Profiling & Optimization

Week 13 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra
IIT Gandhinagar

The Performance Problem

Training is expensive:

- GPT-3 cost ~$4.6M to train
- LLaMA-65B: ~$2-3M in compute
- Even small models can burn through credits

Inference at scale is costly:

- ChatGPT serves millions of requests/day
- 100ms latency improvement = $1M+ savings/year

Developer time is expensive:

- Slow iteration cycles reduce productivity
- 10 min/epoch → 100 epochs = 16+ hours waiting

Goal: Make code faster and more efficient without sacrificing accuracy.

The Optimization Mindset

Donald Knuth's wisdom:

> "Premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

The correct process:

1. **Make it work** (correctness first)
2. **Make it right** (clean code, tests)
3. **Profile to find bottlenecks** (measure, don't guess!)
4. **Make it fast** (optimize the 3% that matters)

Common mistake: Optimizing code that runs once during initialization while ignoring the training loop that runs millions of times.

The Doctor's Approach

Profiling is like a doctor's diagnosis. Don't prescribe medicine based on a hunch - run tests first.

```
Programmer: "My code is slow!"
Bad: "Let me rewrite in C++" (guessing)
Good: "Let me profile first" (measuring)
      → Finds: data loading is 70% of time
      → Fix: Add num_workers=4
      → Result: 2x faster, zero code changes!
```

The bottleneck is almost never where you expect it to be.

Performance Metrics Overview

Training metrics:

- **Throughput**: Samples/second, batches/second
- **Epoch time**: Total time to process entire dataset
- **GPU utilization**: % of time GPU is actively computing
- **Memory usage**: Peak memory allocated

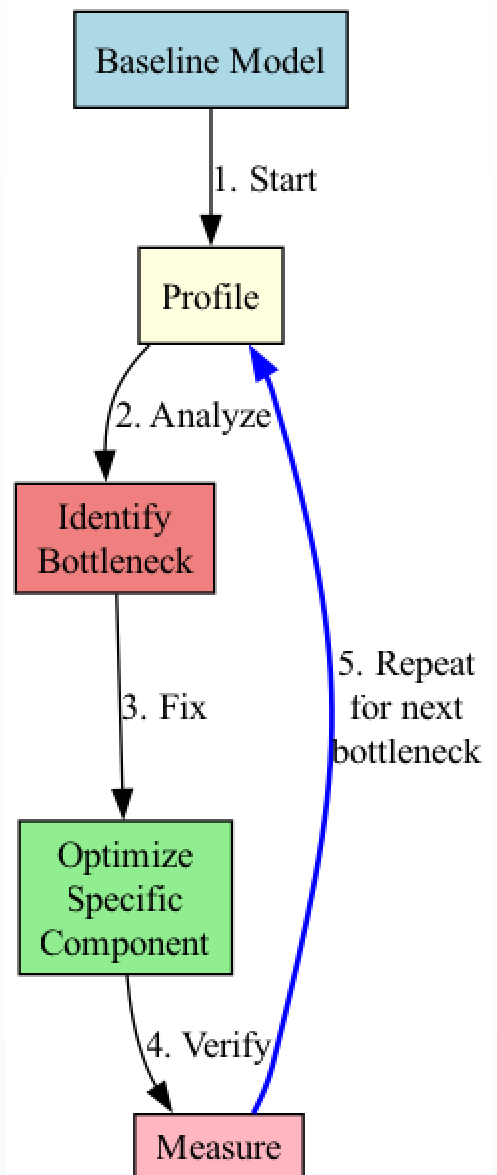
Inference metrics:

- **Latency**: Time per prediction (p50, p95, p99)
- **Throughput**: Predictions/second
- **First token latency**: Time to first output (for GenAI)

Cost metrics:

- **FLOPs**: Floating point operations (theoretical)

The Optimization Loop



Types of Bottlenecks

CPU-bound:

- Data loading and preprocessing
- Tokenization, data augmentation
- Host-to-device memory transfer

GPU compute-bound:

- Too many parameters
- Inefficient operations (small kernels, poor fusion)
- Suboptimal algorithms (e.g., naive attention)

GPU memory-bound:

