Let's talk about running code efficiently on your local (HPC) system

Robert Jan Schlimbach

Credit where credit is due

Slides stolen from:

- CasparVL
- PyTorch
- Ezyang
- CUDA/cuDNN

...and a bit from myself

The three S'

- Know your System
  - What kind of hardware do I have?
  - What features does my hardware support?
- Know your Software
  - Does my software do what I want to do efficiently?
  - Does my software support my hardware features?
- Know your Stack
  - What version of software am I running?
  - Is my stack compiled for my hardware?

System

- https://servicedesk.surf.nl/wiki/display/WIKI/Snellius+hardware+and+file+systems
- Which CPUs?
  - X86_64? Aarch64?
- What kind of storage?
  - SSD (nvme?)? HDD? Tape? Local disk?
- What kind of Networking?
  - Ethernet? IB?
  - IOPs vs Throughput
- What kind of RAM?
  - HBM? DDR?
  - You (AI person) are always memory limited

System

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

System

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

System

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

E.g. training a compute intensive network on a single node

System

- Compute (floating point operations per second, FLOPS)
- Memory bandwidth
- 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

SURF

System

- 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!

System

- 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!

System

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

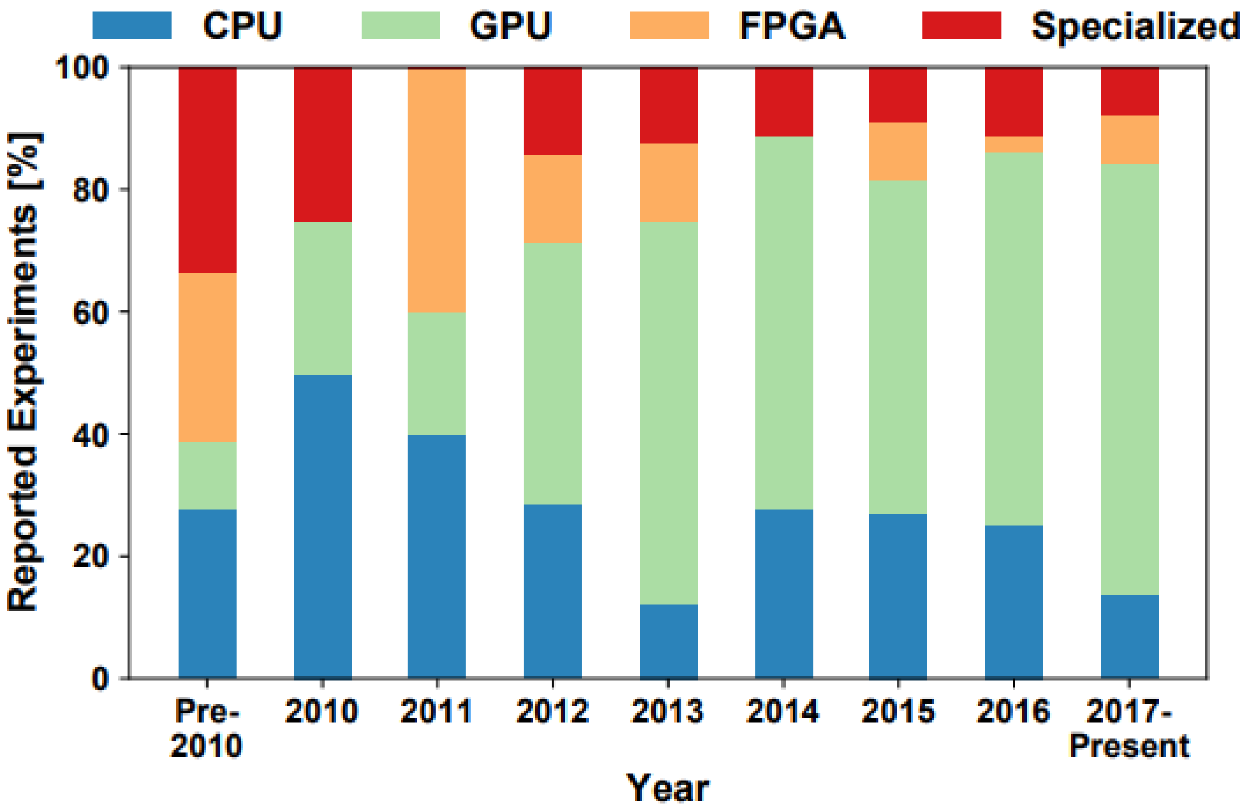
Communication can be limiting in several ways:

- 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

System

A look at the hardware, from a DL perspective:

- Nvidia is dominant, Volta was a game changer
- AMD is finally catching up
- Intel Xeon / SPR
- AMD Rome
- Specialized hardware
- I/O
- Interconnects



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

SURF

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)

SURF

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)

SURF

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.

