AI Optimization and Profiling Overview

From 20ms to 4ms: AI Workload Profiling and Optimization

Presenter: Samuel Antao

Oct 13-16, 2025

AMD @ CASTIEL AI Workshop

together we advance_

Advanced AI Programming

- AI Applications can challenge even the most extreme computing hardware
- Optimizing the applications can save both application run time and the hardware resources needed
- Due to high-level nature of AI apps, it can be difficult to determine the bottlenecks for performance
- Profiling tools are needed for different levels of analysis

AI Model Optimization

- We will be focusing on Pytorch optimizations, though similar approaches are used in other frameworks

- Tunning that can be applied are:
  - System-specific
    - Affinity
    - Communication fabric selection
  - Environment:
    - Communication
    - Libraries
  - Framework:
    - Kernel fusion techniques
    - Triton kernel usage
    - Memory optimization
  - Model-specific
    - Batch size optimization
    - Model parallelism – data/tensor/pipeline

Controlling device visibility

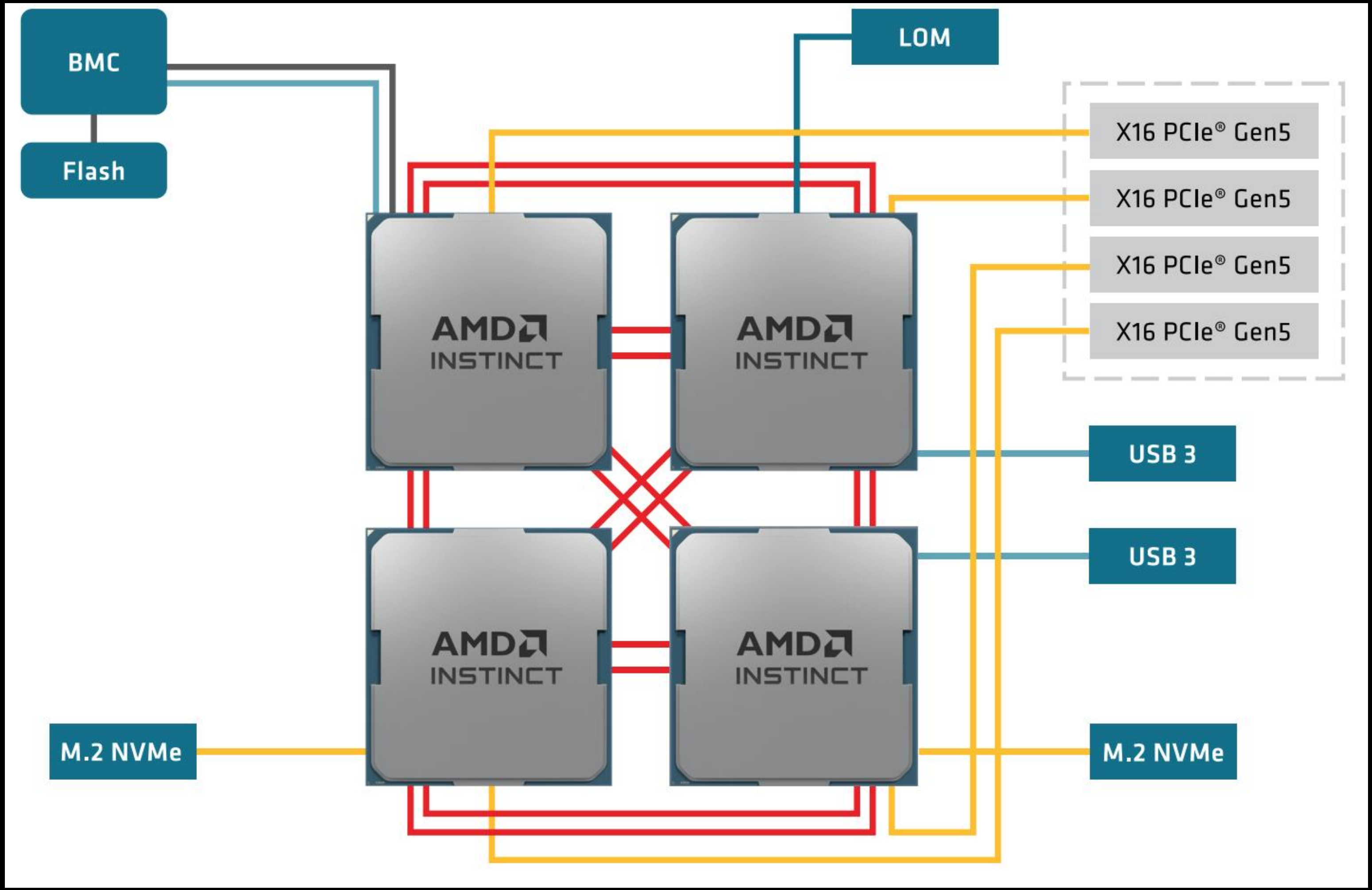
- Many GPUs can exist in a single node: which one to use?
- Controlling visibility
  - HIP_VISIBLE_DEVICES=0,1,2,3 python -c 'import torch; print(torch.cuda.device_count())'
  - ROCR_VISIBLE_DEVICES=0,1,2,3 python -c 'import torch; print(torch.cuda.device_count())’

