ROCm PyTorch Inference Benchmark

Standardized Performance Measurement for Deep Learning Models

Presenter: Bob Robey

Oct 13-16, 2025

AMD @ CASTIEL AI Workshop

AMD

together we advance_

What is Inferencing?

Inference is the process of using a trained neural network to make predictions on new data.

Key Differences from Training:

Aspect	Training	Inference
Purpose	Learn patterns from data	Make predictions
Direction	Forward + Backward pass	Forward pass only
Gradients	Required	Not required
Batch size	Usually Larger	Often smaller
Performance Goal	Throughput (samples/sec)	Latency (ms/sample) AND throughout
Memory Usage	High (stores activations)	Lower (no gradient storage)

Why Benchmark Inference?

- Optimize for production deployment
- Understand hardware utilization
- Compare different models
- Justify hardware purchases
- Identify bottlenecks

Workshop Mission Statement

Challenge

- Modern AI workloads run at 10-20% of theoretical hardware performance

Our goal

- Increase samples per second
- Reduce memory usage
- Reduce number of kernels (fuse kernels)

What you'll learn

- Workshop Philosophy: Optimization Cycle
  Measure → Analyze → Optimize → Validate
- Master the complete optimization toolkit: profiling → analysis → optimization → validation

Key Performance Metrics Explained

- Throughput (samples/sec or tokens/sec)
- Latency (milliseconds)
- Memory Usage (MB or GB)
- Kernel Count (launches per iteration)
- Memory Bandwidth Utilization (%)
- Arithmetic Intensity (FLOPs/byte)
- GPU Occupancy (%)
- FLOPS Efficiency (%)

Common Pitfalls and Workshop Guardrails

- Pitfall 1: Optimizing without profiling
- Pitfall 2: Ignoring numerical accuracy
- Pitfall 3: Over-optimizing small operations
- Pitfall 4: Hardware-specific tuning too early

Profiling Ecosystem Overview

- **PyTorch Profiler**
  - Operator execution time breakdown
  - CPU/GPU timeline visualization
  - Memory allocation tracking
  - Chrome trace export for detailed analysis
- **DeepSpeed FLOPSProfiler**
  - FLOPS per layer (theoretical vs. achieved)
  - MACs (multiply-accumulate operations)
  - Parameter counts and activation memory
  - Model efficiency percentage
- **ROCm Micro-benchmarking**
  - Standard model baselines for comparison
  - Multi-precision performance characterization
  - Multi-GPU scaling validation

Introduction to Hands-on Exercises

Step 0: Environment Setup - Installation verification

- Get a node with a GPU
  `salloc -N 1 --gpus=2 --cpus=8`
- Check ROCm
  `rocminfo | grep "Name:"  # Should show your GPU`
- If you see an error check if ROCm is installed
  which rocminfo
- check GPU Status
  rocm-smi
- Expected Output:
  GPU[0]   : GPU ID: 0GPU[0]   : GPU Name: AMD Instinct MI325XGPU[0]   : Temperature: 35.0°CGPU[0]   : GPU Memory Usage: 256 MB / 196608 MBGPU[0]   : GPU Utilization: 0
- If not found, ROCm is not installed (contact system admin)
- Check PyTorch + ROCm
  python -c "import torch; print(torch.__version__); print(torch.cuda.is_available())“
- Expected Output:
  `2.7.1+rocm6.4.4, True`
- Check Triton
  python -c "import triton; print(triton.__version__)"

```
See hands-on exercises instructions at
HPCTrainingExamples/MLExamples/inferen
ce_benchmark/INFERENCE_BENCHMARK.md
```