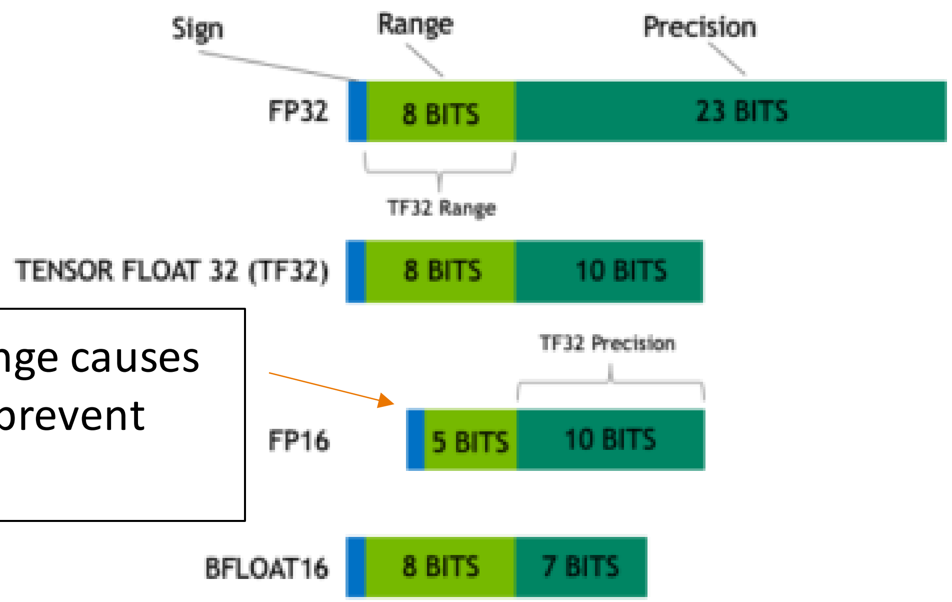
$$\underbrace{D}_{\text{FP16 or FP32}} = \underbrace{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}}_{\text{FP16}} \underbrace{\begin{pmatrix} B_{0,0} & B_{0,1}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

- High bandwidth memory, 2nd generation (HBM2) (vs. DDR)

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:

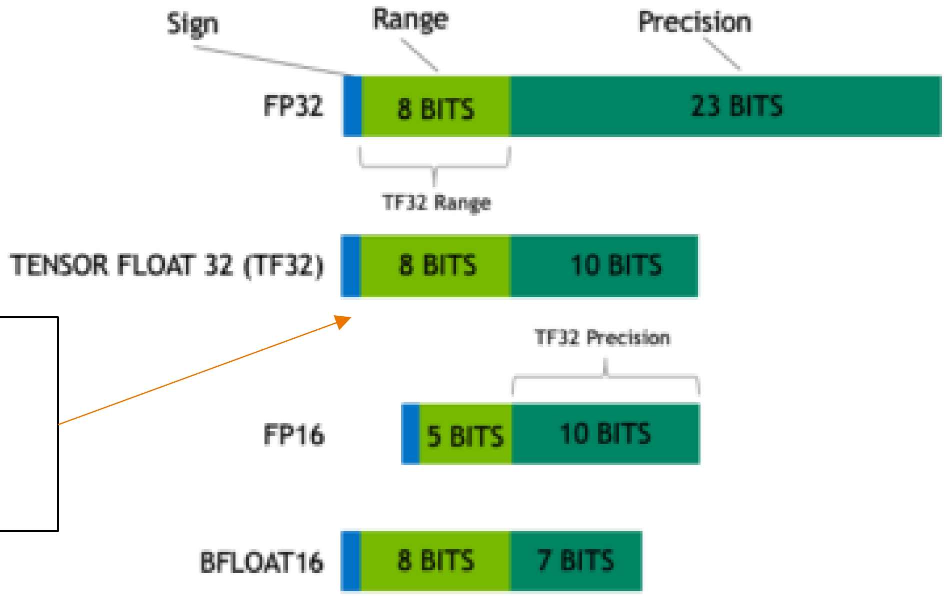


Mixed precision uses FP16. Reduced range causes (potential) need for gradient scaling to prevent underflows

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:

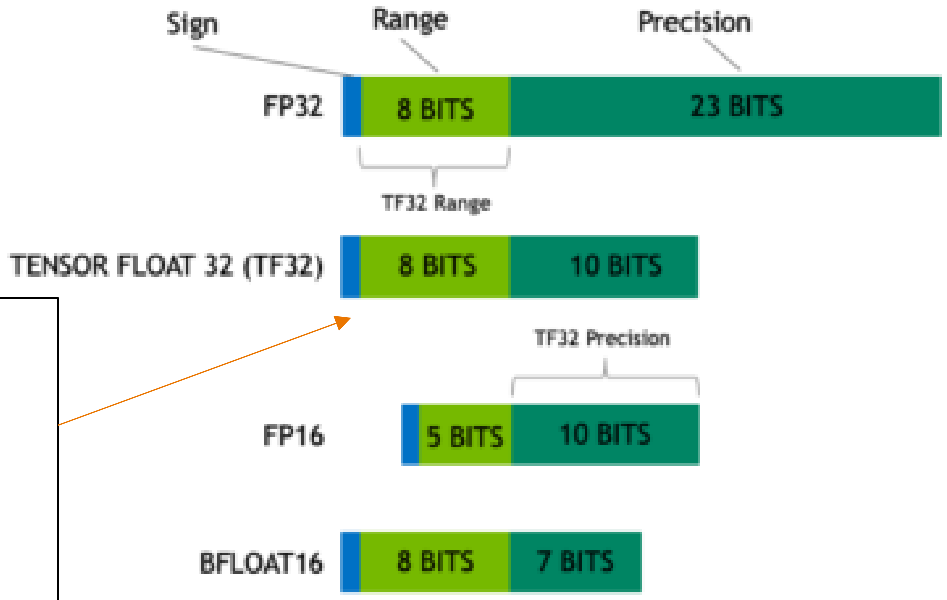


TF32 tries to combine best of both worlds: range of an FP32 (no gradient scaling needed), but with reduced precision (for speedup & less memory)

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

And on CPU:

```
[robertsc@tcn2 ~]$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              128
On-line CPU(s) list: 0-127
Thread(s) per core:  1
Core(s) per socket:  64
Socket(s):           2
NUMA node(s):        8
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               49
Model name:          AMD EPYC 7H12 64-Core Processor
Stepping:            0
CPU MHz:             2600.000
CPU max MHz:         2600.0000
CPU min MHz:         1500.0000
BogoMIPS:            5190.45
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           32K
L2 cache:            512K
L3 cache:            16384K
NUMA node0 CPU(s):   0-15
NUMA node1 CPU(s):   16-31
NUMA node2 CPU(s):   32-47
NUMA node3 CPU(s):   48-63
NUMA node4 CPU(s):   64-79
NUMA node5 CPU(s):   80-95
NUMA node6 CPU(s):   96-111
NUMA node7 CPU(s):   112-127
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep
_good nopl nonstop_tsc cpuid extd_apicid aperfmperf pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand lahf_lm cmp_legacy svm extapic cr8_legacy
 abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt tce topoext perfctr_core perfctr_nb bpext perfctr_llc mwaitx cpb cat_l3 cdp_l3 hw_pstate ssbd mba ibrs ibpb stibp vmmcall fsgs
base bmi1 avx2 smep bmi2 cqm rdt_a rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local clzero irperf xsaveerptr wbn
oinvd amd_ppin arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold avic v_vmsave_vmload vgif v_spec_ctrl umip rdpid overflow_recov
succor smca
```

```
[robertsc@gcn2 ~]$
Architecture:
CPU op-mode(s):
Byte Order:
CPU(s):
On-line CPU(s) lis
Thread(s) per core
Core(s) per socket
Socket(s):
NUMA node(s):
Vendor ID:
CPU family:
Model:
Model name:
Stepping:
CPU MHz:
CPU max MHz:
CPU min MHz:
BogoMIPS:
Virtualization:
L1d cache:
L1i cache:
L2 cache:
L3 cache:           55296K
NUMA node0 CPU(s):  0-35
NUMA node1 CPU(s):  36-71
Flags:              fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic
movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 invpcid_single intel_ppin ssbd mba ibrs ibpb stibp ibrs_enhanced tpr_shadow
vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb intel_pt avx512cd sha_ni a
vx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local split_lock_detect wbnoinvd dtherm ida arat pln pts avx512vbmi umip pku ospke avx512_v
bmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg tme avx512_vpopcntdq la57 rdpid fsrm md_clear pconfig flush_l1d arch_capabilities
```

Second S: Software

- PyTorch vs Numpy
  - PyTorch has GPU support, Numpy does not
  - Numpy on AVX2....brrrr
- CUDA? ROCm? <insert exotic hardware here>
  - E.g.: Intel SapphireRapids
  - IPEX: https://github.com/intel/intel-extension-for-pytorch
  - LUMI https://www.lumi-supercomputer.eu/
- BFloat16 support?

