Tiny LLaMA: Architecture, Setup, and Baseline Profiling

Presenter: Bob Robey
Oct 13-16, 2025
AMD @ CASTIEL AI Workshop

AMD
together we advance_

Tiny LLaMA example

Overview

- Tiny LLaMA is a scaled-down implementation of the LLaMA (Large Language Model Meta AI) architecture, designed specifically for educational purposes and performance profiling workshops. This document provides comprehensive technical details of the model architecture, mathematical formulations, and implementation patterns used throughout the workshop versions.

Workshop exercises key design principles

- Educational Focus: Simplified scale for understanding optimization techniques
- Performance Profiling: Designed to showcase bottlenecks and optimization opportunities
- Progressive Optimization: Architecture supports incremental improvements across workshop versions
- Hardware Awareness: Implementation patterns optimized for AMD ROCm and modern GPU architectures

Through these exercises, we will look at programming optimizations and the various profiling tools that can be used on Pytorch applications

Tiny LLaMA Architecture Overview

Scaled-Down LLaMA for Educational Focus

- Small configuration below for rapid iteration
- Representative of production LLM architectures
- Exposes optimization bottlenecks clearly
- Hardware-aware implementation patterns

Config	Speedup
Small (BS= 8, S=128)	5.8x
Medium (BS=16, S=512)	3.14x

Results for our testing of the Small configuration on an AMD Instinct GPU for all optimizations. You may be on a different GPU model. What do you get?

Mathematical Foundations

The Tiny LLaMA model uses a standard transformer decoder architecture with the mathematical formulation:

Transformer Architecture Core Equations

- Input Embedding and Positional Encoding encodes the text as a series of tokens
- RMSNorm (Root Mean Square Normalization) instead of LayerNorm for improved numerical stability
- Rotary Position Embeddings (RoPE)

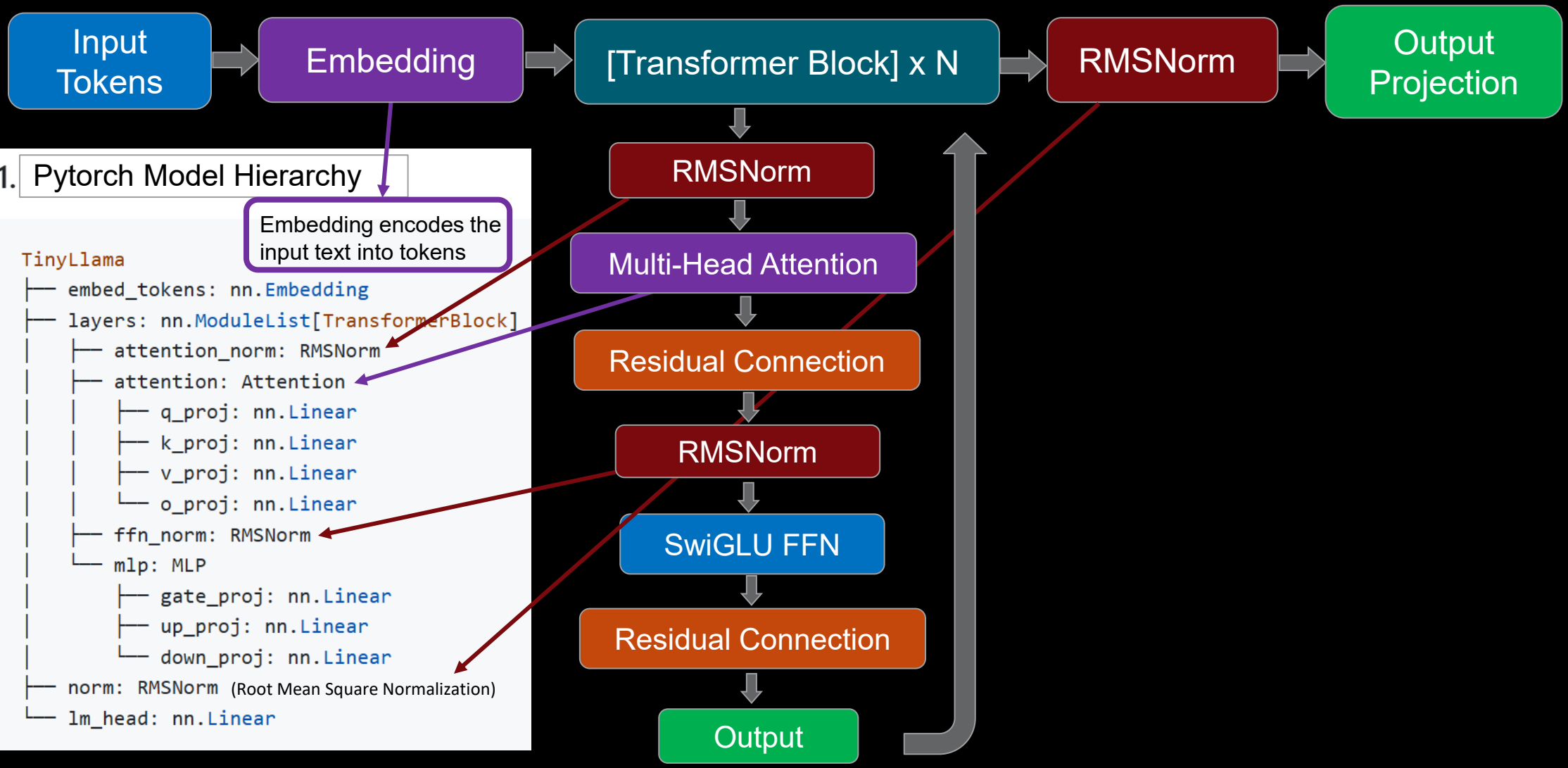
Multi-Head Attention Mechanism

- Includes Query, Key, Value, and Output projects
- Causal Attention Mask to prevent looking ahead

SwiGLU Feed-Forward Network (FFN)

- Combination of Swish and GLU – hence its name
- GLU – Gated Linear Units is a sigmoid activation function
- Swish is a type of a sigmoid function
- Improved properties over standard ReLU

Model Architecture



1. Pytorch Model Hierarchy

Embedding encodes the input text into tokens

```
TinyLlama
├── embed_tokens: nn.Embedding
├── layers: nn.ModuleList[TransformerBlock]
│   ├── attention_norm: RMSNorm
│   ├── attention: Attention
│   │   ├── q_proj: nn.Linear
│   │   ├── k_proj: nn.Linear
│   │   ├── v_proj: nn.Linear
│   │   └── o_proj: nn.Linear
│   ├── ffn_norm: RMSNorm
│   └── mlp: MLP
│       ├── gate_proj: nn.Linear
│       ├── up_proj: nn.Linear
│       └── down_proj: nn.Linear
├── norm: RMSNorm (Root Mean Square Normalization)
└── lm_head: nn.Linear
```