Workshop Integration and Workflow

- **Benchmark-Driven Optimization Cycle**
- **Performance Analysis Workflow**
  1. Establish baseline: Standard model (ResNet50) with default configuration
  2. Parameter sweep: Batch size, precision, optimization flags
  3. Identify bottlenecks: ROCm profiler integration, kernel analysis
  4. Apply optimizations: MIOpen tuning, graph compilation, custom kernels
  5. Validate improvements: Re-benchmark, compare against baseline
  6. Document results: Performance database, optimization log

```
1. MEASURE: Profile baseline performance
   Tools: PyTorch Profiler, DeepSpeed
   Output: Execution times, memory, FLOPS
                    ↓
2. ANALYZE: Identify bottlenecks
   Tools: ROCprof-compute, roofline
   Output: Compute vs memory bound
                    ↓
3. OPTIMIZE: Apply targeted optimizations
   Techniques: Fusion, custom kernels
   Tools: Triton, PyTorch optimizations
                    ↓
4. VALIDATE: Measure improvement
   Check: Speedup, correctness, stability
   Tools: Numerical accuracy validation
                    ↓
           Repeat until optimal
```

Step 1: PyTorch Profiler Integration

```
with torch.profiler.profile(
    activities=[
        torch.profiler.ProfilerActivity.CPU,
        torch.profiler.ProfilerActivity.CUDA,
    ],
    record_shapes=True,
    profile_memory=True,
) as prof:
    model(inputs)

print(prof.key_averages().table(
    sort_by="cuda_time_total", row_limit=10))
```

Works on ROCm (CUDA API Compatibility)

- **Basic Usage Pattern**
  - **Output Metrics**
    - Kernel execution time (microseconds)
    - Memory bandwidth utilization
    - Operator call counts
    - Tensor shapes and memory footprint
- Start with running the benchmark

```
python3 micro_benchmarking_pytorch.py \    --network resnet50       # Model to benchmark    --batch-size 64          # Number of samples per batch    --iterations 20          # Number of iterations to run
```

- **Profiling Argument -** Add optional arguments

```
--autograd-profiler          # Enable PyTorch autograd profiler
--kineto                     # Enable Kineto profiler (PyTorch 1.8+)
--flops-prof-step 10         # Enable DeepSpeed FLOPS profiler at step 10
```

- **Multi-GPU Arguments --** `Using torchrun`

```
torchrun --nproc-per-node 2 micro_benchmarking_pytorch.py --network resnet50
```

- **PyTorch 2.0 Argument – optimizes a computational graph for additional performance (almost for free)**

```
--compile                    # Enable torch.compile
--compileContext "{'mode': 'max-autotune'}"  # Compilation options
```

See https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

Understanding Output

When you run the benchmark, you'll see output like this:

```
Using network: resnet50
Batch size: 64
Iterations: 20
FP16: False

Warming up...
Warmup complete.

Epoch 0: Loss = 6.9088, Time = 0.145 seconds ← SLOW (first run)
Epoch 1: Loss = 6.9088, Time = 0.042 seconds ← Much faster
Epoch 2: Loss = 6.9088, Time = 0.041 seconds ← Stable
...
Epoch 19: Loss = 6.9088, Time = 0.040 seconds

========================================
Performance Summary:
========================================
Average time per iteration: 0.041 seconds
Throughput: 1560.9 samples/sec
Memory usage: 4523 MB
========================================
```

Why is the first iteration slow?

- Kernel compilation (Triton, ROCm)
- GPU memory allocation
- Cache warming
- cuDNN/MIOpen autotuning

Understanding Key Metrics

Before we begin the exercises, let's understand what we're measuring:

Throughput (samples/sec or images/sec)

- **What:** Number of samples processed per second
- **Higher is better**
- **Use case:** Batch inference, data center deployments
- **Formula:** `(batch_size × num_iterations) / total_time`

Latency (milliseconds)

- **What:** Time to process a single sample or batch
- **Lower is better**
- **Use case:** Real-time applications, interactive systems
- **Formula:** `total_time / num_iterations`

Memory Usage (MB or GB)

- **What:** GPU memory consumed by model and data
- **Lower is better (allows larger batches)**
- **Includes:** Model weights, activations, gradients (if training)

GPU Utilization (%)

- **What:** Percentage of GPU compute used
- **Higher is better (approaching 100%)**
- **Note:** Can be low if memory-bound or CPU-bound

Memory Usage (MB or GB)

- **What:** GPU memory consumed by model and data
- **Lower is better (allows larger batches)**
- **Includes:** Model weights, activations, gradients (if training)

GPU Utilization (%)

- **What:** Percentage of GPU compute used
- **Higher is better (approaching 100%)**
- **Note:** Can be low if memory-bound or CPU-bound

FLOPS (Floating Point Operations Per Second)

What: Computational throughput

Higher is better

Theoretical vs Achieved: Gap indicates optimization opportunity

together we advance_

Optional: Try Different Batch Sizes

Why does batch size matter?