- Considerations:
  - Does my app expects GPU visibility to be set in the environment?
  - Does my app expects arguments to define target GPUs
  - Does my app make any assumption on the device based on other information:
    - MPI rank
    - CPU-range
    - Auto-determined

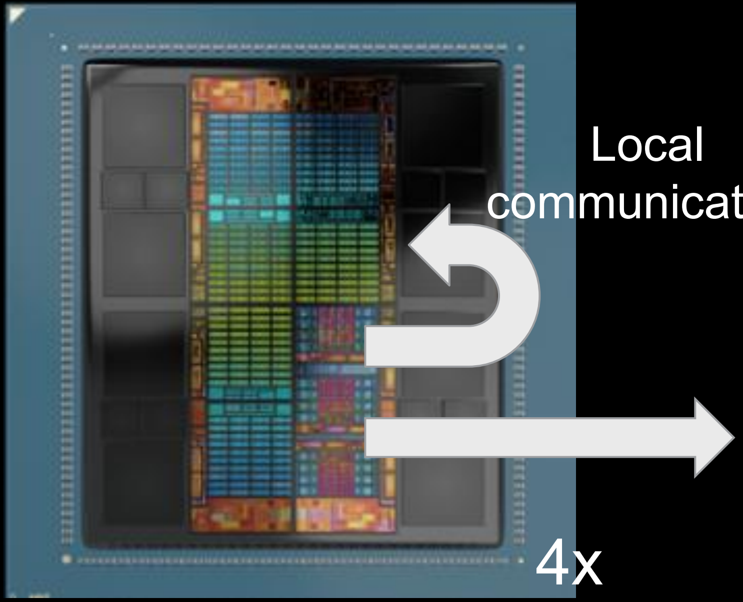
Most Pytorch applications and driver scripts assume the GPU to be used corresponds to the local rank!!!

Testing affinity – MI300A example

Each chip contains both CPU and GPU capabilities



Testing affinity – MI300A



- What CPUs I have available and their NUMA domain?
  - lscpu

- What GPUs I have
  - rocm-smi –showtopo

NUMA node0 CPU(s): 0-23,96-119
NUMA node1 CPU(s): 24-47,120-143
NUMA node2 CPU(s): 48-71,144-167
NUMA node3 CPU(s): 72-95,168-191

GPU[0] : (Topology) Numa Node: 0
GPU[0] : (Topology) Numa Affinity: 0
GPU[1] : (Topology) Numa Node: 1
GPU[1] : (Topology) Numa Affinity: 1
GPU[2] : (Topology) Numa Node: 2
GPU[2] : (Topology) Numa Affinity: 2
GPU[3] : (Topology) Numa Node: 3
GPU[3] : (Topology) Numa Affinity: 3

- How to control which CPU
  - Numactl: e.g --physcpubind=0-23
  - MPI: mpirun binding arguments
  - Scheduler: *srun … --gpu-bind=closest*

Comms are important! - RCCL CXI plugin

- LUMI, Frontier (and others) directly attaches AMD Instinct™ Accelerators to the Slingshot Network
  - Enable collectives computation on devices
  - Minimize the role of the CPU in the control path – expose more asynchronous computation opportunities
  - Lowest latency for network message passing is from GPU HBM memory

- CXI plugin is a runtime dependency. Requires: HPE Cray libfabric implementation
  - https://github.com/ROCm/aws-ofi-rccl
  - 3-4x faster collectives
- **Make sure your containerized worflows use this plugin!**
- `export NCCL_DEBUG=INFO`