Computational Complexity

1. Attention Complexity

- **Standard Attention:**
  - Time: O(S2·d) per layer
  - Memory: O(S2) for attention matrix storage
- **With Sequence Length S=128:**
  - Attention matrix: 128×128=16,384 elements per head
  - Total attention matrices: 8×16,384=131,072 elements per layer

2. Feed-Forward Networks (FFN) Complexity

- **Per Layer:**
  - Time: O(Sxdxdff)
  - Parameters: 2×d×dff+dff×d=3×d×dff
- **For Tiny LLaMA:**
  - FFN operations: S×d×dff=128×256×512=16,777,216 operations per layer

3. Total Model Complexity

- **FLOP Count per Forward Pass:**
  - Embedding:B×S×d
  - Attention:L×B×S×(4×S×d+S2×h)
  - FFN:L×B×S×3×d×dff
  - LayerNorm:L×B×S×d×2
  - Total≈L×B×S×(S×d×h+3×d×dff)
- **For Default Configuration:**
  - Attention: 4×128×128×256×8=134,217,728 FLOPs per layer
  - FFN: 4×128×3×256×512=201,326,592 FLOPs per layer
  - **Total per forward pass**: ~**1.34 GFLOPs** (batch_size=1)

We can calculate the Computational Complexity! Don't worry about the details - just the final number. See the Hands-on Exercises if you want to calculate yourself.

We can also calculate thePerformance Characteristics!

1. Memory Bandwidth Requirements

- **Parameter Access:**
  - Model parameters: ~2.8M parameters × 4 bytes = 11.2 MB
  - Bandwidth requirement: **11.2** MB per forward pass
- **Activation Memory:**
  - Peak activation memory: **~50 MB** (batch_size=8, seq_len=128)
  - Memory bandwidth utilization: Critical for performance

```
Embedding: V × d = 1000 × 256 = 256K params
Attention: 4L × d × d = 4 × 4 × 256 × 256 = 1.05M params
FFN: L × (2d × d_ff + d_ff × d) = 4 × 3 × 256 × 512
   = 1.57M params
LayerNorm: (2L + 1) × d = 9 × 256 = 2.3K params
Total: ~2.8M parameters (~11.2 MB in FP32)
```

```
Attention: B × S × d + B × h × S × S = 8×128×256 + 8×8×128×128
     = 1.3M elements
FFN: B × S × d_ff = 8 × 128 × 512 = 524K elements
Peak: ~50 MB (batch_size=8, seq_len=128)
```

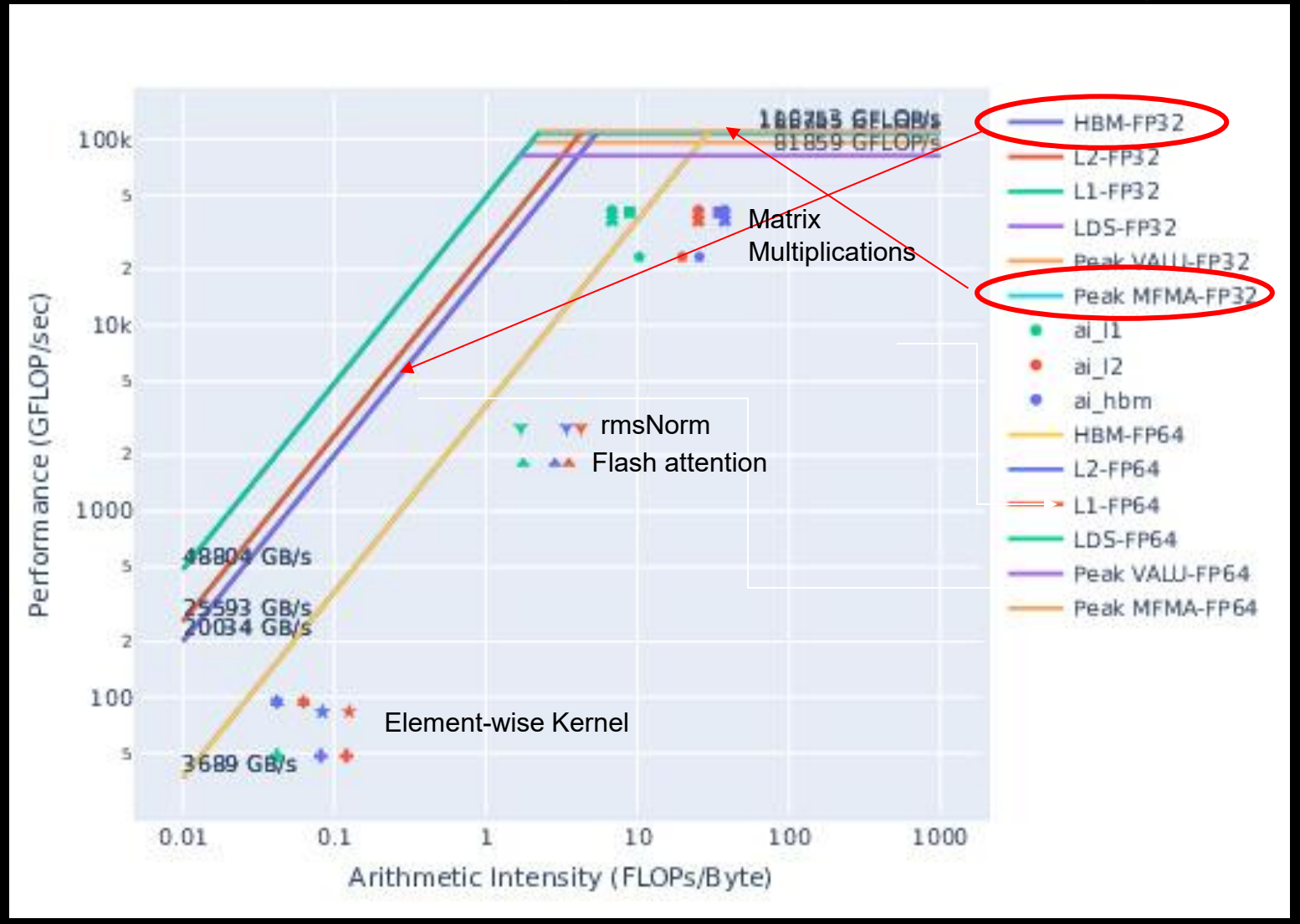
2. Arithmetic Intensity

- **For Tiny LLaMA:**
  - FLOPs per forward pass: ~1.34 × 10^9
  - Memory accessed: **~61.2** MB
- **Arithmetic Intensity**: ~21.9 FLOPs/byte

