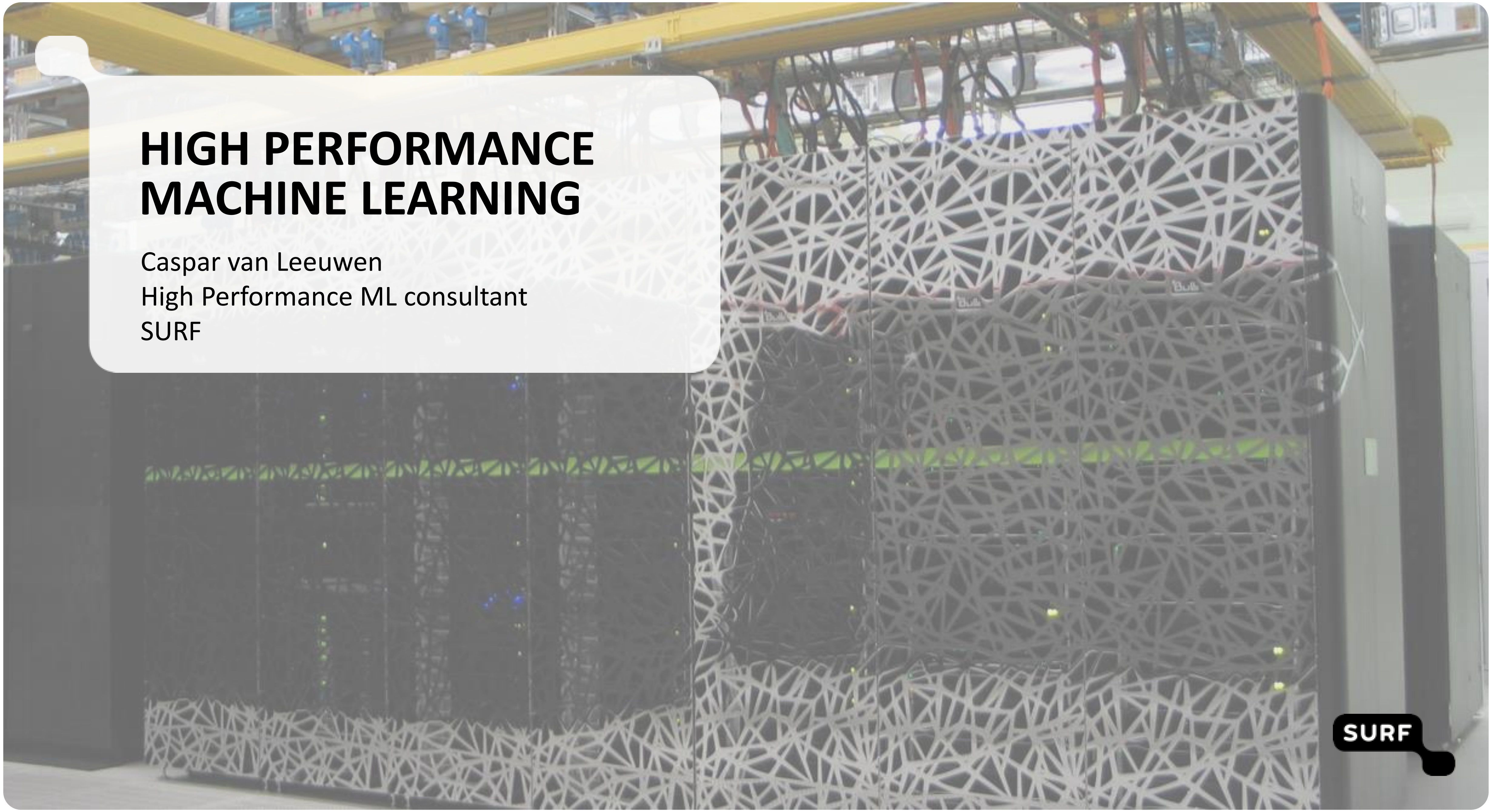


HIGH PERFORMANCE MACHINE LEARNING

Caspar van Leeuwen
High Performance ML consultant
SURF

Hardware

Goals:

- Understand what hardware bottlenecks could be limiting
- Understand pro’s and con’s of various hardware
- Know how to choose appropriate hardware for you DL task
- Know what to do to mitigate bottlenecks

Hardware bottlenecks

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

Hardware bottlenecks

- Compute (floating point operations per second, FLOPS) ← E.g. training a compute intensive network on a single node
- Memory bandwidth
- Memory size
- I/O
- Communication

Hardware bottlenecks

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

> - Data needs to get to the processor in time in order to do compute!
> - Many codes are limited by memory bandwidth

Hardware bottlenecks

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

- Very deep or wide networks, or networks with very large input/output layers (e.g. high resolution images) may be limited by memory size.
- Not a performance bottleneck, but a no-go!

Hardware bottlenecks

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

- HPC systems typically have shared file systems, usually with good bandwidth, but (relatively) low IOPS
- (Very) common bottleneck in distributed learning! Many nodes reading from the same filesystem.
- Other users (& sysadmins) will dislike you if you do I/O in a naive way!