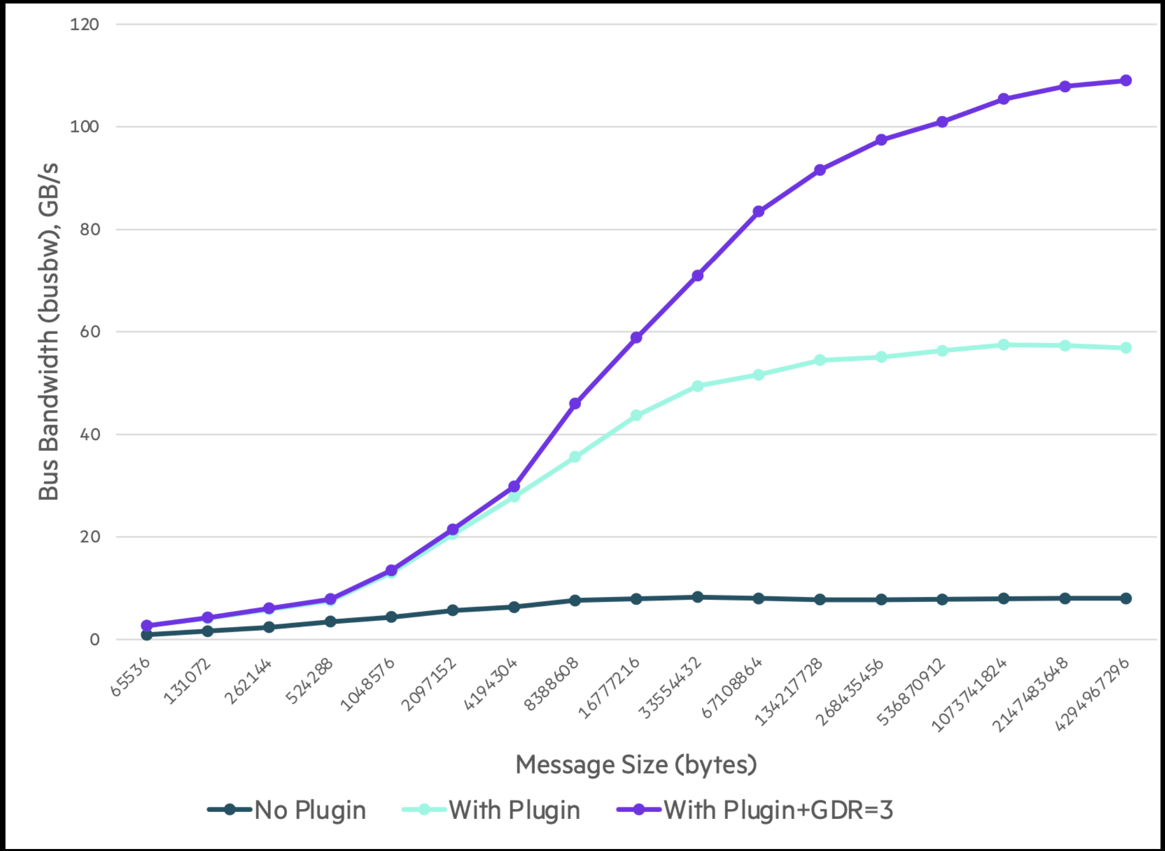
```
export NCCL_DEBUG_SUBSYS=INIT
# and search the logs for:
[0] NCCL INFO NET/OFI Using aws-ofi-rccl 1.4.0
```

Check your site's recommendations. Don't assume the generic setups will work well. And pay special attention to container configurations!

Configuring RCCL environment

- Select high-speed interfaces:
  - `export NCCL_SOCKET_IFNAME=hsn0,hsn1,hsn2,hsn3`
- RCCL should be set configured to use GPU RDMA:
  - `export NCCL_NET_GDR_LEVEL=PHB`
- On ROCm versions 6.2+ this is not needed – it is default.
- Why should I spend time with all this?
  - 3-4x better bandwidth utilization with plugin
  - 2x better bandwidth utilization with RDMA
  - Can scale further!

- **Careful using external containers! You may need to be setting plugin yourself!**



Environment settings

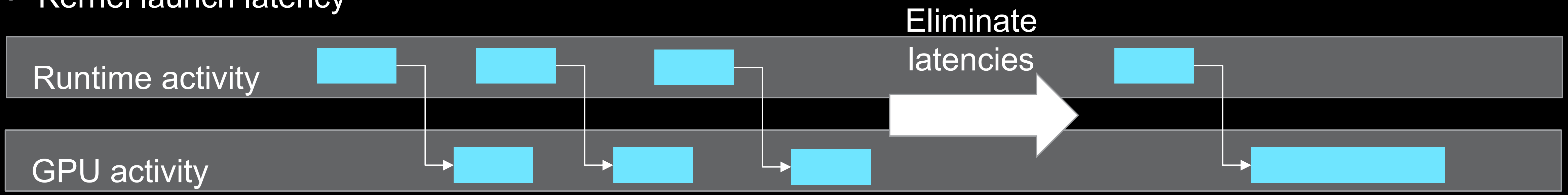
- These are the easiest ways to get a performance speedup.
  - Each AI Framework has their own environment tuning parameters. See https://rocm.docs.amd.com for suggestions or check your AI package documentation.

- Pytorch
  - Will run a suite of GEMMs and pick the fastest. If running the same size GEMM many times, this can dramatically speed up your application
  - `export PYTORCH_TUNABLEOP_ENABLED=1`

- MIOpen
  - Using the default MIOpen database in a shared location may not be writeable and the locks may slow your application. The following puts the database in a fast filesystem and can be user specific.
  - `export MIOPEN_USER_DB_PATH="/tmp/my-miopen-cache"`
  - `export MIOPEN_CUSTOM_CACHE_DIR="/tmp/my-miopen-cache"`
  - For larger numbers of nodes, file system doesn't deal well with SQLite locks when many processes are trying to access it. Solution? Setup individual caches for groups of ranks – we recommend per node:
  - `export MIOPEN_USER_DB_PATH="/tmp/$(whoami)-miopen-cache-$SLURM_NODEID"`
  - `export MIOPEN_CUSTOM_CACHE_DIR=$MIOPEN_USER_DB_PATH`

Kernel fusion and composability

- Training and other AI workloads are composed of different passes each composed of different operations.
- Issuing many small operations is problematic for GPUs
- Kernel launch latency



- Kernel fusion in Pytorch
  - Automatic fusion – torch.compile()
  - Detects fusion opportunities
  - Decorators available for costumization

```
@torch.compile
def my_function(x):
        return x * 0.25 * (1.0 + torch.erf(x / 1.42))
```