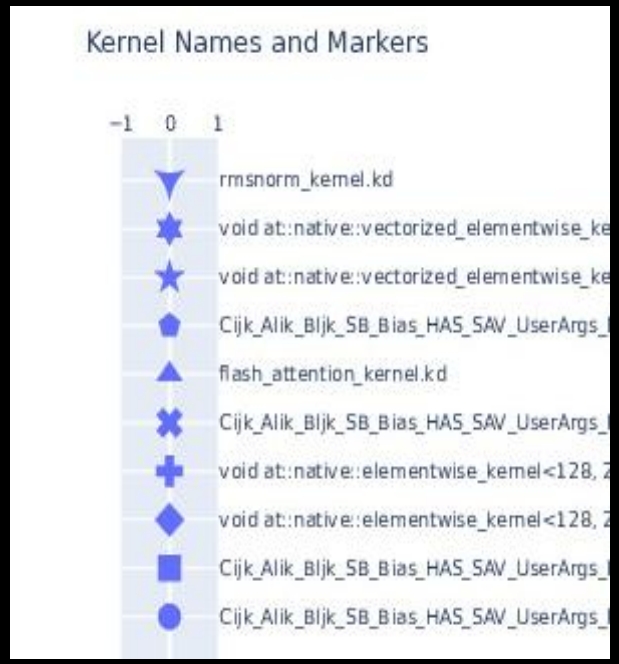
From previous slide

```
Arithmetic Intensity = FLOPs / Bytes Accessed
   = 1.34 × 10⁹ / 61.2 MB
   = 21.9 FLOPs/byte
```

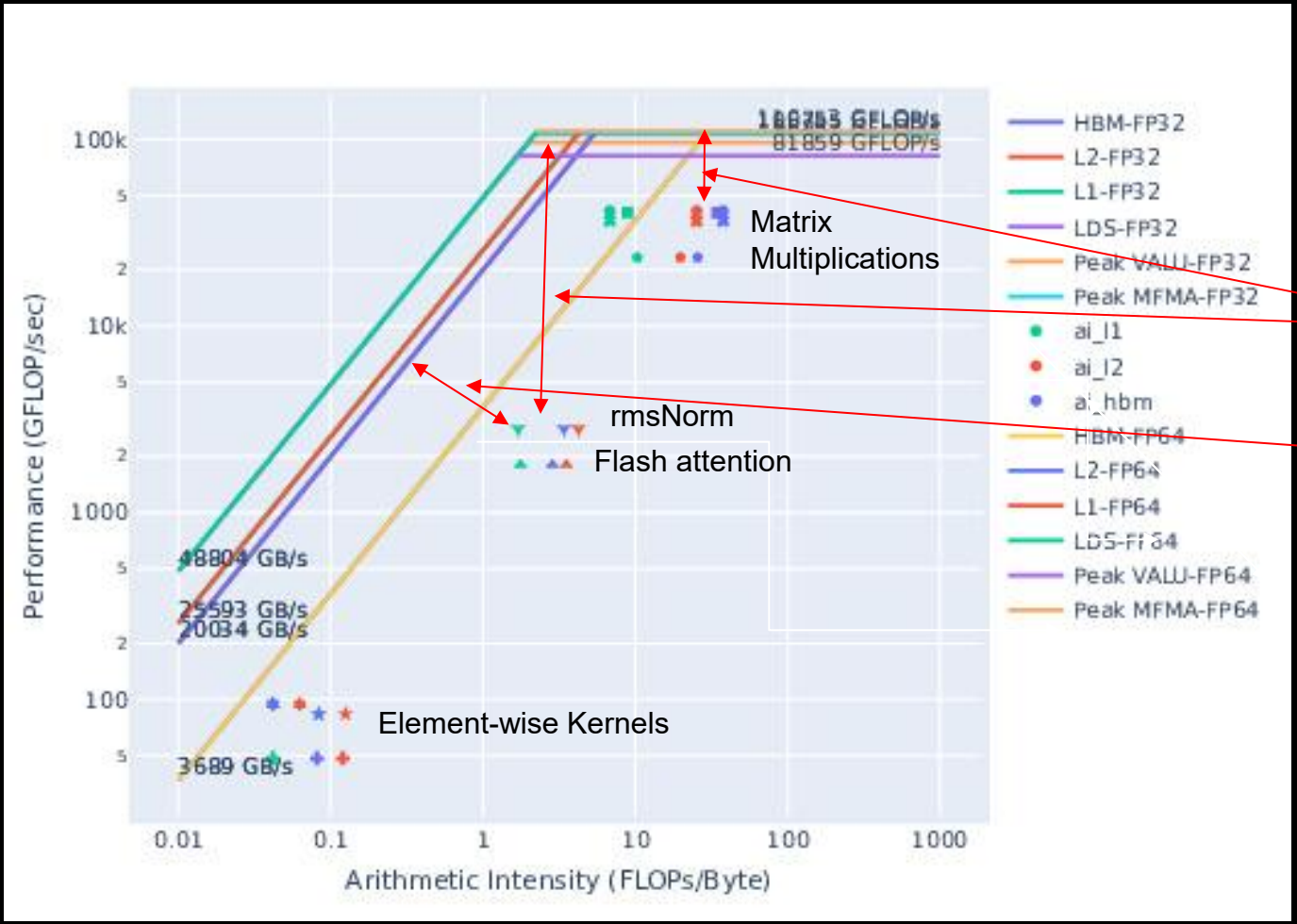
Arithmetic Intensity and Roofline Analysis



Roofline from rocprof-compute



Arithmetic Intensity and Roofline Analysis



- **Compute vs Memory Characteristics**
  - Compute-bound on modern GPUs (good for optimization)
    - Close to the Peak MFMA-FP32 for Matrix Multiplies but still room for improvement.
  - Benefits significantly from kernel fusion
  - Memory layout optimization crucial
    - Still some gap between HBM-FP32 and operations
  - High Arithmetic Intensity means optimization potential
  - MI200 series: Peak ~45 TFLOPS FP32, ~1.6 TB/s bandwidth
  - Theoretical max: min(45 TFLOPS, 21.9 × 1600 GB/s) = 35 TFLOPS
  - Baseline achieves ~2-3% of theoretical flops roofline (huge optimization gap!)