Larger batches improve GPU utilization but increase memory usage.

```
# Small batch
python3 micro_benchmarking_pytorch.py --network resnet50 --batch-size 8 --iterations 20

# Medium batch (your baseline)
python3 micro_benchmarking_pytorch.py --network resnet50 --batch-size 32 --iterations 20

# Large batch
python3 micro_benchmarking_pytorch.py --network resnet50 --batch-size 128 --iterations 20

# Very large batch (might OOM!)
python3 micro_benchmarking_pytorch.py --network resnet50 --batch-size 256 --iterations 20
```

Create a quick comparison table:

Batch Size	Throughput (samples/sec)	Memory (MB)	Samples/sec per GB
8	?	?	?
32	516.1	4523	0.114
128	?	?	?
256	OOM or ?	?	?

What do you observe?

- Throughput increases with batch size... but not linearly
- Memory increases with batch size
- There's a sweet spot for efficiency

Exercise: Precision Comparison (FP32 vs FP16)

Objective

Compare FP32 (32-bit floating point) vs FP16 (16-bit floating point) precision.

What you'll learn:

- What FP16 is and why it matters
- Performance benefits of reduced precision
- Memory savings from FP16
- When to use FP16 vs FP32

What is FP16?

Floating Point Precision:

```
FP32 (Float32):  32 bits = 1 sign + 8 exponent + 23 mantissa
                 Range: ±1.4 × 10⁻⁴⁵ to ±3.4 × 10³⁸
                 Precision: ~7 decimal digits

FP16 (Float16):  16 bits = 1 sign + 5 exponent + 10 mantissa
                 Range: ±6.0 × 10⁻⁸ to ±6.5 × 10⁴
                 Precision: ~3 decimal digits
```

Analyzing the Results

Let's compare FP32 vs FP16:

Create a comparison table:

Metric	FP32	FP16	Improvement
Throughput (samp/s)	516.1	1032.3	2.00x faster
Memory (MB)	4523	2834	37% less
Time per batch (ms)	62.0	31.0	2.00x faster
Numerical accuracy	Full	Reduced	-

Exercise: Precision Comparison (FP32 vs FP16)

Benefits of FP16:
- 2x less memory (16 bits vs 32 bits)
- 2x more data per memory transaction
- 2-4x faster compute (specialized hardware)
- Lower power consumption

Drawbacks of FP16:
- Lower precision (can cause numerical issues)
- Smaller range (risk of overflow/underflow)
- Requires careful model design

For inference: FP16 is usually safe and recommended!

Running FP32 Baseline (Repeat)

First, let's re-run FP32 to have a fresh comparison:

```
python3 micro_benchmarking_pytorch.py \
    --network resnet50 \
    --batch-size 32 \
    --iterations 20 \
    --fp16 0
```

Running FP16 Benchmark

Now let's run with FP16:

```
python3 micro_benchmarking_pytorch.py \
    --network resnet50 \
    --batch-size 32 \
    --iterations 20 \
    --fp16 1
```

Exercise: Precision Comparison (FP32 vs FP16)

Why is it faster?