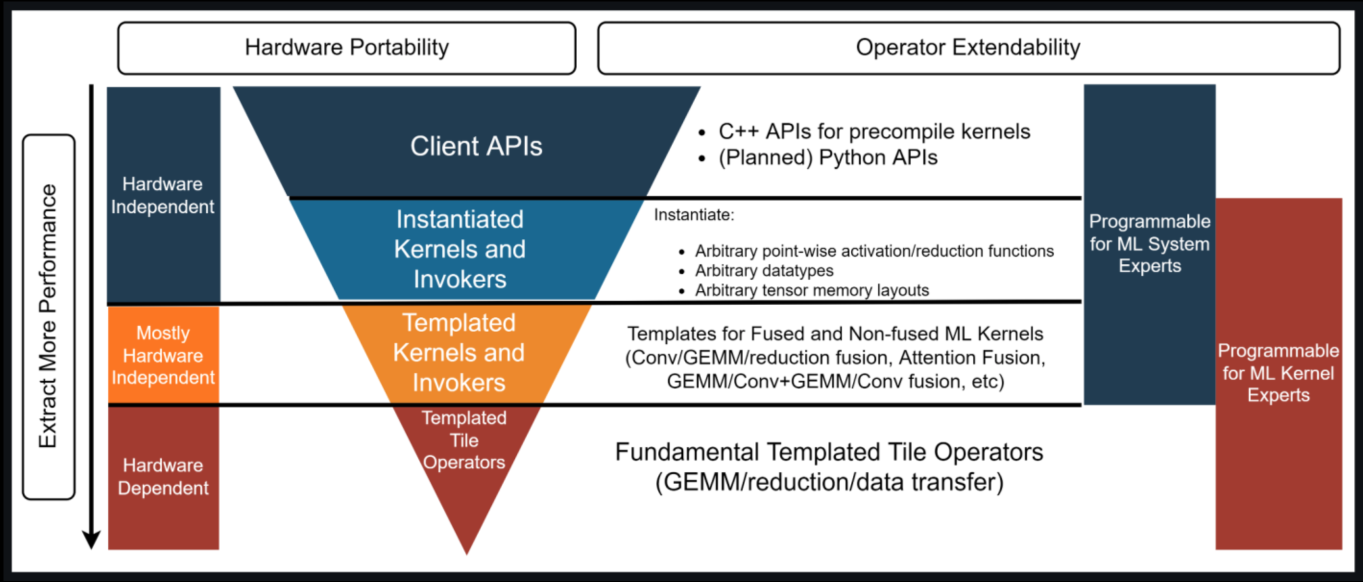
Kernel fusion and composability

- Triton backend
  - "Python code that you can compile"
  - Domain specific language and decorators.
- Ahead-of-time Triton – AOTriton
  - Pre-built triton kernels
  - Flash Attention implementation
  - https://github.com/ROCm/aotriton
- Composable kernel (CK)
  - Performance critical ML primitives
  - Tile base programming model
  - Tend to perform better than Triton-based operands
  - https://github.com/ROCm/composable_kernel
- Good news: all these libraries should be available in you Pytorch installation for ROCm

```python
import triton
import triton.language as tl


# Define a simple element-wise addition kernel
@triton.jit
def add_kernel(x_ptr, y_ptr, out_ptr, n_elem, BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(axis=0)
  block_start = pid * BLOCK_SIZE
  offsets = block_start + tl.arange(0, BLOCK_SIZE)
  mask = offsets < n_elem

  x  = tl.load(x_ptr + offsets, mask=mask)
  y = tl.load(y_ptr + offsets, mask=mask)
  output = x + y
  tl.store(out_ptr + offsets, output, mask=mask)
```