Performance Implications and Optimization Opportunities

Performance Implications:

- Compute-bound on modern GPUs (good for optimization)
- Benefits significantly from kernel fusion
- Memory layout optimization crucial for performance

Optimization Opportunities

- **Identified Bottlenecks:**
  - **Attention Memory**: O(S2) memory scaling
  - **Kernel Launch Overhead**: Multiple small operations
  - **Memory Bandwidth**: Activation tensor transfers
  - **Load Imbalance**: Variable sequence lengths

Optimization Strategies:

- **Kernel Fusion**: Combine multiple operations
- **Flash Attention**: Reduce memory complexity
- **Custom Kernels**: Triton implementations
- **Memory Layout**: Optimize tensor arrangements

References and Further Reading (and viewing)

Note that the references are all very recent. It shows how much theoretical development is happening

- **LLaMA Architecture**: "LLaMA: Open and Efficient Foundation Language Models" (Touvron et al., 2023)
- **RMSNorm**: "Root Mean Square Layer Normalization" (Zhang & Sennrich, 2019)
- **RoPE**: "RoFormer: Enhanced Transformer with Rotary Position Embedding" (Su et al., 2021)
- **SwiGLU**: "GLU Variants Improve Transformer" (Shazeer, 2020)
- **Flash Attention**: "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness" (Dao et al., 2022)
- **Grouped-Query Attention**: "GQA: Training Generalized Multi-Query Transformer Models" (Ainslie et al., 2023)

- Good videos on Neural Networks on Youtube at 3Blue1Brown channel - https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&si=_Hrhz_jYup6V3j0c, especially the ones on Large Language Models

Workshop Structure and Timeline

Session	Memory	Compared to Baseline
Session 1: Foundation and benchmarking Transformer architecture deep dive Computational complexity analysis PyTorch Profiler and DeepSpeed FLOPSProfiler	522 MB This Presentation!!!	Baseline
Session 2: Kernel fusion and ROCm tools QKV fusion and Flash Attention SwiGLU fusion techniques ROCm profiling tools (rocprof-sys, rocprof-compute)	~450 MB	1.7x
Session 3: Custom GPU Kernels Triton language fundamentals Custom RMSNorm, SwiGLU, Flash Attention kernels Memory hierarchy optimization	282 MB	5.5x
Session 4: Production Optimization PyTorch Scaled Dot-Product Attention Backend selection (Flash Attention 2) Production deployment strategies	270 MB	5.8x

Future Presentations

Model Parameters

Parameter	Value	Description
vocab_size	1000	Vocabulary (reduced for workshop)
hidden_dim	256	Model dimension
n_layers	4	Transformer blocks
n_heads	8	Attention heads
n_kv_heads	4	KV heads (GQA)
intermediate_dim	512	FFN dimension
max_seq_len	128	Context window

Exercise 1 – Step 1 Run the basic training

- Get the examples

```
git clone https://github.com/AMD/HPCTrainingExamples
cd HPCTrainingExamples/MLExamples/TinyTransformer/version1_pytorch_baseline
```

- Get a node and set up the environment

```
salloc -N 1 -n 4 --gpus=4
module load rocm pytorch
```

- First, let's get the baseline performance

```
mkdir pytorch_profiles
python3 tiny_llama_v1.py --batch-size 8 --seq-len 128 --num-steps 20
```

- Free the allocation

```
exit
```

HPCTrainingExamples/MLExamples/TinyTransformer/version1_pytorch_baseline/exercises/exercise_1_baseline_analysis.md

Sample output from basic training run

```
Running 5 warmup steps to eliminate compilation overhead...
Warmup complete. Starting measured training loop...
============================================================
Step   0/20 | Loss: 7.0185 | Speed: 335.3 samples/sec | Memory:  370.2 MB | Time:  23.9ms
Step  10/20 | Loss: 6.9978 | Speed: 278.9 samples/sec | Memory:  370.2 MB | Time:  28.7ms
============================================================

Performance Summary:
   Total samples processed: 160
   Average training speed: 296.3 samples/xsec
   Throughput: 37923 tokens/sec
   Average batch time: 27.2 ms
   Average forward time: 12.4 ms
   Average backward time: 13.2 ms
   Average optimizer time: 1.6 ms
   Final loss: 6.9556
   Peak memory usage: 370.2 MB

Performance data saved to: pytorch_profiles/performance_summary.json
```

Exercise 1 – Step 2: Run with the PyTorch Profiler

- Get a node and set up the environment

```
salloc -N 1 -n 4 --gpus=4
module load rocm pytorch
```

- First, let's get the baseline performance

```
mkdir pytorch_profiles
python3 tiny_llama_v1.py --batch-size 8 --seq-len 128 --num-steps 20 \
    --enable-pytorch-profiler --profile-dir ./exercise1_profiles
```

- Free the allocation

```
exit
```

Sample output from pytorch profiler run

```
Running 5 warmup steps to eliminate compilation overhead...
Warmup complete. Starting measured training loop...
============================================================
Step   0/20 | Loss: 7.0185 | Speed: 232.4 samples/sec | Memory:  370.2 MB | Time:  34.4ms
[W1013 17:23:11.849665928 collection.cpp:1110] Warning: ROCTracer produced duplicate flow
start: 2338 (function operator())
Step  10/20 | Loss: 6.9978 | Speed: 223.6 samples/sec | Memory:  370.2 MB | Time:  35.8ms
============================================================

Performance Summary:
   Total samples processed: 160
   Average training speed: 187.7 samples/sec
   Throughput: 24025 tokens/sec
   Average batch time: 46.9 ms
   Average forward time: 20.0 ms
   Average backward time: 23.4 ms
   Average optimizer time: 3.4 ms
   Final loss: 6.9556
   Peak memory usage: 370.2 MB

Performance data saved to: exercise1_profiles/performance_summary.json
```

Exercise 1 – Step 3: Analyze Profiling Results

- Get a node and set up the environment

```
salloc -N 1 -n 4 --gpus=4
module load rocm pytorch
```

- Launch TensorBoard to visualize the profiling results

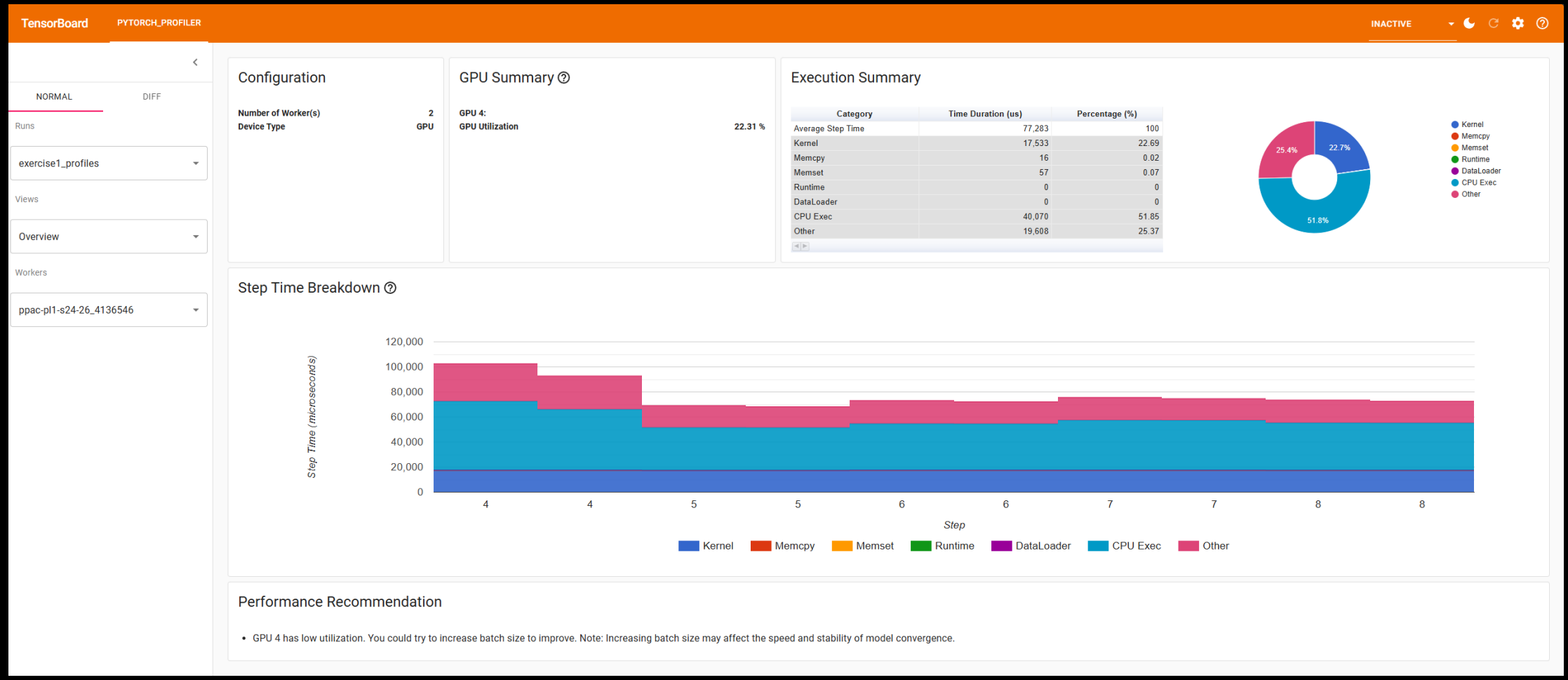
```
tensorboard --logdir ./exercise1_profiles --port 6006 --bind-all &
```