- **Less Memory Traffic:**
  - FP16 tensor: half the size
  - Loading weights from memory: 2x faster
  - Writing activations: 2x faster
- **Specialized Hardware:**
  - AMD MI200/MI300: Matrix Core FP16 instructions
  - 2-4x higher TFLOPS for FP16 vs FP32
- **Cache Efficiency:**
  - More data fits in L2 cache
  - Fewer cache misses

Exercise 3: Combining precision and profiling PyTorch Profiler Integration

Running with PyTorch Profiler

Let's modify our benchmark to use the profiler.

```
python3 micro_benchmarking_pytorch.py \
    --network resnet50 \
    --batch-size 32 \
    --iterations 10 \
    --fp16 0 \
    --autograd-profiler
```

Comparing FP32 vs FP16 Profiling

Let's profile both precisions:

```
# FP32
python3 micro_benchmarking_pytorch.py --network
resnet50 --batch-size 32 --iterations 10 --fp16
0 --autograd-profiler > profile_fp32.txt

# FP16
python3 micro_benchmarking_pytorch.py --network
resnet50 --batch-size 32 --iterations 10 --fp16
1 --autograd-profiler > profile_fp16.txt
```

Understanding the output

You'll see LOTS of output! Let's focus on key sections:

```
========================================
PyTorch Profiler Results:
========================================

Top 10 operations by total CPU time:
--------------------------  ------------  ------------  ------------
Name                        Self CPU %    Self CPU      CPU total
--------------------------  ------------  ------------  ------------
aten::convolution           5.23%         128.45ms      8.52s
aten::batch_norm            2.15%         52.75ms       1.32s
aten::relu_                 1.87%         45.91ms       45.91ms
aten::max_pool2d            0.95%         23.32ms       67.45ms
aten::addmm                 0.78%         19.15ms       234.67ms
aten::linear                0.65%         15.95ms       250.62ms
aten::add_                  0.52%         12.78ms       12.78ms
aten::_convolution          4.87%         119.55ms      8.40s
aten::cudnn_convolution     78.23%        1.92s         1.92s
...
--------------------------  ------------  ------------  ------------

Top 10 operations by total CUDA time:
--------------------------  ------------  ------------  ------------
Name                        Self CUDA     CUDA total    # of Calls
--------------------------  ------------  ------------  ------------
void cudnn::detail::implicit_convolve_sgemm...   1.82s    1.82s      320
void cudnn::bn_fw_tr_1C11...                     234.56ms 234.56ms   160
Memcpy HtoD (Pageable -> Device)                 145.32ms 145.32ms   50
void at::native::vectorized_elementwise...       89.45ms  89.45ms    640
void cudnn::ops::nchwToNhwc...                   67.23ms  67.23ms    160
...
--------------------------  ------------  ------------  ------------

Memory Profiling:
--------------------------  ------------  ------------  ------------
Name                        CPU Mem       CUDA Mem      # of Calls
--------------------------  ------------  ------------  ------------
aten::convolution           0 b           3.52 Gb       320
aten::batch_norm            0 b           834.56 Mb     160
aten::relu_                 0 b           0 b           160
aten::max_pool2d            0 b           256.00 Mb     32
...
--------------------------  ------------  ------------  ------------
```

Exercise: DeepSpeed FLOPS Profiler

Objective

Measure computational efficiency using DeepSpeed FLOPS Profiler.

What you'll learn:

- What FLOPs are and why they matter
- Theoretical vs achieved FLOPS
- Computational efficiency
- Identifying compute vs memory-bound operations

What are FLOPs?

FLOPS = Floating Point Operations Per Second

Key concepts:

- **Operation Count:**
  - Total floating-point operations in your model
  - Example: Matrix multiply (M×K) × (K×N) = 2×M×K×N FLOPs
- **Theoretical Peak:**
  - Maximum FLOPs your hardware can achieve
  - MI325X: ~653 TFLOPS (FP16), ~326 TFLOPS (FP32)
- **Achieved FLOPs:**
  - What your model actually achieves
  - Usually much lower than peak!
- **Efficiency:**
  - (Achieved / Theoretical) × 100%
  - 50%+ is very good!
  - 10-20% is typical for many workloads

Exercise: DeepSpeed FLOPS Profiler

Why Measure FLOPs?

FLOPs efficiency tells you:

- Are you **compute-bound** or **memory-bound**?
  - High efficiency (>40%): Compute-bound (good!)
  - Low efficiency (<20%): Memory-bound (need optimization!)
- How much headroom for optimization?
  - At 10% efficiency: 10x speedup possible!
  - At 80% efficiency: Already well-optimized
- Hardware utilization:
  - Are you getting value from your expensive GPU?