Third S: Stack

- Is my software compiled for my hardware
  - Pip install … ?
  - Module load PyTorch/1.12.0-foss-2022a-CUDA-11.7.0
- Again: SapphireRapids? ROCm?
- PyTorch is crazy: https://github.com/pytorch/pytorch/tree/main/third_party
  - PyTorch has given up on generic software management
  - Ptrblck:"  "

Wait ... so what happens?

- You call torch
  - (Manual) Dispatching
  - E.g. scatter_add_ -> Message psasing in Graph-NNs
- Torch (python) determines the most appropriate function to call
  - Dispatching
- Torch (C++/CU) calls the CPU/GPU kernel
  - Dispatching
- Low-level lib performs actual calculation
  - Dispatching
- PyTorch Dispatcher: http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/
- Your choices matter

In practice: PyTorch + Snellius

- We only have A100 GPUs (H100's coming?)
  - https://servicedesk.surf.nl/wiki/display/WIKI/Deep+Learning+on+A100+GPUs
- Makes optimization a bit easier for you
- You only need to spend a year sifting through documentation
  - Or ask us... :)
  - Every EINF has a default of 4 consultancy hours

Rapidfire: implementation details

- How fast do you want to go?
- https://pytorch.org/docs/stable/notes/cuda.html
- https://docs.nvidia.com/deeplearning/performance/dl-performance-memory-limited/index.html
- https://docs.nvidia.com/deeplearning/performance/index.html

- Duncan: *"You can take any piece of code, spend 6 months optimizing it, and then write a paper about it"*

Rapidfire: implementation details

- How fast do you want to go?
- What is your goal?
  - Largest model possible?
  - Best validation performance?
  - Submit a paper to ISC '23?

Table 1. Examples of neural network operations with their arithmetic intensities. Limiters assume FP16 data and an NVIDIA V100 GPU.

Operation	Arithmetic Intensity	Usually limited by...
Linear layer (4096 outputs, 1024 inputs, batch size 512)	315 FLOPS/B	arithmetic
Linear layer (4096 outputs, 1024 inputs, batch size 1)	1 FLOPS/B	memory
Max pooling with 3x3 window and unit stride	2.25 FLOPS/B	memory
ReLU activation	0.25 FLOPS/B	memory
Layer normalization	< 10 FLOPS/B	memory

SURF

Rapidfire: implementation details

5. DNN Operation Categories

While modern neural networks are built from a variety of layers, their operations fall into three main categories according to the nature of computation.

5.1. Elementwise Operations

Elementwise operations may be unary or binary operations; the key is that layers in this category perform mathematical operations on each element independently of all other elements in the tensor.

For example, a ReLU layer returns `max(0, x)` for each `x` in the input tensor. Similarly, element-wise addition of two tensors computes each output sum value independently of other sums. Layers in this category include most non-linearities (sigmoid, tanh, etc.), scale, bias, add, and others. These layers tend to be memory-limited, as they perform few operations per byte accessed. Further details on activations, in particular, can be found within the **Activations** section in the *Optimizing Memory-Bound Layers User's Guide*.

5.2. Reduction Operations

Reduction operations produce values computed over a range of input tensor values.

For example, pooling layers compute values over some neighborhoods in the input tensor. Batch normalization computes the mean and standard deviation over a tensor before using them in operations for each output element. In addition to pooling and normalization layers, SoftMax also falls into the reduction category. Typical reduction operations have a low arithmetic intensity and thus are memory limited. Further details on pooling layers can be found within **Pooling**.

5.3. Dot-Product Operations

Operations in this category can be expressed as dot-products of elements from two tensors, usually a weight (learned parameter) tensor and an activation tensor.

These include fully-connected layers, occurring on their own and as building blocks of recurrent and attention cells. Fully-connected layers are naturally expressed as matrix-vector and matrix-matrix multiplies. Convolutions can also be expressed as collections of dot-products - one vector is the set of parameters for a given filter, the other is an "unrolled" activation region to which that filter is being applied. Since filters are applied in multiple locations, convolutions too can be viewed as matrix-vector or matrix-matrix multiply operations (refer to **Convolution Algorithms**).