- View TensorBoard results
  1. Set up tunnel to compute node (something like ssh -L 6006:localhost:6006 <server>)
     - only able to access tensorboard running on the login
     - pip install tensorboard and the torch_plugin
  2. Open your browser to `http://localhost:6006`
  3. Navigate to the "PROFILE" tab
  4. Select the most recent run

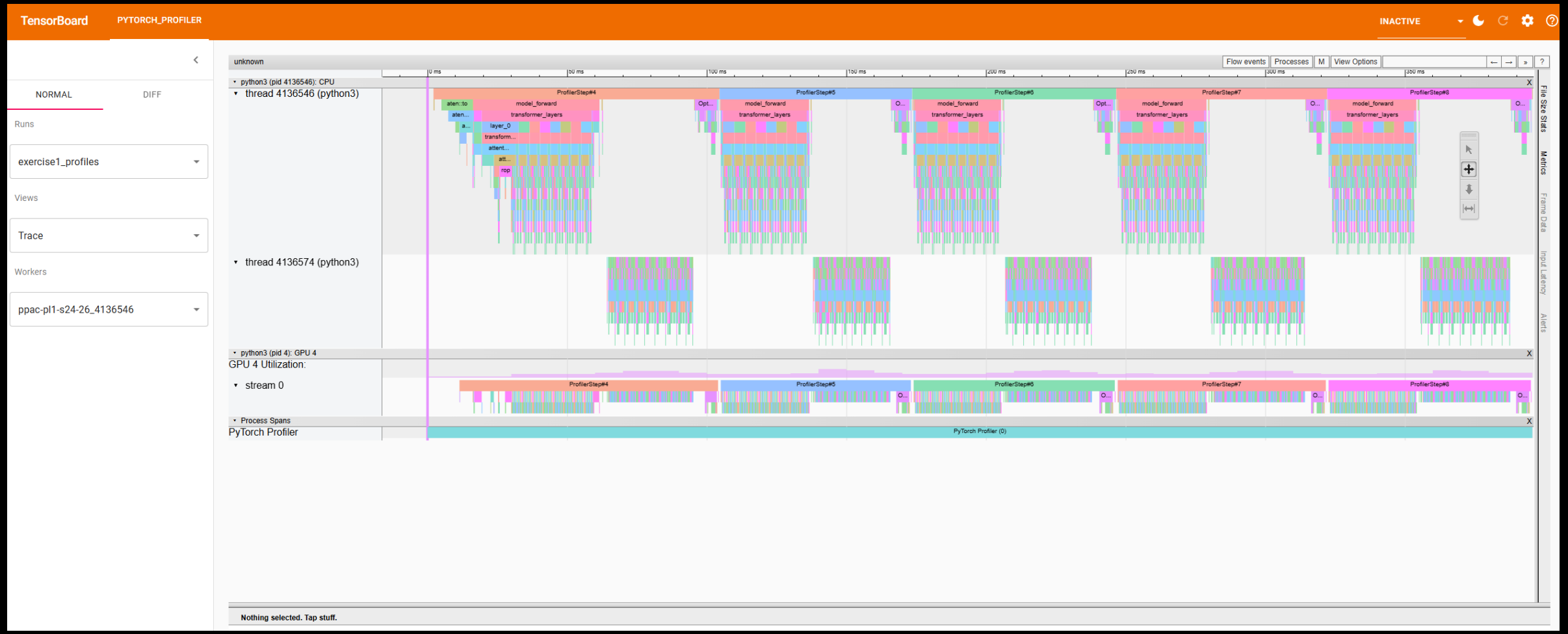
- Free the allocation

```
exit
```

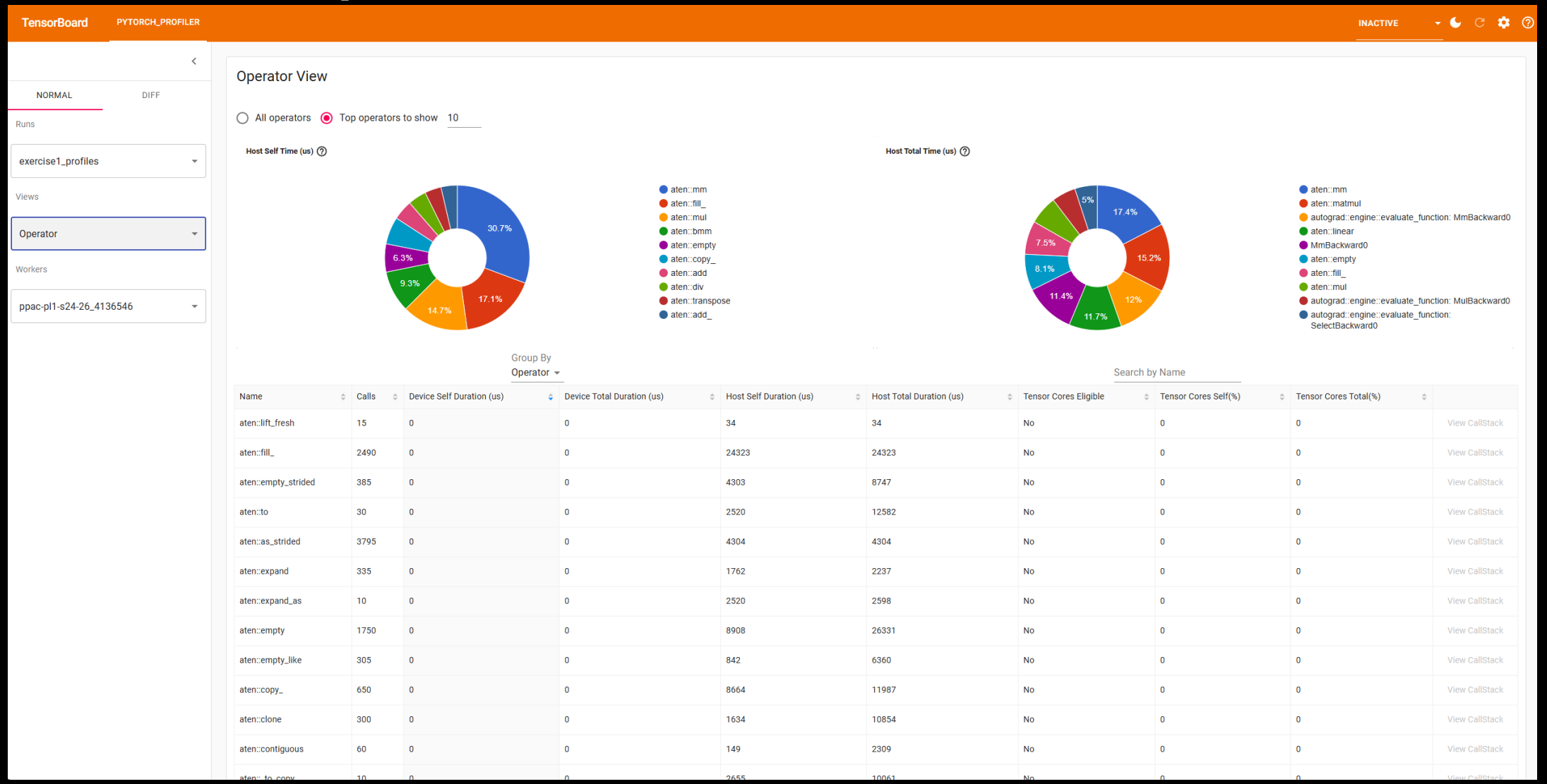
Tensorboard output 1 of 4



Tensorboard output 2 of 4

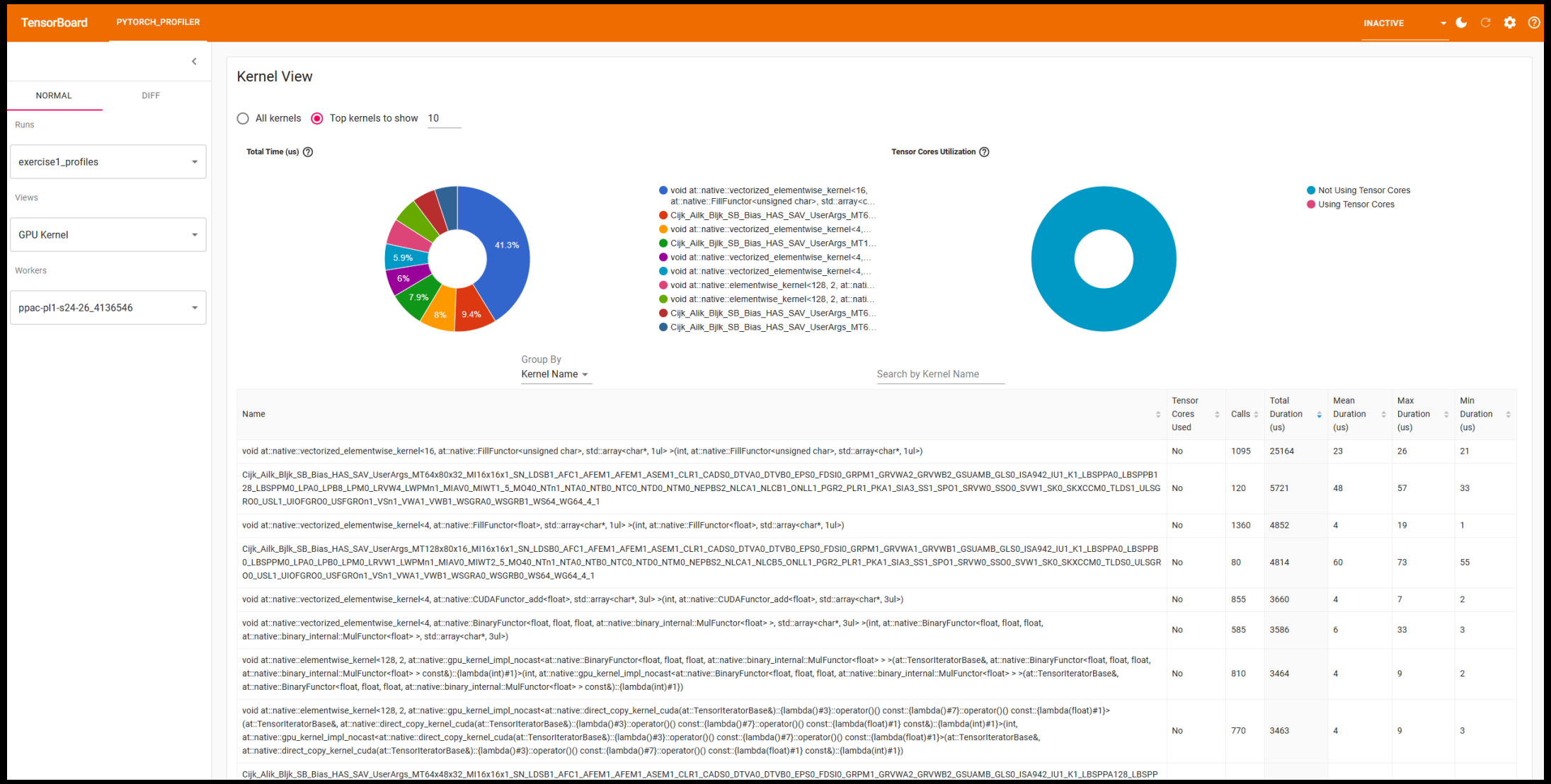


Tensorboard output 3 of 4



Name	Calls	Device Self Duration (us)	Device Total Duration (us)	Host Self Duration (us)	Host Total Duration (us)	Tensor Cores Eligible	Tensor Cores Self(%)	Tensor Cores Total(%)	
aten::lift_fresh	15	0	0	34	34	No	0	0	View CallStack
aten::fill_	2490	0	0	24323	24323	No	0	0	View CallStack
aten::empty_strided	385	0	0	4303	8747	No	0	0	View CallStack
aten::to	30	0	0	2520	12582	No	0	0	View CallStack
aten::as_strided	3795	0	0	4304	4304	No	0	0	View CallStack
aten::expand	335	0	0	1762	2237	No	0	0	View CallStack
aten::expand_as	10	0	0	2520	2598	No	0	0	View CallStack
aten::empty	1750	0	0	8908	26331	No	0	0	View CallStack
aten::empty_like	305	0	0	842	6360	No	0	0	View CallStack
aten::copy_	650	0	0	8664	11987	No	0	0	View CallStack
aten::clone	300	0	0	1634	10854	No	0	0	View CallStack
aten::contiguous	60	0	0	149	2309	No	0	0	View CallStack
aten::_to_copy	10	0	0	2655	10061	No	0	0	View CallStack