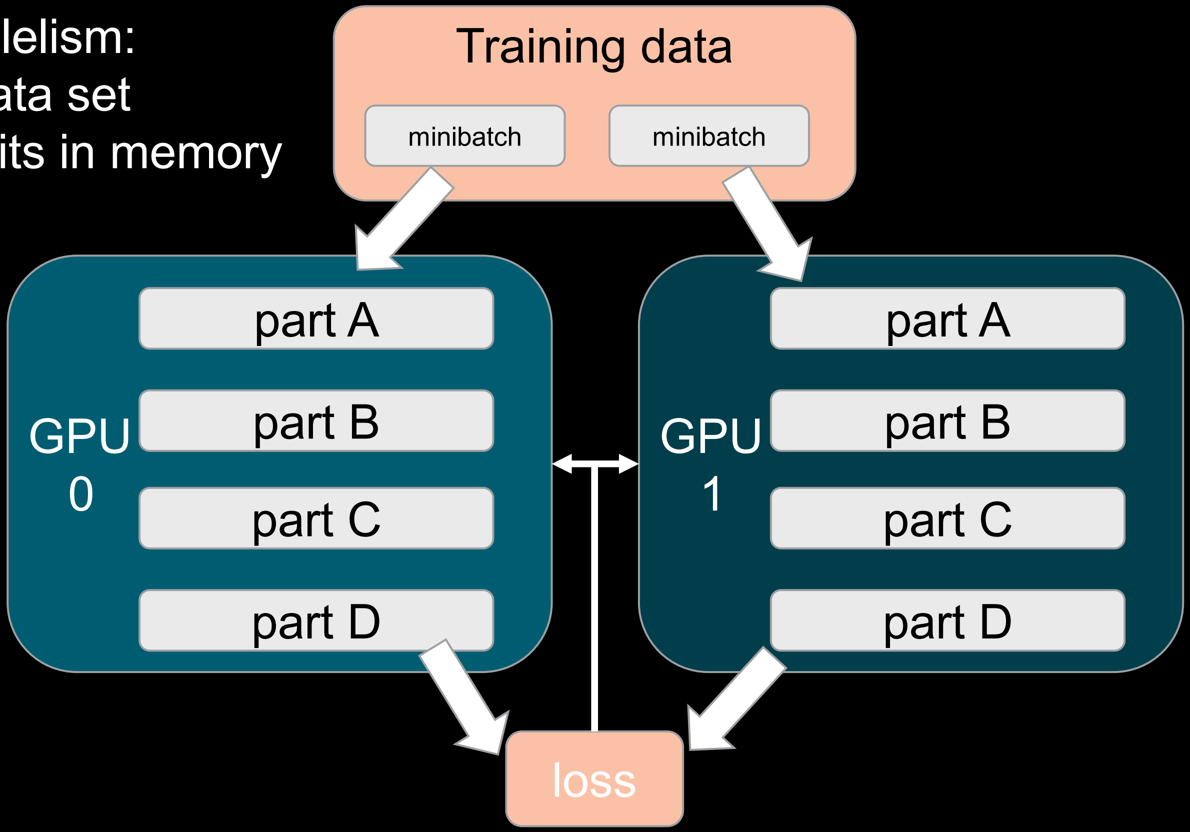


AMD together we advance_

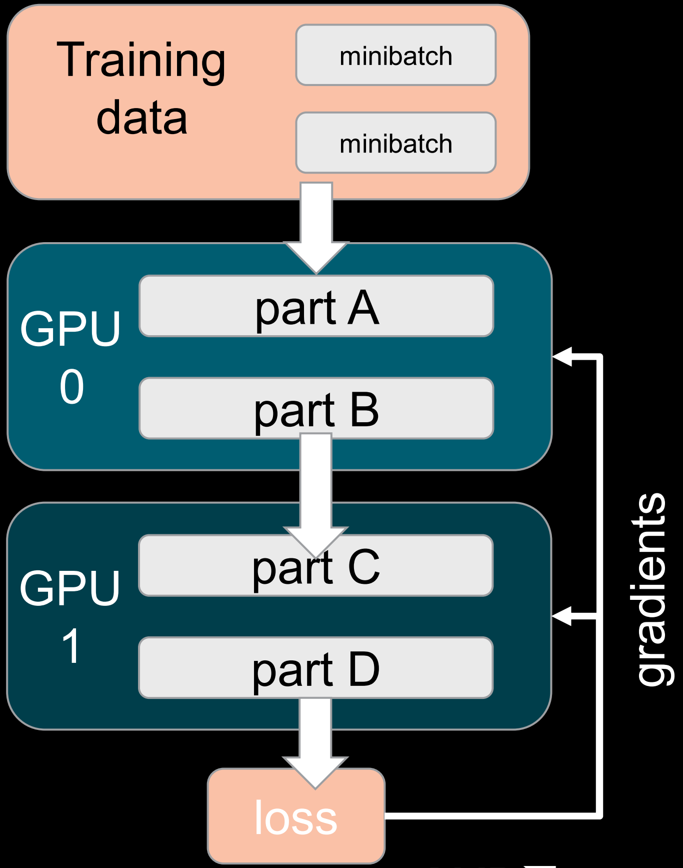
Exploring parallelism

- Great, I have 8 GPUs in my system!
- Depending on your model and data your parallelization strategy needs to be decided

Data parallelism:
- large data set
- model fits in memory



Model parallelism
- model too large
- faster training iteration



AMD together we advance_

Other ways to express parallelism - FSDP

- Fully Sharded Data Parallel is another option.
  - Create shards out of the neural net model more likely to be activated together
  - Try keep less state in the GPU – could support larger models with less GPUs
  - More complexities in configuring the different knobs. Depending on the tunning may require more or less changes to your code.
  - https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/
- Using FSDP requires wrapping your model into the relevant FSDP object.

```
wrapper_kwargs = Dict(cpu_offload=CPUOffload(offload_params=True))
with enable_wrap(wrapper_cls=FullyShardedDataParallel, **wrapper_kwargs):
  fsdp_model = wrap(model())
```

Your original model

- Some tools to control the wrapping in less intrusive ways have been created - ***accelerate***.
  - Enabling FSDP on transformers: https://huggingface.co/docs/transformers

[Public]

Other ways to express parallelism - DeepSpeed

deepspeed

- DeepSpeed is a framework to optimize distributed deep-learning training and inference

```
DS_BUILD_AIO=0 \
DS_BUILD_CCL_COMM=1 \
DS_BUILD_CPU_ADAM=0 \
DS_BUILD_CPU_LION=0 \
DS_BUILD_EVOFORMER_ATTN=0 \
DS_BUILD_FUSED_ADAM=1 \
DS_BUILD_FUSED_LION=1 \
DS_BUILD_CPU_ADAGRAD=0 \
DS_BUILD_FUSED_LAMB=1 \
DS_BUILD_QUANTIZER=0 \
DS_BUILD_RANDOM_LTD=0 \
DS_BUILD_SPARSE_ATTN=0 \
DS_BUILD_TRANSFORMER=0 \
DS_BUILD_TRANSFORMER_INFERENCE=0 \
DS_BUILD_STOCHASTIC_TRANSFORMER=1 \
pip install deepspeed==0.15.0 \
--global-option="build_ext" --global-option="-j32"

ds_report
```

Select all the optimizations not all are enabled for GPUs.