Hardware bottlenecks

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

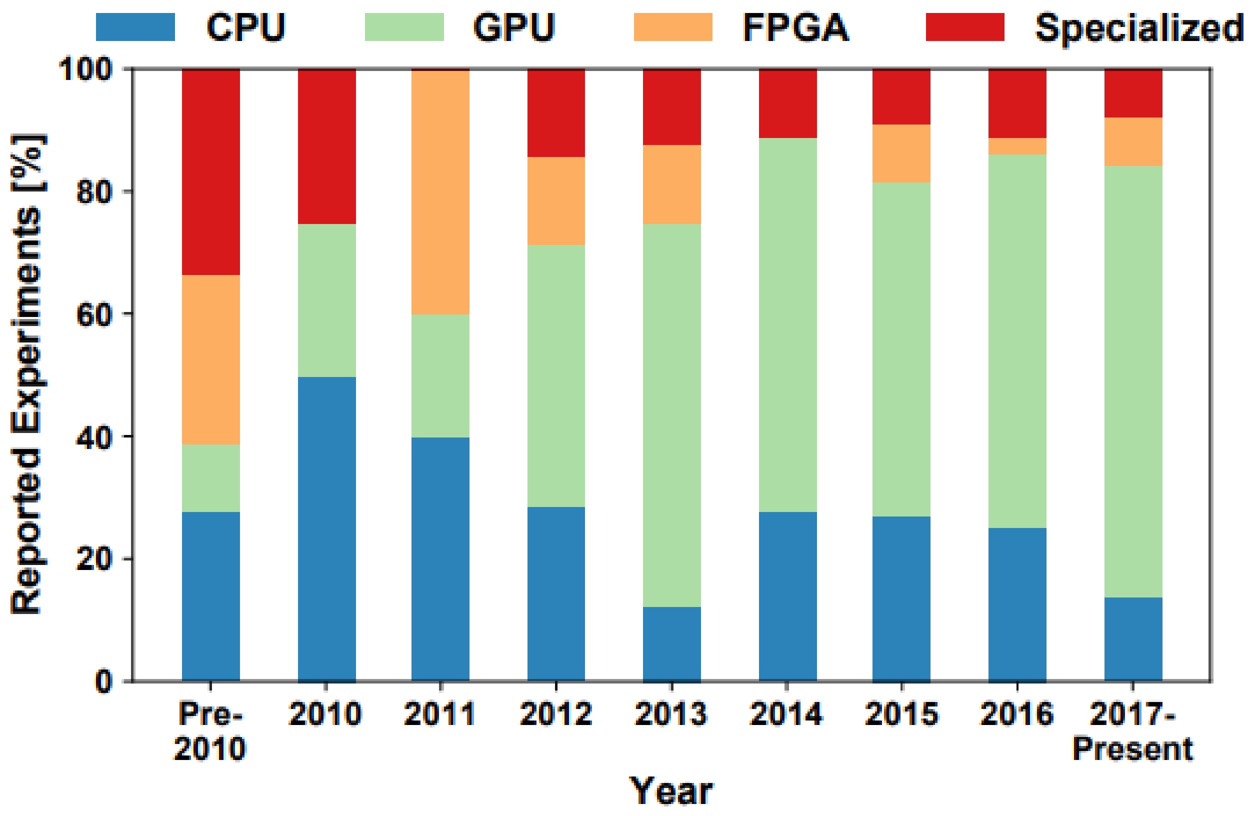
Communication can be limiting in several ways:

- Latency (many, small message send between nodes)
- Bandwidth (few, large messages send between nodes)
- Load imbalance (some workers in distributed job are slower / have more work; others have to wait when synchronization is needed)
- CPU ⇔ GPU

Hardware overview

A look at the hardware, from a DL perspective:

- Nvidia Pascal / Volta GPUs
- AMD Vega / Vega20
- Intel Xeon Scalable
- AMD Zen
- Specialized hardware
- I/O
- Interconnects



Source: Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis, Ben-Nun & Hoefler 2018

Hardware overview

	INT8 [TOPS]	FP16 [TFLOPS]	Bloat16	FP32 [TFLOPS]	FP64 [TFLOPS]	Memory [GB]	Memory Bandwidth [GB/s]	PCIe [GB/s]	Proprietary Interconnect [GB/s]
AMD MI60	58.9	29.5		14.7	7.4	32	1024	31.51	200 (2 × 100)
AMD MI100	184.6	184.6	92.3	23.1 / 46.1	11.5	32	1200	31.51	300 (3×100)
AMD MI250	362.1	362.1	362.1	45.3 / 90.5	45.3 / 90.5	128	3276.8	31.51	800 (8 × 100)
NVIDIA V100	62.8	31.4 / 125		15.7	7.8	16/32	900	15.75	300 (6 × 50)
NVIDIA A100	624	78 / 312	312	19.5 / 156	9.7 / 19.5	40/80	1555/2039	64	600
Intel Xeon Scalable 8180 (per socket)	-	-		3.0 / 4.2	1.5 / 2.1	768 (max)	119 (max)	15.75	-
AMD EPYC 7601 (per socket)	-	-		1.1 / 1.4	0.56 / 0.69	2000 (max)	159 (max)	15.75	-