Tensorboard output 4 of 4



Name	Tensor Cores Used	Calls	Total Duration (us)	Mean Duration (us)	Max Duration (us)	Min Duration (us)
void at::native::vectorized_elementwise_kernel<16, at::native::FillFunctor<unsigned char>, std::array<char*, 1ul> >(int, at::native::FillFunctor<unsigned char>, std::array<char*, 1ul>)	No	1095	25164	23	26	21
Cijk_Ailk_Bljk_SB_Bias_HAS_SAV_UserArgs_MT64x80x32_MI16x16x1_SN_LDSB1_AFC1_AFEM1_AFEM1_ASEM1_CLR1_CADS0_DTVA0_DTVB0_EPS0_FDSI0_GRPM1_GRVWA2_GRVWB2_GSUAMB_GLS0_ISA942_IU1_K1_LBSPPA0_LBSPPB128_LBSPPM0_LPA0_LPB8_LPM0_LRVW4_LWPMn1_MIAV0_MIWT1_5_MO40_NTn1_NTA0_NTB0_NTC0_NTD0_NTM0_NEPBS2_NLCA1_NLCB1_ONLL1_PGR2_PLR1_PKA1_SIA3_SS1_SPO1_SRVW0_SSO0_SVW1_SK0_SKXCCM0_TLDS1_ULSGRO0_USL1_UIOFGRO0_USFGROn1_VSn1_VWA1_VWB1_WSGRA0_WSGRB1_WS64_WG64_4_1	No	120	5721	48	57	33
void at::native::vectorized_elementwise_kernel<4, at::native::FillFunctor<float>, std::array<char*, 1ul> >(int, at::native::FillFunctor<float>, std::array<char*, 1ul>)	No	1360	4852	4	19	1
Cijk_Ailk_Bljk_SB_Bias_HAS_SAV_UserArgs_MT128x80x16_MI16x16x1_SN_LDSB0_AFC1_AFEM1_AFEM1_ASEM1_CLR1_CADS0_DTVA0_DTVB0_EPS0_FDSI0_GRPM1_GRVWA1_GRVWB1_GSUAMB_GLS0_ISA942_IU1_K1_LBSPPA0_LBSPPB0_LBSPPM0_LPA0_LPB0_LPM0_LRVW1_LWPMn1_MIAV0_MIWT2_5_MO40_NTn1_NTA0_NTB0_NTC0_NTD0_NTM0_NEPBS2_NLCA1_NLCB5_ONLL1_PGR2_PLR1_PKA1_SIA3_SS1_SPO1_SRVW0_SSO0_SVW1_SK0_SKXCCM0_TLDS0_ULSGRO0_USL1_UIOFGRO0_USFGROn1_VSn1_VWA1_VWB1_WSGRA0_WSGRB0_WS64_WG64_4_1	No	80	4814	60	73	55
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add<float>, std::array<char*, 3ul> >(int, at::native::CUDAFunctor_add<float>, std::array<char*, 3ul>)	No	855	3660	4	7	2
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float> >, std::array<char*, 3ul> >(int, at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float> >, std::array<char*, 3ul>)	No	585	3586	6	33	3
void at::native::elementwise_kernel<128, 2, at::native::gpu_kernel_impl_nocast<at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float> > >(at::TensorIteratorBase&, at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float> > const&)::{lambda(int)#1}>(int, at::native::gpu_kernel_impl_nocast<at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float> > >(at::TensorIteratorBase&, at::native::BinaryFunctor<float, float, float, at::native::binary_internal::MulFunctor<float> > const&)::{lambda(int)#1})	No	810	3464	4	9	2
void at::native::elementwise_kernel<128, 2, at::native::gpu_kernel_impl_nocast<at::native::direct_copy_kernel_cuda(at::TensorIteratorBase&)::{lambda()#3}::operator()() const::{lambda()#7}::operator()() const::{lambda(float)#1}>(at::TensorIteratorBase&, at::native::direct_copy_kernel_cuda(at::TensorIteratorBase&)::{lambda()#3}::operator()() const::{lambda()#7}::operator()() const::{lambda(float)#1} const&)::{lambda(int)#1}>(int, at::native::gpu_kernel_impl_nocast<at::native::direct_copy_kernel_cuda(at::TensorIteratorBase&)::{lambda()#3}::operator()() const::{lambda()#7}::operator()() const::{lambda(float)#1}>(at::TensorIteratorBase&, at::native::direct_copy_kernel_cuda(at::TensorIteratorBase&)::{lambda()#3}::operator()() const::{lambda()#7}::operator()() const::{lambda(float)#1} const&)::{lambda(int)#1})	No	770	3463	4	9	3
Cijk_Alik_Bljk_SB_Bias_HAS_SAV_UserArgs_MT64x48x32_MI16x16x1_SN_LDSB1_AFC1_AFEM1_AFEM1_ASEM1_CLR1_CADS0_DTVA0_DTVB0_EPS0_FDSI0_GRPM1_GRVWA2_GRVWB2_GSUAMB_GLS0_ISA942_IU1_K1_LBSPPA128_LBSPP						

Exercise 2 – Step 1 Memory-Focused Profiling

- Run profiling with enhanced memory analysis:
- Memory profiling with different batch sizes
- Batch size 4

```
python tiny_llama_v1.py \
    --batch-size 4 \
    --seq-len 128 \
    --num-steps 15 \
    --enable-pytorch-profiler \
    --enable-memory-profiling \
    --profile-dir ./memory_analysis_bs4
```

- Repeat for batch size 8 and 16

Exercise 2 – Step 2 Memory Timeline Analysis

- Analyze memory patterns using TensorBoard:
- Launch TensorBoard for memory analysis

```
tensorboard --logdir ./memory_analysis_bs8 --port 6007 --bind-all &
```

- View TensorBoard results
  1. Set up tunnel to compute node (something like ssh –L 6006:localhost:6006 <server>)
  2. Open your browser to `http://localhost:6006`
  3. Navigate to the "PROFILE" tab
  4. Select the most recent run

Exercise 2 – Step 3 Sequence Length Scaling

- Test how memory scales with sequence length:
- Test different sequence lengths

```
python tiny_llama_v1.py \
    --batch-size 8 \
    --seq-len 64 \
    --num-steps 10 \
    --enable-memory-profiling \
    --profile-dir ./memory_seq64
```