Operations in the dot-product category can be math-limited if the corresponding matrices are large enough. However, for the smaller sizes, these operations end up being memory-limited. For example, a fully-connected layer applied to a single vector (a tensor for a mini-batch of size 1)) is memory limited. Matrix-matrix multiplication performance is discussed in more detail in the **NVIDIA Matrix Multiplication Background User's Guide**. Information on modeling a type of layer as a matrix multiplication can be found in the corresponding guides.

Level 1: shouldn't impact model convergence:

Table 19. Limitations Of `Mha-fprop` Fusions

	Limitations Of `Mha-fprop` Fusions
MatMul	• Compute type for both MatMul ops must be float. • Input tensors must have data type `FP16` or `BF16`. • Output tensors must have data type `FP16`, `BF16`, or `FP32` (`TF32`).
Pointwise operations in g_3 and g_4	Compute type must be `FP32` (`TF32`).
Reduction operations in g_3 and g_4	I/O types and compute types must be `FP32` (`TF32`).
RNG operation in g_3 and g_4	• Data type of `yTensor` must be `FP32` (`TF32`). • The `CUDNN_TYPE_RNG_DISTRIBUTION` must be `CUDNN_RNG_DISTRIBUTION_BERNOULLI`.

Layout requirements of `Mha-fprop` fusions include:

- All I/O tensors must have 4 dimensions, with the first two denoting the batch dimensions. The usage of rank-4 tensors in MatMul ops can be read from the NVIDIA cuDNN Backend API documentation.
- The contracting dimension (dimension `K`) for the first MatMul must be 64.
- The non-contracting dimension (dimensions `M` and `N`) for the first MatMul must be less than or equal to 512. In inference mode, any sequence length is functional. For training, support exists only for multiples of 64.
- The last dimension (corresponding to hidden dimensions) in `Q`, `V`, and `O` is expected to have stride 1.
- For the `K` tensor, the stride is expected to be 1 for the 2nd last dimension.
- The `S` tensor is expected to have `CUDNN_ATTR_TENSOR_REORDERING_MODE` set to `CUDNN_TENSOR_REORDERING_F16x16`.

Level 1: shouldn't impact model convergence:

- Avoid CPU-GPU sync
  - Create tensors directly on the device
- Don't (accidentally) aggregate tensors which still require grad, e.g.: losses.append(loss) (:o)

```
torch.rand(size).cuda()
```

→

```
torch.rand(
    size,
    device=torch.device('cuda'),
)
```

Also applicable to:

`torch.empty()`, `torch.zeros()`, `torch.full()`, `torch.ones()`, `torch.eye()`, `torch.randint()`, `torch.randn()`

and similar functions.

- Operations which require synchronization:
  - `print(cuda_tensor)`
  - `cuda_tensor.item()`
  - memory copies: `tensor.cuda()`, `cuda_tensor.cpu()` and `tensor.to(device)` calls
  - `cuda_tensor.nonzero()`
  - python control flow which depends on operations on CUDA tensors e.g.

```
if (cuda_tensor != 0).all()
```

SURF

Level 1: shouldn't impact model convergence:

- torch.backends.cuda.matmul.allow_tf32 = True
  - The default in PyTorch (silently)
- torch.set_float32_matmul_precision('high')
- torch.backends.cudnn.benchmark = True
- Align bytes: multiples of 2, 8, 1024
  - DNN libs want nicely aligned memory
- torch.channels_last
- JIT/Compile: PyTorch 2.0