Source: Prace best practice guide – deep learning
http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf
https://en.wikipedia.org/wiki/Ampere_(microarchitecture)

Hardware overview

GPUs support reduced precision, has higher performance

	INT8 [TOPS]	FP16 [TFLOPS]	Bloat16	FP32 [TFLOPS]	FP64 [TFLOPS]	Memory [GB]	Memory Bandwidth [GB/s]	PCIe [GB/s]	Proprietary Interconnect [GB/s]
AMD MI60	58.9	29.5		14.7	7.4	32	1024	31.51	200 (2 × 100)
AMD MI100	184.6	184.6	92.3	23.1 / 46.1	11.5	32	1200	31.51	300 (3×100)
AMD MI250	362.1	362.1	362.1	45.3 / 90.5	45.3 / 90.5	128	3276.8	31.51	800 (8 × 100)
NVIDIA V100	62.8	31.4 / 125		15.7	7.8	16/32	900	15.75	300 (6 × 50)
NVIDIA A100	624	78 / 312	312	19.5 / 156	9.7 / 19.5	40/80	1555/2039	64	600
Intel Xeon Scalable 8180 (per socket)	-	-		3.0 / 4.2	1.5 / 2.1	768 (max)	119 (max)	15.75	-
AMD EPYC 7601 (per socket)	-	-		1.1 / 1.4	0.56 / 0.69	2000 (max)	159 (max)	15.75	-

Source: Prace best practice guide – deep learning
http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf
https://en.wikipedia.org/wiki/Ampere_(microarchitecture)

Hardware overview

GPUs have a lot of FLOPS compared to CPUs

	INT8 [TOPS]	FP16 [TFLOPS]	Bloat16	FP32 [TFLOPS]	FP64 [TFLOPS]	Memory [GB]	Memory Bandwidth [GB/s]	PCIe [GB/s]	Proprietary Interconnect [GB/s]
AMD MI60	58.9	29.5		14.7	7.4	32	1024	31.51	200 (2 × 100)
AMD MI100	184.6	184.6	92.3	23.1 / 46.1	11.5	32	1200	31.51	300 (3×100)
AMD MI250	362.1	362.1	362.1	45.3 / 90.5	45.3 / 90.5	128	3276.8	31.51	800 (8 × 100)
NVIDIA V100	62.8	31.4 / 125		15.7	7.8	16/32	900	15.75	300 (6 × 50)
NVIDIA A100	624	78 / 312	312	19.5 / 156	9.7 / 19.5	40/80	1555/2039	64	600
Intel Xeon Scalable 8180 (per socket)	-	-		3.0 / 4.2	1.5 / 2.1	768 (max)	119 (max)	15.75	-
AMD EPYC 7601 (per socket)	-	-		1.1 / 1.4	0.56 / 0.69	2000 (max)	159 (max)	15.75	-

Source: Prace best practice guide – deep learning
http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf
https://en.wikipedia.org/wiki/Ampere_(microarchitecture)

Hardware overview

CPUs have a lot of memory compared to GPUs

	INT8 [TOPS]	FP16 [TFLOPS]	Bloat16	FP32 [TFLOPS]	FP64 [TFLOPS]	Memory [GB]	Memory Bandwidth [GB/s]	PCIe [GB/s]	Proprietary Interconnect [GB/s]
AMD MI60	58.9	29.5		14.7	7.4	32	1024	31.51	200 (2 × 100)
AMD MI100	184.6	184.6	92.3	23.1 / 46.1	11.5	32	1200	31.51	300 (3×100)
AMD MI250	362.1	362.1	362.1	45.3 / 90.5	45.3 / 90.5	128	3276.8	31.51	800 (8 × 100)
NVIDIA V100	62.8	31.4 / 125		15.7	7.8	16/32	900	15.75	300 (6 × 50)
NVIDIA A100	624	78 / 312	312	19.5 / 156	9.7 / 19.5	40/80	1555/2039	64	600
Intel Xeon Scalable 8180 (per socket)	-	-		3.0 / 4.2	1.5 / 2.1	768 (max)	119 (max)	15.75	-
AMD EPYC 7601 (per socket)	-	-		1.1 / 1.4	0.56 / 0.69	2000 (max)	159 (max)	15.75	-

Source: Prace best practice guide – deep learning
http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf
https://en.wikipedia.org/wiki/Ampere_(microarchitecture)