- Repeat for sequence length 256 and 512

Exercise 2 – Step 4: Memory Bandwidth Analysis

- Use the memory profiling results to analyze bandwidth utilization:
- Run bandwidth-focused analysis

```
python run_deepspeed_flops.py \
    --batch-size 8 \
    --seq-len 128 \
    --num-steps 15 \
    --computational-intensity \
    --output-dir ./bandwidth_analysis
```

- Check the `bandwidth_analysis/computational_intensity.json` file:
- View bandwidth metrics

```
python -c "
import json
data = json.load(open('./bandwidth_analysis/computational_intensity.json'))
print('Arithmetic Intensity:', data['arithmetic_intensity_flops_per_byte'])
print('Memory Bandwidth Used:', data['memory_bandwidth_used_gb_per_sec'], 'GB/s')
print('Bandwidth Utilization:', data['memory_bandwidth_utilization_percent'], '%')
print('Workload Type:', data['memory_bound_vs_compute_bound’])
“
```

Exercise 3 – Step 1: Comprehensive Profiling Run

- Run the complete profiling suite to gather all necessary data:
- Run comprehensive profiling analysis:

```
bash run_all_profilers.sh \
    --batch-size 8 \
    --seq-len 128 \
    --num-steps 30 \
    --profile-dir ./bottleneck_analysis
```

Exercise 3 – Step 2: Operator-Level Bottleneck Analysis

- Analyze the detailed profiling results to identify computational bottlenecks:
- View the comprehensive profiling report:

```
cat ./bottleneck_analysis/performance_summary_report.md
```

- Examine PyTorch profiler operator breakdown

```
python run_pytorch_profiler.py \
    --analyze-existing ./bottleneck_analysis/pytorch_profiling \
    --generate-report \
    --output-dir ./detailed_analysis
```

Exercise 3 – Step 3: FLOPS Efficiency Analysis

- Examine computational efficiency using the FLOPS analysis:
- View FLOPS analysis results:

```
python -c "
import json
with open('./bottleneck_analysis/flops_analysis/flops_profile.json', 'r') as f:
    data = json.load(f)

print('=== FLOPS EFFICIENCY ANALYSIS ===')
print(f'Model FLOPS Utilization: {data[\"efficiency_metrics\"][\"mfu_percent\"]:.1f}%')
print(f'Achieved FLOPS/sec: {data[\"performance_metrics\"][\"flops_per_sec\"]:.2e}')
print(f'Peak Device FLOPS: {data[\"efficiency_metrics\"][\"device_peak_flops\"]:.2e}')
print(f'FLOPS per Parameter: {data[\"flops_analysis\"][\"flops_per_parameter\"]:.2f}')
print(f'Throughput: {data[\"performance_metrics\"][\"throughput_samples_per_sec\"]:.1f} samples/sec')
"
```

Exercise 3 – Step 4: Memory Bottleneck Assessment

- Analyze memory-related bottlenecks:
- Check computational intensity analysis:

```
python -c "
import json
import os

intensity_file = './bottleneck_analysis/flops_analysis/computational_intensity.json'
if os.path.exists(intensity_file):
    with open(intensity_file, 'r') as f:
        data = json.load(f)

    print('=== MEMORY BOTTLENECK ANALYSIS ===')
    print(f'Arithmetic Intensity: {data[\"arithmetic_intensity_flops_per_byte\"]:.2f} FLOPS/byte')
    print(f'Memory Bandwidth Used: {data[\"memory_bandwidth_used_gb_per_sec\"]:.1f} GB/s')
    print(f'Bandwidth Utilization: {data[\"memory_bandwidth_utilization_percent\"]:.1f}%')
    print(f'Workload Type: {data[\"memory_bound_vs_compute_bound\"]}')
else:
    print('Computational intensity analysis not available')
"
```

Troubleshooting Common Issues

- **Issue 1: No HIP GPUs Available**
- **Issue 2: Triton Import Error**
- **Issue 3: CUDA Out of Memory**

```
pip install --upgrade triton
# Or from source
pip install git+https://github.com/openai/triton.git@main
```

```
# Reduce batch size
python tiny_llama_v3.py --batch-size 4 --seq-len 64 --num-steps 10

# Clear cache
python3 -c "import torch; torch.cuda.empty_cache()"
```

Next Steps

Version 1: PyTorch Baseline - Profiling Foundation

- **Purpose and Implementation Characteristics**
- Establish baseline performance metrics
- Comprehensive profiling (PyTorch Profiler + DeepSpeed FLOPSProfiler)
- Identify optimization targets
- Reproducible measurement methodology

```
# Attention: Separate projections (3 kernel launches)
query = self.q_proj(hidden_states)   # Launch 1
key = self.k_proj(hidden_states)     # Launch 2
value = self.v_proj(hidden_states)   # Launch 3

# FFN: Separate projections + sequential ops (5 kernel launches)
gate = self.gate_proj(hidden_states)  # Launch 1
up = self.up_proj(hidden_states)      # Launch 2
gate_activated = F.silu(gate)         # Launch 3
intermediate = gate_activated * up     # Launch 4
output = self.down_proj(intermediate)  # Launch 5
```

Optimization Opportunity Identification

- **Bottleneck Analysis with Optimization Targets**
- Kernel launches: 100+ per forward pass (15-20% overhead)
- Attention memory: 131K elements/layer × 4 = 524K total
- Small ops: High call count, low utilization
- V2: Fuse projections | V3: Custom Triton kernels + Flash Attention | V4: SDPA + ultra-fusion

```
Attention Bottlenecks                    FFN Bottlenecks
├── 3 separate Q/K/V projections    →    ├── Separate gate/up projections  →
│   kernel launch overhead               │   fusable into single GEMM
├── Attention matrix O(S²) memory   →    ├── Intermediate tensor storage   →
│   Flash Attention approach             │   memory overhead reduction
├── Multiple tensor reshapes        →    ├── Sequential SiLU + multiply    →
│   memory layout inefficiency           │   fusable activation
└── Sequential operations           →    └── Element-wise ops              →
    limited parallelization                  kernel launch overhead

LayerNorm/RMSNorm Bottlenecks
└── Small tensor operations → fuse with adjacent linear layers
```

Performance Optimization Methodology

- **Systematic Workflow**

```
1. Profile Baseline
   ├── Quantitative bottleneck identification
   └── Operator-level time breakdown

2. Prioritize Targets
   ├── 80/20 rule: Focus high-impact operations
   └── Validate with profiler data

3. Apply Optimization
   ├── Kernel fusion (V2)
   ├── Custom kernels (V3)
   └── Advanced fusion (V4)

4. Measure Impact
   ├── Speedup validation
   ├── Memory efficiency
   └── Correctness verification (loss check)

5. Iterate
   └── Profile again, identify remaining bottlenecks
```

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.


AMD