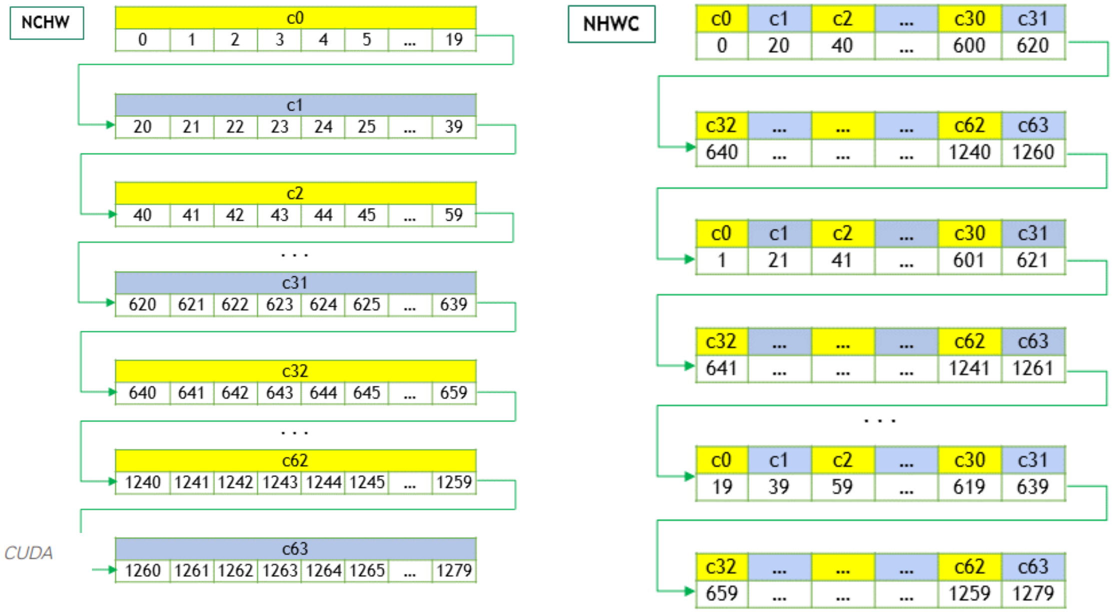
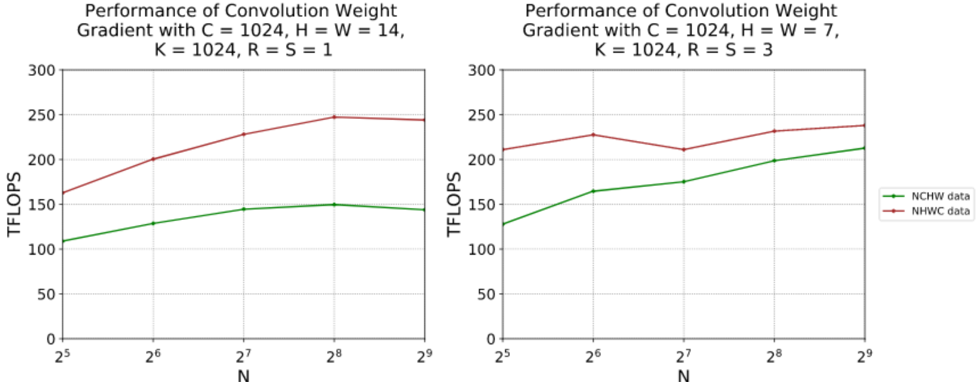


Figure 2. Kernels that do not require a transpose (NHWC) perform better than kernels that require one or more (NCHW). NVIDIA A100-SXM4-80GB, CUDA 11.2, cuDNN 8.1.



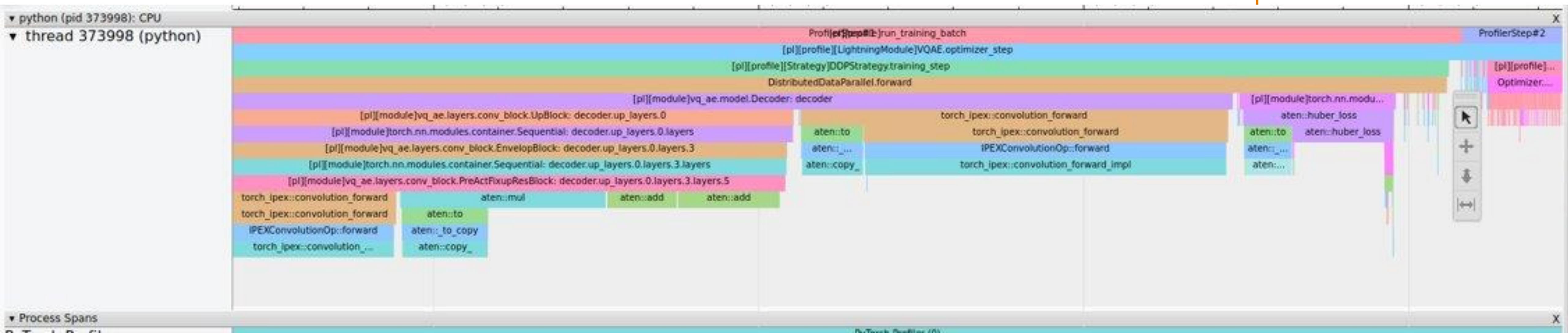
Kernels using Tensor Core operations are available for:

- Convolutions
- RNNs
- Multi-Head Attention

Level 2: Danger zone

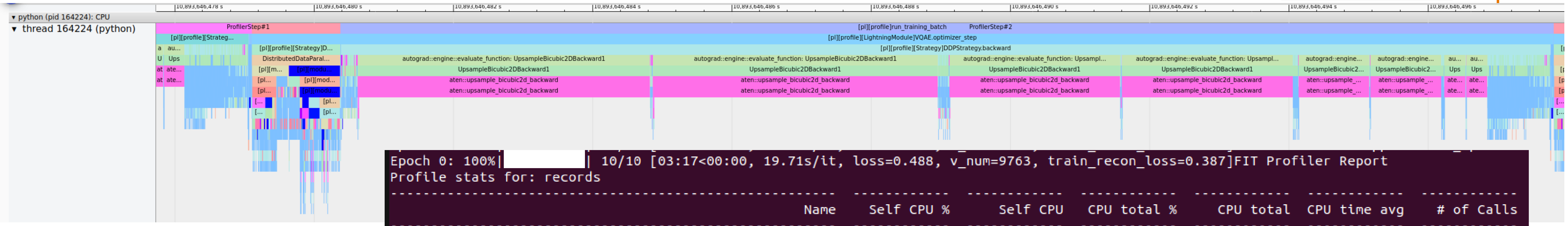
- torch.backends.cuda.matmul.allow_fp16_reduced_precision_reduction = True
- torch.backends.cuda.matmul.allow_bf16_reduced_precision_reduction = True
- Lol: A similar flag (as above) exists for BFloat16 GEMMs. Note that this switch is set to *False* by default for BF16 as we have observed numerical instability in PyTorch CI tests (e.g., test/test_matmul_cuda.py).

- Torch.set_float32_matmul_precision('medium')
- Cast model and data to (B)Float16 before the forward pass
  - We're not supposed to...but:



Change your model

- Worst case scenario:




```
Epoch 0: 100%|          | 10/10 [03:17<00:00, 19.71s/it, loss=0.488, v_num=9763, train_recon_loss=0.387]FIT Profiler Report
Profile stats for: records
```

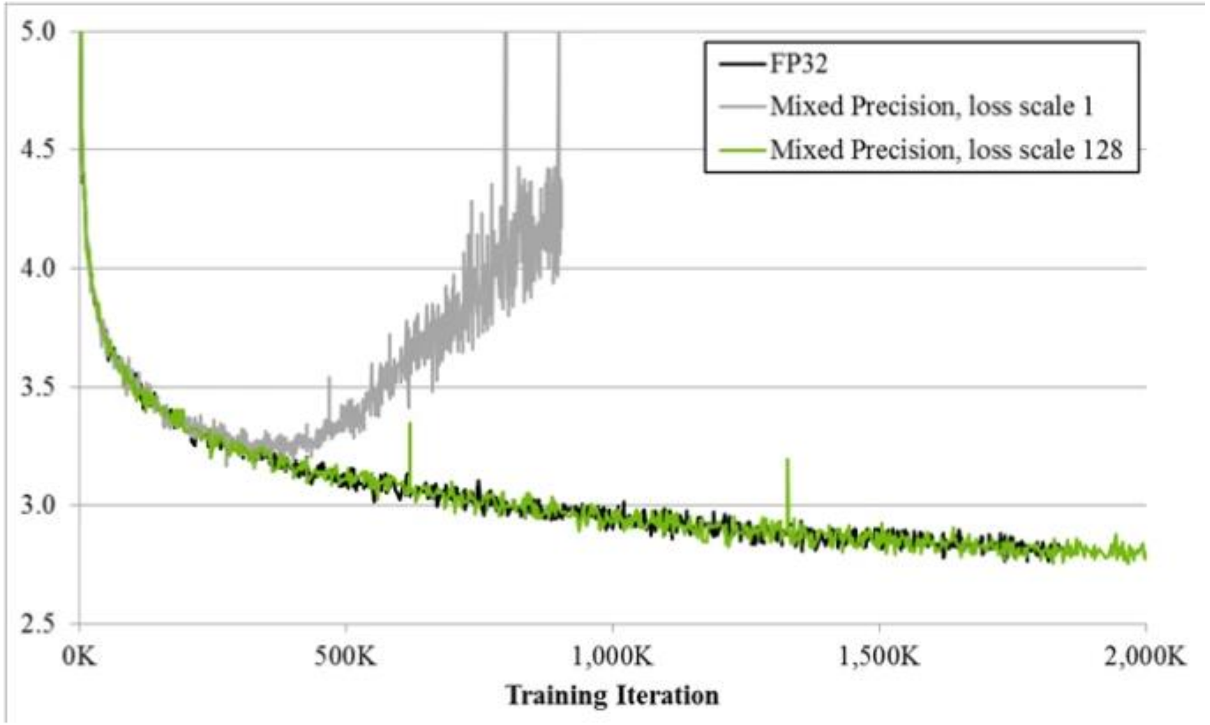
Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
ProfilerStep*	2.77%	2.611s	60.21%	56.680s	18.893s	3
[pl][profile][Strategy]DDPStrategy.backward	0.11%	106.540ms	55.55%	52.298s	17.433s	3
autograd::engine::evaluate_function: UpsampleBicubic...	0.00%	410.000us	50.59%	47.628s	1.985s	24
UpsampleBicubic2DBackward1	0.00%	153.000us	50.59%	47.628s	1.984s	24
aten::upsample_bicubic2d_backward	50.33%	47.382s	50.59%	47.628s	1.984s	24
[pl][profile]run_training_batch	0.00%	3.242ms	41.60%	39.158s	13.053s	3
[pl][profile][LightningModule]VQAE.optimizer_step	37.11%	34.931s	41.59%	39.155s	13.052s	3
[pl][profile][Strategy]DDPStrategy.training_step	0.00%	346.000us	4.49%	4.224s	1.408s	3
DistributedDataParallel.forward	0.01%	9.523ms	4.49%	4.223s	1.408s	3
[pl][module]vq_ae.model.Decoder: decoder	0.00%	540.000us	2.58%	2.427s	808.864ms	3
autograd::engine::evaluate_function: torch::autograd...	0.02%	19.580ms	2.54%	2.391s	1.779ms	1344
torch::autograd::CppNode<torch_ipex::cpu::IPEXConvol...	-0.00%	-448.000us	2.52%	2.372s	1.765ms	1344
IPEXConvolutionOp::backward	0.01%	9.489ms	2.52%	2.369s	1.763ms	1344
torch_ipex::convolution_backward	2.43%	2.292s	2.51%	2.363s	1.758ms	1344
[pl][module]vq_ae.model.Encoder: encoder	0.00%	732.000us	1.89%	1.779s	592.886ms	3
[pl][module]vq_ae.layers.conv_block.UpBlock: decoder...	0.00%	84.000us	1.82%	1.715s	571.554ms	3
[pl][module]torch.nn.modules.container.Sequential: d...	0.00%	416.000us	1.82%	1.715s	571.525ms	3
aten::add	1.56%	1.464s	1.56%	1.464s	359.683us	4071
torch_ipex::convolution_forward	-0.01%	-8343.000us	1.44%	1.357s	1.010ms	1344
IPEXConvolutionOp::forward	0.03%	31.301ms	1.42%	1.338s	995.645us	1344

