_j(w)}$$

$$w = w - \eta \nabla \mathrm{Q(w)}$$

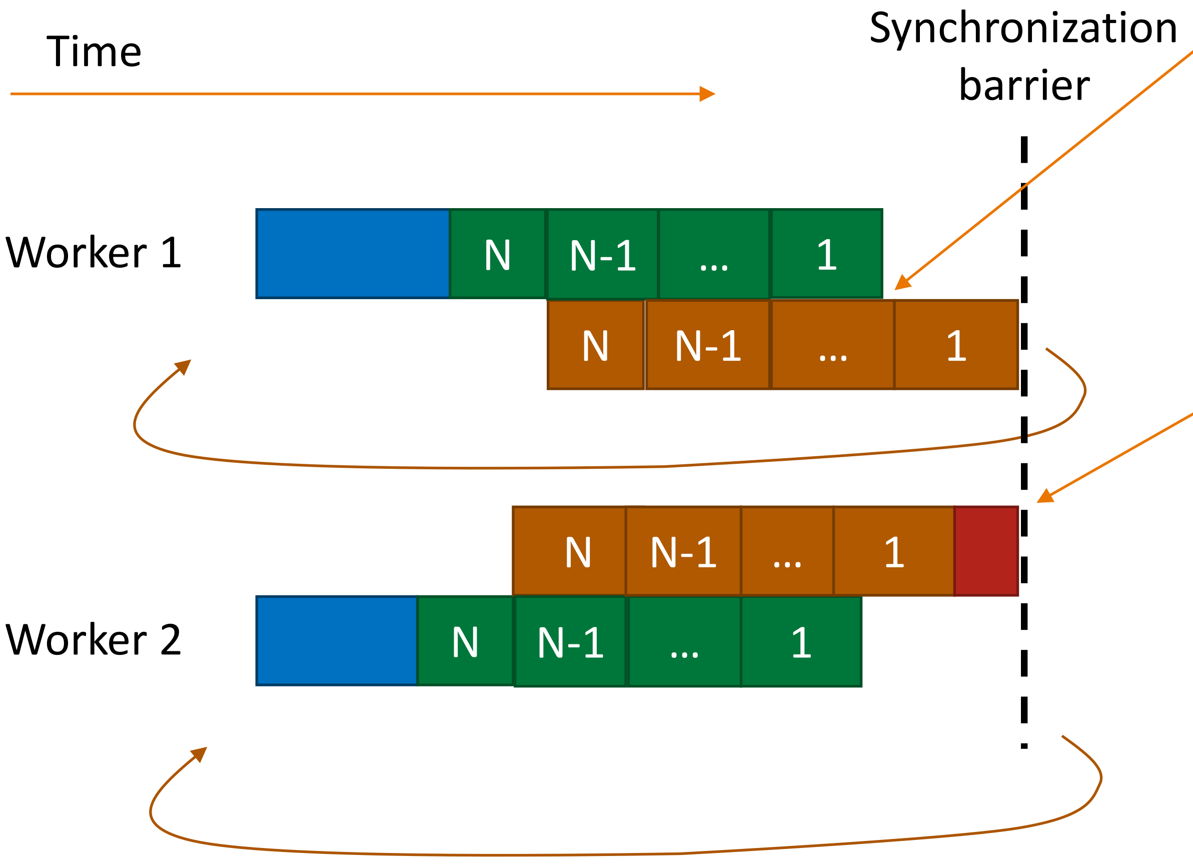
Data parallel synchronous SGD

Effect on batch size:

- For *N* workers that each see *n* examples: batch size effectively n × N.
- Larger batch => generally needs to be compensated by higher learning rate.
- No exact science!
    - Some use $\eta_{distributed} = \eta_{serial} \cdot N$
    - Some use $\eta_{distributed} = \eta_{serial} \cdot \sqrt{N}$
    - Experiment!

Data parallel synchronous SGD

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 ☺

Pro tip 2:

- Penalty for *synchronous* SGD: you have to wait for the slowest worker, before next iteration.
- Make sure all workers are equally fast!

🟦 Forward pass

🟩 N Backward pass,
compute gradient of layer N

🟧 N Backward pass,
communicate gradient N

🟥 Wait

Frameworks for distributed learning

- TensorFlow’s tf.distribute: quite tricky to program. Lot’s of code changes needed from serial to distributed (https://www.tensorflow.org/guide/distributed_training)
- TensorFlow + Horovod: serial => distributed with minimal code changes (https://horovod.readthedocs.io/en/stable/tensorflow.html)
- PyTorch’s torch.distributed (https://pytorch.org/tutorials/intermediate/dist_tuto.html)
- PyTorch + Horovod: serial => distributed with minimal code changes (https://horovod.readthedocs.io/en/stable/pytorch.html)
- PyTorch Lightning: hides a lot of boiler plate code (also nice for serial training). Very little changes needed between serial & parallel execution, especially on a SLURM cluster (https://pytorch-lightning.readthedocs.io/en/latest/clouds/cluster.html#slurm-managed-cluster)

Further reading

- Distributed TensorFlow using Horovod: https://towardsdatascience.com/distributed-tensorflow-using-horovod-6d572f8790c4
- Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis: https://arxiv.org/pdf/1802.09941.pdf
- Prace best practice guide for Deep Learning: http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf
- Technologies behind Distributed Deep Learning: https://preferredresearch.jp/2018/07/10/technologies-behind-distributed-deep-learning-allreduce/
- PyTorch Distributed: https://pytorch.org/tutorials/beginner/dist_overview.html and https://pytorch.org/docs/stable/distributed.html