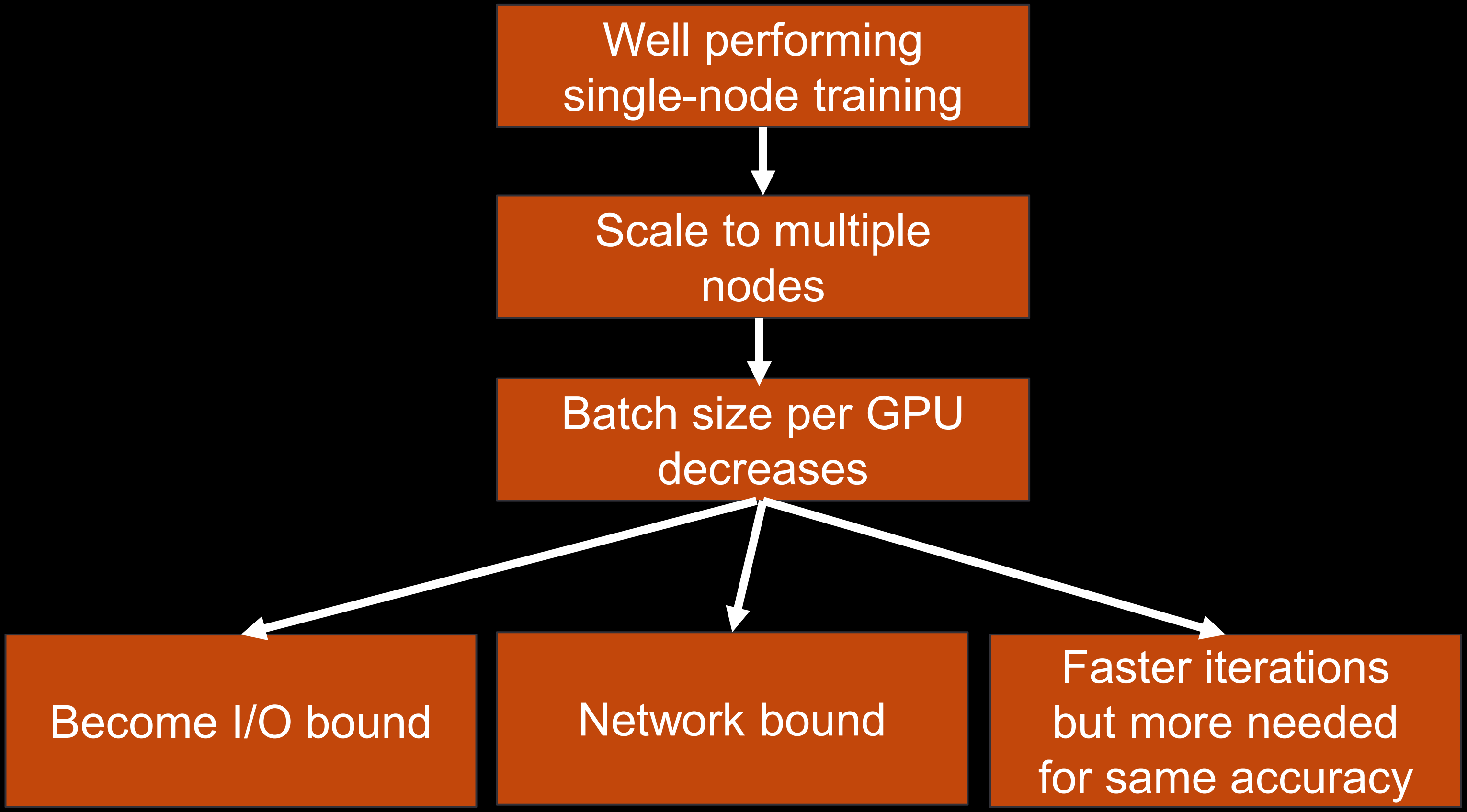
Allow multiple process builds.

Utility to report supported capabilities.

```
--------------------------------------------------
op name ................ installed .. compatible
--------------------------------------------------
async_io ............... [NO] ....... [NO]
fused_adam ............. [YES] ...... [OKAY]
cpu_adam ............... [NO] ....... [OKAY]
cpu_adagrad ............ [NO] ....... [OKAY]
cpu_lion ............... [NO] ....... [OKAY]
evoformer_attn ......... [NO] ....... [NO]
fused_lamb ............. [YES] ...... [OKAY]
fused_lion ............. [YES] ...... [OKAY]
inference_core_ops ..... [NO] ....... [OKAY]
cutlass_ops ............ [NO] ....... [OKAY]
transformer_inference .. [NO] ....... [OKAY]
quantizer .............. [NO] ....... [OKAY]
ragged_device_ops ...... [NO] ....... [OKAY]
ragged_ops ............. [NO] ....... [OKAY]
random_ltd ............. [NO] ....... [OKAY]
sparse_attn ............ [NO] ....... [NO]
spatial_inference ...... [NO] ....... [OKAY]
transformer ............ [NO] ....... [OKAY]
stochastic_transformer . [YES] ...... [OKAY]
--------------------------------------------------
```