Self CPU time total: 94.141s

```python
def unsorted_segment_mean(data, segment_ids, num_segments):
    # Impl 1
    # lengths = torch.zeros((num_segments,), dtype=torch.int, device=data.device)
    # idx, cnts = torch.unique_consecutive(segment_ids, return_counts=True)
    # lengths[idx] = cnts.int()
    #
    # res = (
    #     torch.segment_reduce(data=data, reduce='mean', lengths=lengths).nan_to_num()
    #     # / lengths[:, None].clamp(min=1)
    # )
    #
    # return res
    #
    result_shape = (num_segments, data.size(1))
    segment_idx = segment_ids.unsqueeze(-1).expand(-1, data.size(1))
    #
    # # Impl 2
    lengths = torch.zeros((num_segments,), dtype=torch.int, device=data.device)
    idx, cnts = torch.unique_consecutive(segment_ids, return_counts=True)
    lengths[idx] = cnts.int()

    # return torch.scatter_reduce(
    #     input=torch.zeros(result_shape, device=data.device, dtype=data.dtype),
    #     dim=0, index=segment_idx, src=data, reduce='sum', include_self=False
    # ) / lengths[:, None].clamp(min=1)

    # Impl 3

    res = torch.zeros(result_shape, dtype=data.dtype, device=data.device)
    res.scatter_add_(0, segment_idx, data) / lengths[:, None].clamp(min=1)
    return res

    # Baseline

    result = data.new_full(result_shape, 0)  # Init empty result tensor.
    count = data.new_full(result_shape, 0)
    result.scatter_add_(0, segment_idx, data)
    count.scatter_add_(0, segment_idx, torch.ones_like(data))
    res = result / count.clamp(min=1)

    return res
```

Always

- Have a baseline
- Check convergence statistics



Profile

- Profile
- Profile
- Profile

More generally; listen to Duncan:

- Start small
- Have a goal
- Profile your code