Step 2: DeepSpeed FLOPSProfiler Integration

- **Benchmark Integration**
- **Micro-benchmark Usage**
  - **Key Metrics**
    - Total FLOPS: Theoretical peak compute
    - Achieved FLOPS: Measured throughput
    - Efficiency: (Achieved / Theoretical) × 100%
    - Bottleneck identification: Compute vs. memory bound

```
from deepspeed.profiling.flops_profiler import FlopsProfiler

prof = FlopsProfiler(model)
prof.start_profile()
outputs = model(inputs)
prof.stop_profile()
prof.print_model_profile(profile_step=1)
```

- To profile with DeepSpeed FLOPSProfiler

```
python micro_benchmarking_pytorch.py \
  --network resnet50 \
  --amp-opt-level=2 \
  --batch-size=256 \
  --flops-prof-step 10
```

https://www.deepspeed.ai/tutorials/flops-profiler/

Exercise: DeepSpeed FLOPS Profiler

Understanding Compute vs Memory Bound

```
Compute-bound:
- Lots of arithmetic operations
- GPU cores fully utilized
- Examples: Matrix multiply, convolutions with
large kernels
- Optimization: Use faster compute (FP16, Tensor
Cores)

Memory-bound:
- Lots of memory reads/writes
- Memory bandwidth saturated
- Examples: Element-wise operations, small
convolutions, attention
- Optimization: Reduce memory traffic (fusion,
better layouts)
```

Install DeepSpeed

```
# Install DeepSpeed
pip install deepspeed
```

Run with FLOPS profiler

```
python3 micro_benchmarking_pytorch.py \
    --network resnet50 \
    --batch-size 32 \
    --iterations 20 \
    --fp16 0 \
    --flops-prof-step 10
```

Note: `--flops-prof-step 10` means profile at iteration 10 (after warmup)

Workshop Summary & Key Takeaways

Core Workshop Philosophy: The Optimization Cycle

- **Measure** → **Analyze** → **Optimize** → **Validate** (then repeat)
- Never optimize blindly - always profile first
- Validation ensures correctness and confirms improvements

Optimization Techniques Demonstrated

- **Precision optimization**: FP16 provides ~2x speedup + 37% memory reduction with minimal accuracy impact
- **Batch size tuning**: Balance between throughput and memory - find the sweet spot
- **torch.compile**: Nearly free performance gains through graph optimization
- **Multi-GPU scaling**: Use `torchrun` for distributed inference

Key Findings from Exercises

- Modern AI workloads typically run at **10-20% of theoretical hardware performance**
- First iteration is always slow (kernel compilation, memory allocation, cache warming)
- FP16 delivers **2-4x faster compute** due to specialized Matrix Core hardware
- **Convolution operations** dominate ResNet50 execution time (~80% of GPU time)
- Memory-bound operations need **fusion and layout optimization**, not faster compute

Common Pitfalls to Avoid

- ❌ Optimizing without profiling (wasted effort)
- ❌ Ignoring numerical accuracy validation
- ❌ Over-optimizing operations that contribute <5% to total runtime
- ❌ Hardware-specific tuning before algorithmic optimization

Actionable Next Steps

- **Establish baseline**: Profile your model with PyTorch Profiler + DeepSpeed
- **Identify bottlenecks**: Check if compute-bound or memory-bound
- **Apply low-hanging fruit**: Enable FP16, torch.compile, optimal batch size
- **Target hotspots**: Focus optimization on operations taking >10% of runtime
- **Validate everything**: Compare outputs, measure improvements, document results
- **Iterate**: Re-profile after each change to confirm improvements

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED “AS IS” WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.

LLVM is a trademark of LLVM Foundation
Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries
OpenCL is a trademark of Apple Inc. used by permission by Khronos Group, Inc.
Intel® is a trademark of Intel Corporation or its subsidiaries
The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board

AMD