Hardware overview

Memory bandwidth relative to FLOPS is approximately the same

	INT8 [TOPS]	FP16 [TFLOPS]	Bloat16	FP32 [TFLOPS]	FP64 [TFLOPS]	Memory [GB]	Memory Bandwidth [GB/s]	PCIe [GB/s]	Proprietary Interconnect [GB/s]
AMD MI60	58.9	29.5		14.7	7.4	32	1024	31.51	200 (2 × 100)
AMD MI100	184.6	184.6	92.3	23.1 / 46.1	11.5	32	1200	31.51	300 (3×100)
AMD MI250	362.1	362.1	362.1	45.3 / 90.5	45.3 / 90.5	128	3276.8	31.51	800 (8 × 100)
NVIDIA V100	62.8	31.4 / 125		15.7	7.8	16/32	900	15.75	300 (6 × 50)
NVIDIA A100	624	78 / 312	312	19.5 / 156	9.7 / 19.5	40/80	1555/2039	64	600
Intel Xeon Scalable 8180 (per socket)	-	-		3.0 / 4.2	1.5 / 2.1	768 (max)	119 (max)	15.75	-
AMD EPYC 7601 (per socket)	-	-		1.1 / 1.4	0.56 / 0.69	2000 (max)	159 (max)	15.75	-

Source: Prace best practice guide – deep learning
http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf
https://en.wikipedia.org/wiki/Ampere_(microarchitecture)

Nvidia Volta (e.g. V100, TitanRTX)

Features

- INT8 & FP16 support

Generally *you* are responsible for specifying a reduced precision: DL frameworks don’t do this automatically since it may impact your networks accuracy, convergence, etc

- (Volta) Tensor cores: fused multiply-add units that support mixed precission (multiply in FP16, add in FP32). High performance: 120 TOPS.

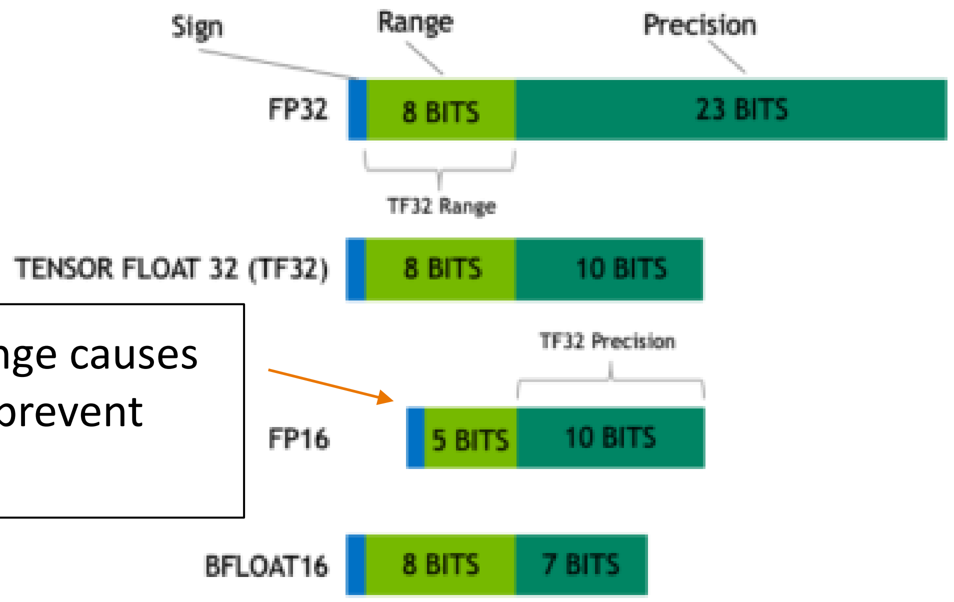
$$\underset{\text{FP16 or FP32}}{D} = \underset{\text{FP16}}{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}} \underset{\text{FP16}}{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}} + \underset{\text{FP16 or FP32}}{\begin{matrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{matrix}}$$

- High bandwidth memory, 2nd generation (HBM2)
- NVLink: allows direct $GPU_1 \Leftrightarrow GPU_2$ communication in a multi-GPU node (much faster than having to go over PCIe)

Nvidia Ampere GPUs (e.g. A100)

Features (see also https://servicedesk.surf.nl/wiki/display/WIKI/Deep+Learning+on+A100+GPUs)

- Tensor Cores support more datatypes (FP64, TF32, FP16, BF16, Int8, Int4, Binary)
- TensorFloat32 datatype:

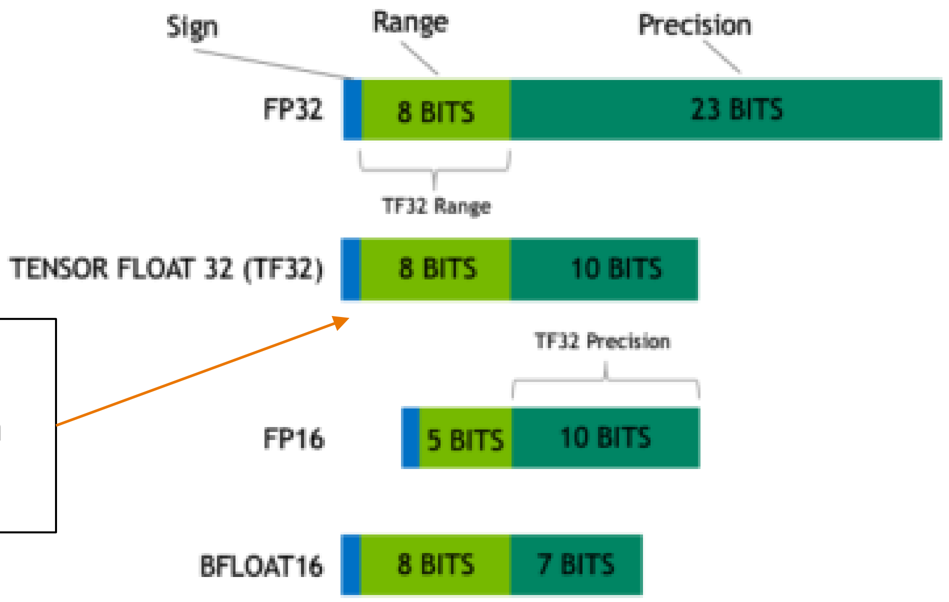


Mixed precision uses FP16. Reduced range causes (potential) need for gradient scaling to prevent underflows

Nvidia Ampere GPUs (e.g. A100)

Features (see also https://servicedesk.surf.nl/wiki/display/WIKI/Deep+Learning+on+A100+GPUs)

- Tensor Cores support more datatypes (FP64, TF32, FP16, BF16, Int8, Int4, Binary)
- TensorFloat32 datatype:

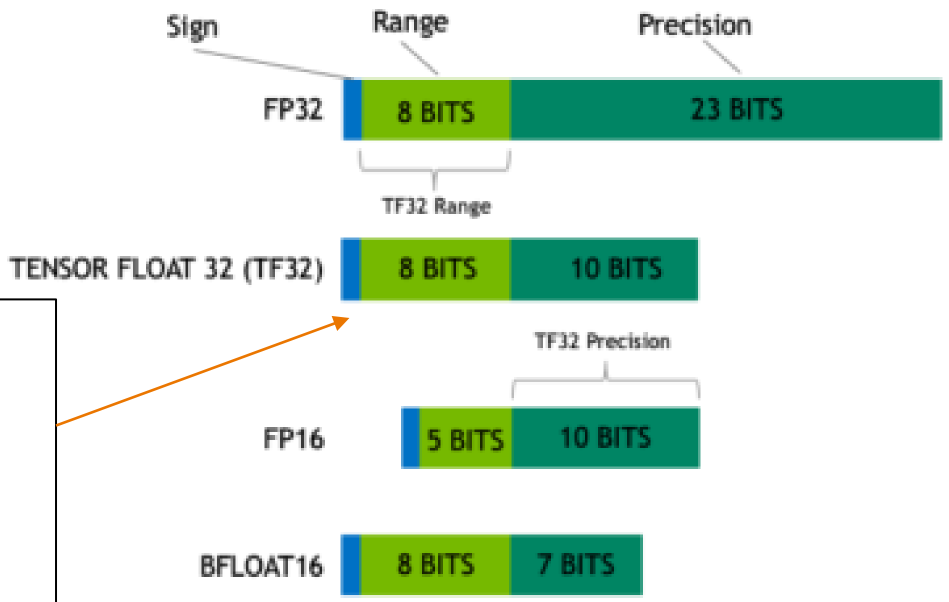


TF32 tries to combine best of both worlds: range of an FP32 (no gradient scaling needed), but with reduced precision (for speedup & less memory)

Nvidia Ampere GPUs (e.g. A100)

Features (see also https://servicedesk.surf.nl/wiki/display/WIKI/Deep+Learning+on+A100+GPUs)

- Tensor Cores support more datatypes (FP64, TF32, FP16, BF16, Int8, Int4, Binary)
- TensorFloat32 datatype:



Automatically used *unless*
- Environment variable *NVIDIA_TF32_OVERRIDE=0* is set
- *torch.backends.cuda.matmul.allow_tf32 = False* and torch.*backends.cudnn.allow_tf32 = False* are set

Nvidia Ampere GPUs (e.g. A100)

Model	Precision	Throughput (img/s)	Speedup (compared to FP32)	Loss (1st iteration)
ResNet50	FP32	455.9	1	7.457645893096924
ResNet50	TF32	750.6	1.65	7.456507205963135
ResNet50	FP16 (input) + FP32 (accumulator)	1087.6	2.38	7.45703125
VGG19	FP32	212.7	1	6.9077839851379395
VGG19	TF32	550.3	2.59	6.907783508300781
VGG19	FP16 (input) + FP32 (accumulator)	1099.8	5.17	6.90625
DenseNet121	FP32	391.9	1	6.96142053604126
DenseNet121	TF32	591.5	1.51	6.961450576782227
DenseNet121	FP16 (input) + FP32 (accumulator)	876.2	2.24	6.9609375

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

Nvidia Volta / Ampere GPUs

(Low-level) library support

- CUDA
- cuBLAS: basic linear algebra
- cuSPARSE: sparse matrix algebra
- cuDNN: primitives for deep neural networks
- NCCL: NVIDIA collective communications library (implements efficient allreduce and supports e.g. NVLink & RDMA)

Efficient low-level libraries are just as important as hardware itself! Otherwise, theoretical hardware specs might be great, but you'll never get that performance!