Scaling implications

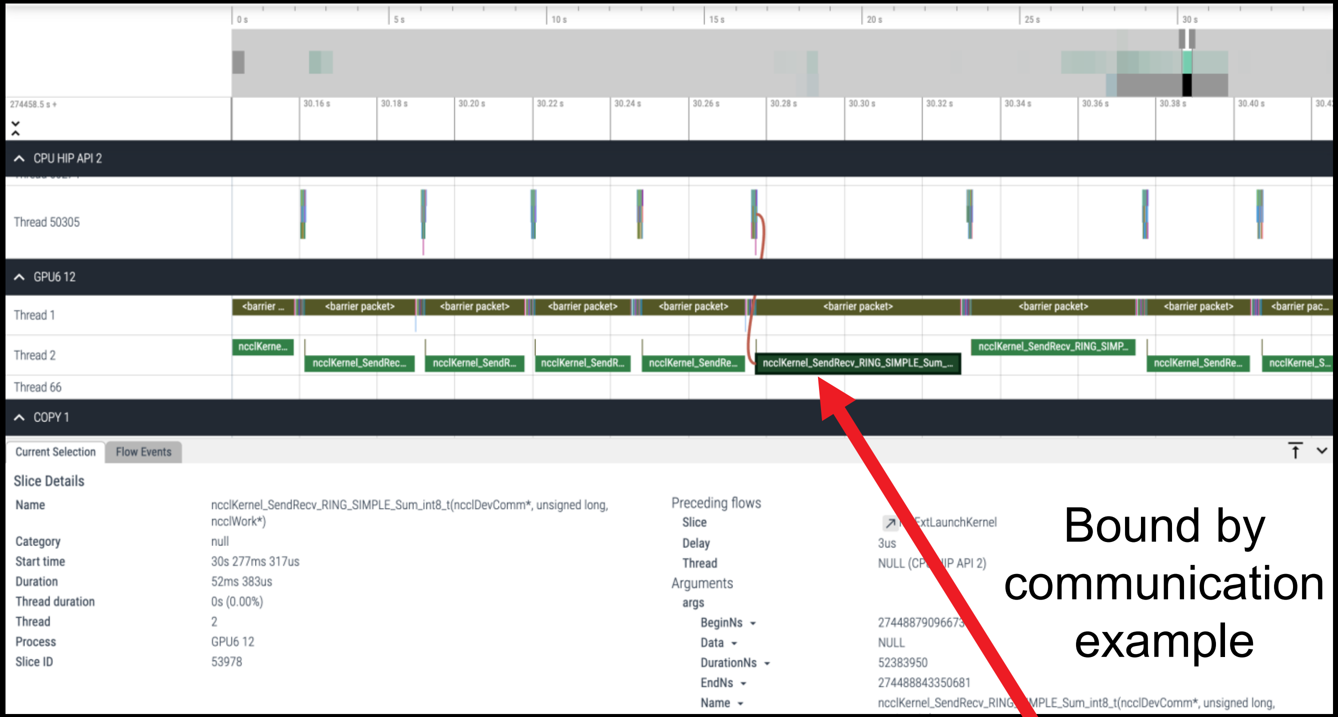
- Why would I want to scale my model?
  - Train faster – strong-scaling
  - Train bigger – weak-scaling
  - My model doesn't fit in just a few GPUs
- How far can I go?
  - Depends on your model
  - Scaling can change the bottlenecks
  - Scaling can change convergence
- Monitor the regime in which you are operating the GPUs at all times!
- The goal is to learn faster not run faster
  - Faster iterations can be offset by slower convergence



- You'll always be bound by some type of communication at some point!!!