AMD Aldebaran

Features

- INT8 & FP16 support
- High bandwidth memory, 2nd generation (HBM2)
- PCIe 4.0 support (large bandwidth CPU ⇔ GPU)
- Infinity fabric: proprietary interconnect for high bandwidth GPU ⇔ GPU communication in a multi-GPU node.

AMD Aldebaran

Features

- INT8 & FP16 support
- High bandwidth memory, 2nd generation (HBM2)
- PCIe 4.0 support (large bandwidth CPU ⇔ GPU)
- Infinity fabric: proprietary interconnect for high bandwidth GPU ⇔ GPU communication in a multi-GPU node.

AMD AMD

(Low-level) library support

- Heterogeneous-computing Interface for Portability (HIP): a C++ dialect that was designed to ease conversion of CUDA applications to portable C++ code
- MIOpen: machine learning primitives (based on the OpenCL or HIP)
- rocBLAS: basic linear algebra
- ROCm: software ecosystem for GPU computing
- ROCm: has forks of TensorFlow, Caffe 2, PyTorch, MxNet and CNTK with MIOpen support.

While software stack is ‘young’ compared to NVidia, performance is competitive – provided the ROCm forks of the frameworks are used when computing on AMD hardware!

Intel Xeon Scalable

Pro tip:
Low-end Xeon Scalable processors have only 1 AVX-512 FMA unit. AVX2 may perform better on these processors, because AVX-512 instructions are executed at lower clock speeds!

- Supports AVX-512 vector instructions
- Supports very large memory (more than 1 TB per node)
- Intel Math Kernel Library (MKL): optimized BLAS, LAPACK and FFTW routines
- Intel MKL-DNN: primitives for deep learning.

AMD Zen

- Supports AVX-2 vector instructions
- Supports very large memory (more than 1 TB per node)
- BLIS: a BLAS implementation optimized for AMD EPYC processors
- libFLAME: portable library for dense matrix multiplications

Note: the Zen processor is not really marketed as a processor for DL task. DL frameworks don’t generally support its low level libraries.

More info: PRACE AMD EPYC best practice guide, http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-AMD.pdf

Specialized hardware

Tensor processing units (TPU)

- Chip designed specifically for machine learning (tensor operations)
- Available only in Google Cloud
- Google Cloud TPU v3: 420 TFLOPS, 128 GB HBM. About 8 USD/TPU-hour
- Supported in TensorFlow

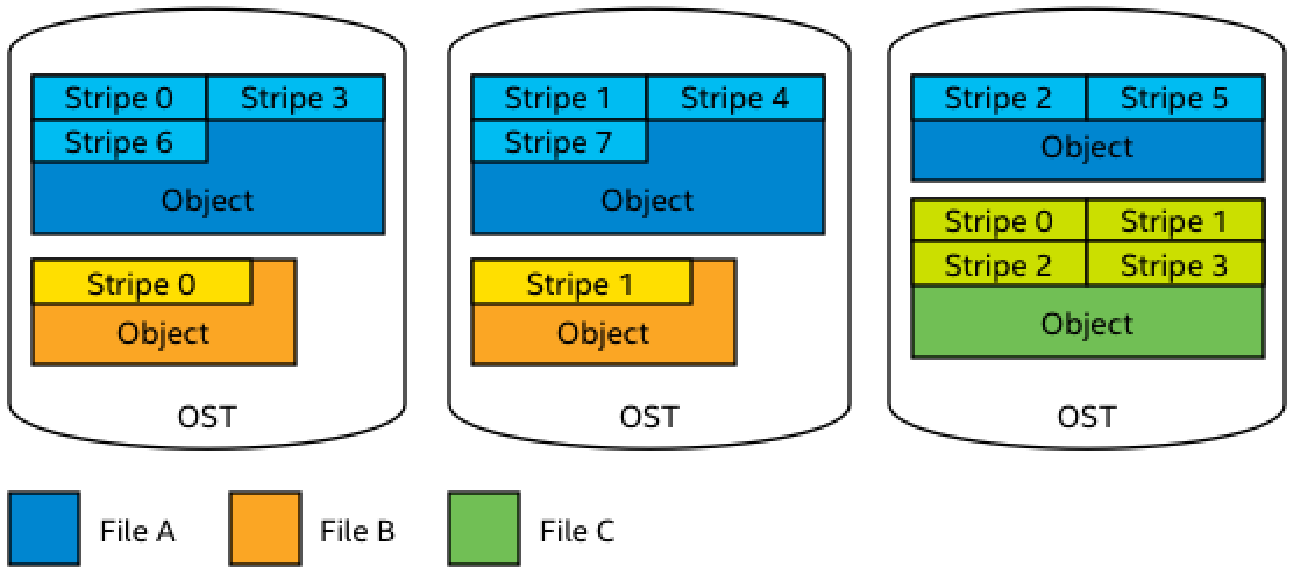
Field Programmable Gate Arrays (FPGA)

- Programmable chips
- Can do e.g. compute in any desired accuracy
- Very experimental, no framework support
- Maybe in the future…

I/O

HPC systems typically use parallel shared file systems.

- Parallel file system: one file can be distributed over many physical disks, to increase I/O bandwidth.



I/O

Two main types of parallel filesystems

- Lustre
  - Metadata (filename, size, location, etc) stored on separate server
  - Object Storage Target (OST) stores actual file
  - Striping over multiple OSTs can be managed by user
- GPFS
  - Metadata and actual file stored on the same server
  - Striping is managed automatically, by file system. User has no control.

Mitigate hardware bottlenecks

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

- Use high-FLOP hardware (GPUs, TPUs)
- Use specialized vector instructions: AVX, AVX2, AVX-512 (CPUs)
- Use reduced precision training (GPUs)
- Use distributed learning

Pro tip for Tensorflow CPU training
'Pip install tensorflow' installs a TensorFlow binary with only AVX support. Ops that are accelerated with MKL-DNN are ok (use newer AVX instructions), but poor performance for ops that are *not* accelerated by MKL-DNN.

- Build from source yourself (tricky, but allows perfect optimization for your system)
- Use prebuilt Intel optimized Tensorflow on Intel CPUs: pip install intel-tensorflow, or conda. (easy, but TF versions may lag behind) See https://software.intel.com/en-us/articles/intel-optimization-for-tensorflow-installation-guide

Mitigate hardware bottlenecks

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

- Low level libraries (cuDNN, MKL-DNN) have optimized memory access patterns, meaning you get the most out of your memory bandwidth
- Make sure your DL framework is build against the appropriate low level libraries
- Use a tensors in channels-last memory format on Nvidia GPUs (https://pytorch.org/tutorials/intermediate/memory_format_tutorial.html)

Mitigate hardware bottlenecks

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

- Choose different architecture. E.g. TPUs, or CPUs (distributed CPU training can provide as many FLOPS as serial training on GPUs, but much more memory!)
- Prune your model, if you can
- Model/pipeline parallelism

Mitigate hardware bottlenecks

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

Depends on what's limiting:

- IOPS: pack small files into large files. Many frameworks provide dedicated file formats for this (TensorFlows: TFRecords, Caffe: LMDB), though other packed data formats such as HDF provide similar performance benefits. Particularly important if reading from Lustre filesystem: metadata servers don't scale!
- Bandwidth: exploit parallel filesystems. E.g. use Lustre striping. See PRACE Parallel I/O best practice guide http://www.prace-ri.eu/best-practice-guide-parallel-i-o/
- Stage on local disks, if nodes have it. This means read from shared filesystem only occurs once – further reads (each epoch) are done from local filesystem
- Stage in RAM (/dev/shm). Only an option for small datasets.

Mitigate hardware bottlenecks

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

Depends on limiting factor:

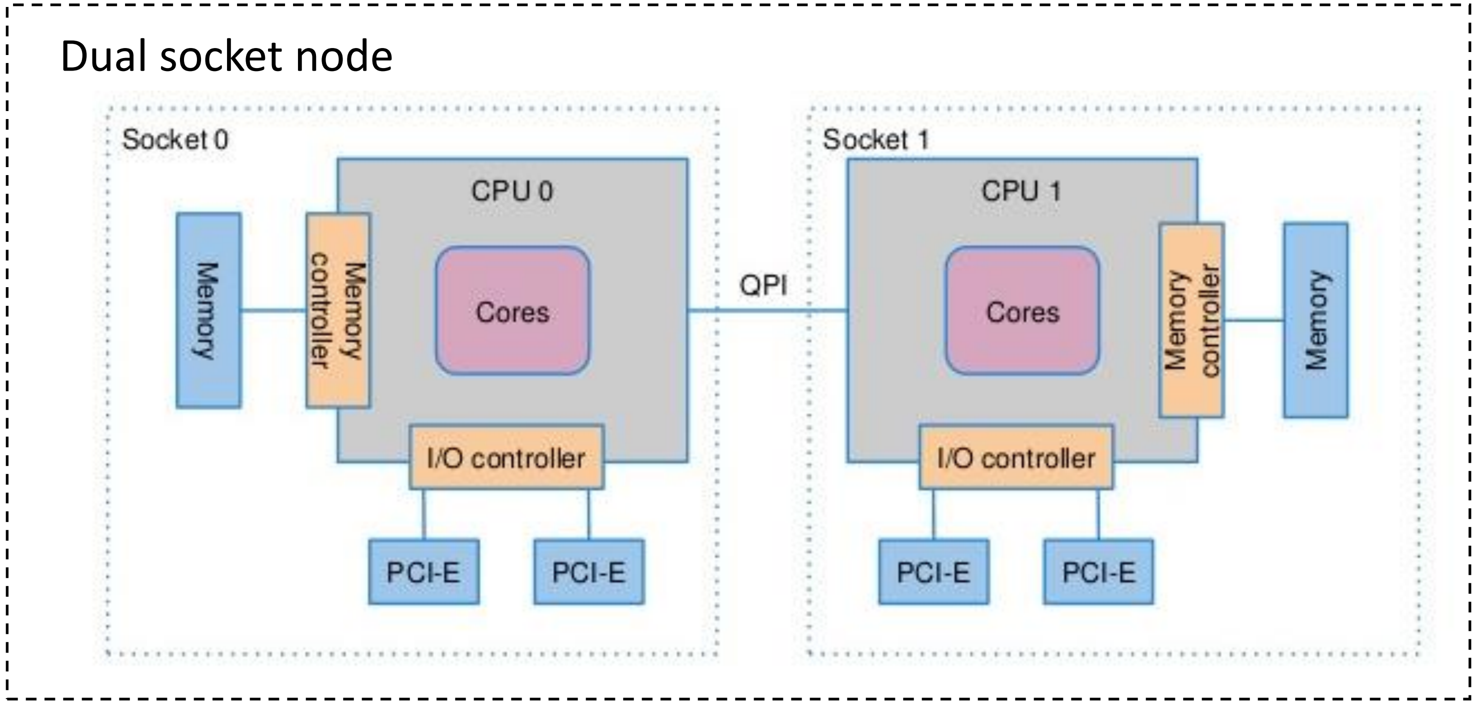
- Latency: bundle small messages into larger ones (e.g. tensor fusion)
- Bandwidth: send as little data as possible (efficient reduction operation) using as much of the network between nodes as possible (parameter server = single bottleneck)
- Load imbalance: use homogeneous node types, all with same hardware.
- SHARP (https://docs.nvidia.com/networking/display/SHARPv200) capable switches can do e.g. reduction operations, substantially reducing the amount of memory sent over the network for MPI_All* operations