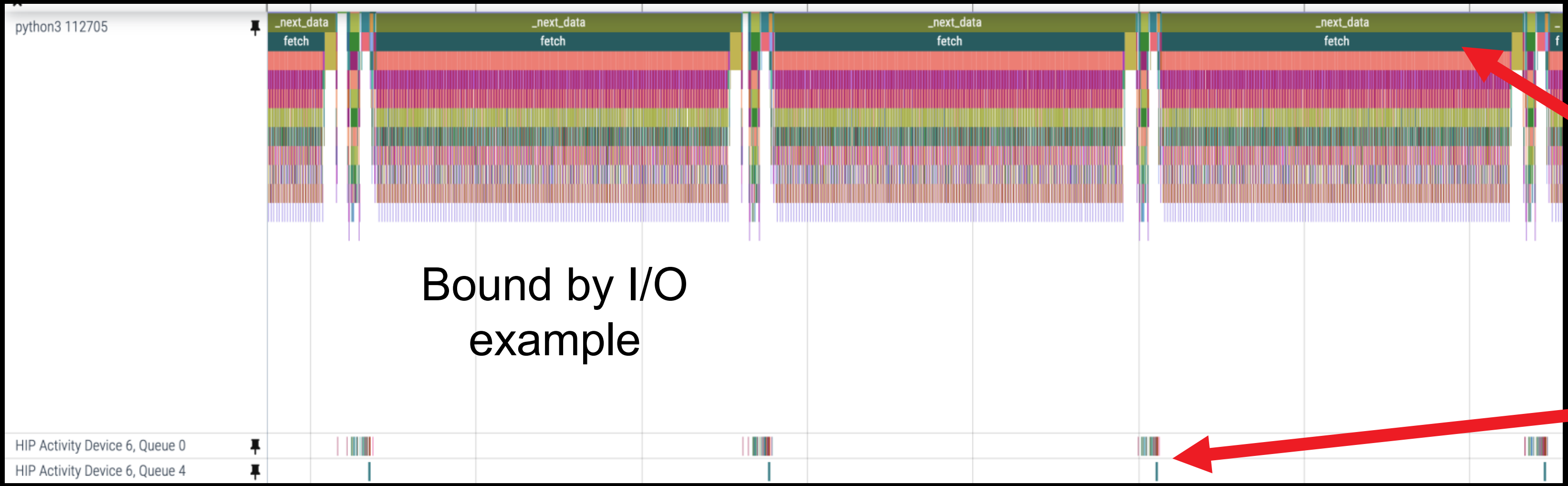
The goal is for a model to learn faster!

- Your bottleneck might not be the GPU
- Check your accuracies are as expected
- Loss-curves
- Adopt an holistic approach to performance analysis
  - Communication
  - I/O



Bound by communication example

Collectives kernels dominate profile



Bound by I/O example

Data fetching dominate profile

Minimal GPU activity

AMD
together we advance_

Pytorch Profiling Tools Progression

- Start with simple high-level profiling tools - progress to detailed ones as needed

Framework-level tools

- PyTorch Profiler: Framework-level performance analysis
- DeepSpeed FLOPS Profiler: Computational efficiency metrics
- Triton Profiler (Proton): Analyze kernel performance

System-level tools

- ROCsmi (AMDsmi):
- ROCprofv3 /ROCprof-sys: System-level performance analysis

Kernel-level tools

- ROCprofv3: Basic ROCm profiling of GPU activity
- ROCprof-compute: Advanced GPU kernel-level analysis and optimization

Framework-Level Tools

- PyTorch Profiler
  ├── Purpose: Operator-level performance analysis
  ├── Output: Execution time, memory, call counts, tensor shapes
  ├── Usage: torch.profiler.profile() context manager
  └── Export: Chrome trace, table summaries, TensorBoard
- DeepSpeed FLOPSProfiler
  ├── Purpose: Computational efficiency metrics
  ├── Output: FLOPS per layer, multiply-accumulate operations (MACs), parameters, efficiency %
  ├── Usage: FlopsProfiler(model) wrapper: https://www.deepspeed.ai/tutorials/flops-profiler/
  └── Analysis: Theoretical vs achieved compute, bottleneck identification
- Triton Profiler (Proton)
- ├── Purpose: Kernel performance
  ├── Output: Timings, plots
  ├── Usage: triton.testing.do_bench wrapper
  └── Analysis: Comparisons with different approaches

System-Level Tools

- ROCm SMI (System Management Interface)
  ├── Purpose: GPU health and utilization monitoring
  ├── Output: Temperature, power, memory usage, GPU utilization
  ├── Usage: rocm-smi (real-time monitoring)
  └── Analysis: Thermal throttling, memory saturation, multi-GPU balance

- rocprof-sys (System Profiler)
  ├── Purpose: System-wide performance monitoring
  ├── Output: Multi-process coordination, resource utilization, CPU/GPU interaction
  ├── Usage: rocprof-sys record -o trace.otf2 -- python script.py
  └── Analysis: Timeline visualization, process synchronization, I/O bottlenecks

Kernel-Level Tools

- rocprofv3 (Basic Hardware Counter Profiler)
  ├── Purpose: Basic GPU kernel profiling and metrics
  ├── Output: Kernel execution times, memory transfers, GPU utilization
  ├── Usage: rocprofv3 --stats python script.py
  └── Analysis: Kernel-level timing, backward compatibility

- rocprof-compute (Advanced Profiler)
  ├── Purpose: Detailed compute kernel analysis and optimization
  ├── Output: Hardware counters, roofline analysis, optimization hints
  ├── Usage: rocprof-compute profile --kernel-names --roofline -- python script.py
  └── Analysis: Memory hierarchy utilization, arithmetic intensity, bottleneck classification

Tool Selection Decision Tree

Start Here: What do you want to optimize?

Question 1: Do you know which operation is slow?
- No → Use PyTorch Profiler + ROCprof-sys (identify hot operators)
- Yes → Continue to Question 2

Question 2: Is it compute-bound or memory-bound?
- Unknown → Use ROCprof-compute (roofline analysis)
- Compute → Focus on algorithm/fusion optimization
- Memory → Focus on memory access patterns

Question 3: Are you writing custom kernels?
- No → Use PyTorch Profiler + ROCprof-sys
- Yes → Use ROCprof-compute + Triton profiler

Question 4: Production deployment?
- No → Focus on performance iteration
- Yes → Add monitoring (performance regression tests)

Disclaimer

AMD