More about tensor fusion:
https://github.com/horovod/horovod/blob/6f400014b8cb45aa013077aad0060032a4dda713/docs/tensor-fusion.rst

Launching parallel workloads on CPU

NUMA domains

- Memory from CPU 0 and CPU 1 is ‘one memory’ from programmers point of view
- Cores from CPU 0 can access memory from CPU 1, but is slower!
- Dual socket: 2 tasks (data parallel) per node ≈ 20-50% faster than 1 task per node



Launching parallel workloads on CPU

Problem:

- Performance hit if process moves from CPU 0 => CPU 1 (NUMA)
- Potential performance hit when multithreading across sockets (NUMA)
- Cache misses if thread is moved from one core to another (any multithreading application)

Solution:

- Launch 1 worker per socket
- Bind (MPI) processes to sockets
- Set number of threads to number of cores available per socket
- Bind threads to cores

Launching parallel workloads

Binds threads to (hyperthreading)cores

Set nr of threads to nr of cores per socket

Example, 2-socket node, 12 cores per socket

- KMP_AFFINITY="granularity=fine,compact,1,0" OMP_NUM_THREADS=12 mpirun -np 2 --map-by ppr:1:socket --bind-to socket python train.py

Maps one process to each socket

Binds to socket

Launches two processes

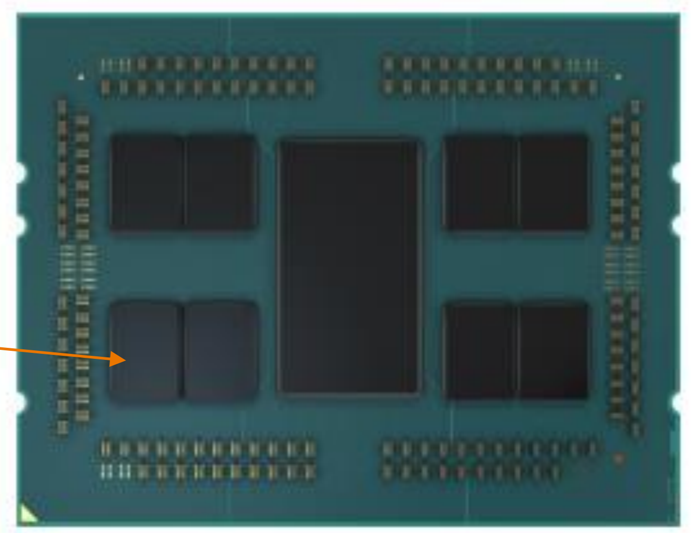
- Tip: specify '--report-bindings' to get verbose output from *mpirun* on how processes are mapped / bound.
- Tip: specify 'KMP_AFFINITY="granularity=fine,verbose,compact,1,0" to get verbose output on how threads are bound

Launching parallel workloads on AMD CPUs

AMD CPUs nowadays have many numa domains, because a single CPU consists of ‘chiplets’

- Use ‘lstopo’ command to see number of numa domains
- Launch 1 MPI process per numa domain

8 Chiplets on an AMD ZEN3 CPU



Launching parallel workloads on GPU

Depends on the code!

Code designed for multi-GPU node (e.g. uses tf.device('/gpu:0'), tf.device('/gpu:1') etc):

- Launch a single process per node

Code designed for single-GPU, but parallelized with e.g. Horovod:

- Typically: launch one (MPI) process per GPU

Recap

Goals:

- Understand what hardware bottlenecks could be limiting
- Understand pro’s and con’s of various hardware
- Know how to choose appropriate hardware for you DL task
- Know what to do to mitigate bottlenecks