- Out of memory (OOM) errors
- Batch size limited by VRAM

Profiling Tool Hierarchy

Level 1: Quick checks (seconds)

- `nvidia-smi`: GPU utilization snapshot
- `time` command: Total execution time
- Manual timers: `time.time()`, `time.perf_counter()`

Level 2: Python profiling (minutes)

- `cProfile`: Function-level CPU profiling
- `line_profiler`: Line-by-line profiling
- `memory_profiler`: Memory usage per line

Level 3: Deep profiling (hours)

- PyTorch Profiler: Op-level GPU/CPU profiling
- Nsight Systems: System-wide CUDA profiling

Quick Check: nvidia-smi

Basic monitoring:

```
nvidia-smi
```

Watch mode (update every 1 second):

```
nvidia-smi -l 1
```

Key metrics:

- **GPU-Util**: % of time GPU was busy (aim for >85%)
- **Memory-Usage**: Current / Total VRAM
- **Power**: Current draw vs TDP
- **Temperature**: Thermal throttling at ~85°C

Red flags:

GPU-Util <50%: Likely CPU bottleneck

Python Profiling: cProfile

Built-in function-level profiler:

```
import cProfile
import pstats

# Profile a function
profiler = cProfile.Profile()
profiler.enable()

train_model()  # Your code here
```

Output columns:

- `ncalls`: Number of calls
- `tottime`: Total time in function (excluding sub-calls)
- `cumtime`: Cumulative time (including sub-calls)
- `percall`: Time per call

cProfile Example Output

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002   45.231   45.231 train.py:23(train_epoch)
     1563   12.450    0.008   30.125    0.019 dataloader.py:45(__next__)
     1563    8.234    0.005   15.678    0.010 transforms.py:12(augment)
   156300    4.123    0.000    4.123    0.000 {method 'random' of '_random.Random'}
     1563    3.456    0.002   10.234    0.007 model.py:67(forward)
```

Analysis:

- Data loading (`__next__`) takes 30s out of 45s → CPU bottleneck!
- Random augmentation is expensive → consider caching or GPU augmentation
- Model forward pass is fast (10s) → GPU is underutilized

Line-Level Profiling: line_profiler

More granular than cProfile:

```
from line_profiler import LineProfiler

lp = LineProfiler()
lp.add_function(preprocess_data)
lp.add_function(model.forward)

lp.enable()
train_one_epoch()
lp.disable()
```

Output:

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    15         1      12500.0  12500.0     45.2  img = cv2.imread(path)
    16         1       8500.0   8500.0     30.7  img = cv2.resize(img, (224, 224))
    17         1       6700.0   6700.0     24.1  img = normalize(img)
```

Insight: `cv2.imread` is the slowest → use faster libraries or cache.

Memory Profiling: memory_profiler

Track memory usage line by line:

```python
from memory_profiler import profile

@profile
def train_step(batch):
    images, labels = batch  # Line 1
    images = images.cuda()  # Line 2
    outputs = model(images)  # Line 3
```

Output:

```
Line #    Mem usage    Increment   Line Contents
================================================
     1     2145 MB        0 MB       images, labels = batch
     2     4290 MB     2145 MB       images = images.cuda()
     3     8580 MB     4290 MB       outputs = model(images)
     4     8585 MB        5 MB       loss = criterion(outputs, labels)
```

Insight: Gradients double memory (line 5) → use gradient checkpointing.

PyTorch Built-in Profiling

Torch profiler with CPU/GPU tracing:

```python
from torch.profiler import profile, record_function, ProfilerActivity

with profile(
    activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
    record_shapes=True,
    profile_memory=True,
    with_stack=True
) as prof:
    with record_function("train_epoch"):
        for i, batch in enumerate(dataloader):
            if i >= 10:  # Profile first 10 batches
                break

            with record_function("forward"):
                output = model(batch)

            with record_function("backward"):
                loss.backward()
```

PyTorch Profiler Output

Table view:

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

Output:

```
-------------------------------------------------------  ------------
Name                                     Self CPU time  Self CUDA time
-------------------------------------------------------  ------------
aten::conv2d                                    1.2ms       125.4ms
aten::batch_norm                                0.8ms        45.2ms
```

Insights:

- Convolutions dominate GPU time (expected)
- HtoD memcpy is 23ms → data transfer bottleneck! Use `pin_memory`

TensorBoard Profiler Visualization

Export for TensorBoard:

```python
with profile(
    activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
    on_trace_ready=torch.profiler.tensorboard_trace_handler('./log/resnet18')
) as prof:
```

View in TensorBoard:

```
tensorboard --logdir=./log
```

Visualizations:

- **Timeline**: See GPU kernels, data loading, CPU ops on timeline
- **Operator view**: Breakdown by operation type
- **Kernel view**: GPU kernel efficiency
- **Trace view**: Detailed event trace

Interpreting GPU Timeline

Ideal timeline:

```
GPU: ██████████████████████  (100% busy)
CPU: █  █  █  █  █  █  █  █  █  █  █  (loading data)
```

CPU bottleneck:

```
GPU: █      █      █      █      █      █      █      (idle gaps)
CPU: ██████████████████████  (100% busy)
     ▲     ▲     ▲
     Gaps while waiting for data!
```

Memory transfer bottleneck:

```
GPU: █        █        █        █        █        █
MEM: ▓███▓███▓███▓███▓███▓  (memcpy)
     ▲ Large memory transfers stalling GPU
```

Data Loading Optimization

Problem: GPU idle while CPU loads data.

Solutions:

1. Multi-process data loading:

```
DataLoader(dataset,
    batch_size=32,
    num_workers=4,          # Spawn 4 worker processes
    pin_memory=True,        # Faster GPU transfer
    persistent_workers=True  # Reuse workers across epochs
)
```

2. Prefetching (automatic with `num_workers > 0`):

```
Worker 1: Load batch 1 → Load batch 3 → Load batch 5
Worker 2: Load batch 2 → Load batch 4 → Load batch 6
GPU:      Process batch 1 → Process batch 2 → Process batch 3
```

Data Loading Best Practices

Rule of thumb for `num_workers`:

- Start with `num_workers = min(4, num_cpus)`
- Profile and tune (diminishing returns after ~8)
- Too many workers → memory overhead

Optimization checklist:

```
DataLoader(
    dataset,
    batch_size=32,
    num_workers=4,                # Multi-process loading
```

Advanced: GPU preprocessing:

```
# Use NVIDIA DALI or Kornia for GPU-accelerated augmentation
import kornia.augmentation as K
augment = K.AugmentationSequential(
```

Mixed Precision Training Theory

Float32 (FP32):

- 1 sign bit, 8 exponent bits, 23 fraction bits
- Range: ~10^-38 to 10^38
- Standard for training

Float16 (FP16):

- 1 sign bit, 5 exponent bits, 10 fraction bits
- Range: ~10^-8 to 65504
- 2x memory savings, 2-3x speedup on Tensor Cores

Problem with pure FP16:

- Small gradients underflow to zero
- Large activations overflow to infinity

The Precision Goldilocks Zone

Use "just enough" precision for each operation. Match the tool to the task's needs.

Operation	Precision	Why?
Master weights	FP32	Accumulate tiny updates
Forward pass	FP16	Just math, speed matters
Loss scaling	FP32	Small values matter
Softmax	FP32	Numerical stability

Automatic Mixed Precision (AMP)

Solution: Mixed precision training

Strategy:

1. **Master weights** in FP32 (stored in optimizer)
2. **Forward pass** in FP16 (faster)
3. **Loss** in FP32 (precision for small values)
4. **Backward pass** in FP16 (faster)
5. **Gradient scaling** to prevent underflow
6. **Weight update** in FP32 (master weights)

Gradient scaling:

- Multiply loss by scale factor (e.g., 1024) before backward
- Prevents small gradients from becoming zero in FP16
- Unscale gradients before optimizer step

AMP Implementation in PyTorch

```python
from torch.cuda.amp import autocast, GradScaler

model = MyModel().cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scaler = GradScaler()  # Gradient scaler

for epoch in range(num_epochs):
    for batch in dataloader:
        images, labels = batch
        images, labels = images.cuda(), labels.cuda()

        optimizer.zero_grad()

        # Forward in FP16
        with autocast():
            outputs = model(images)
            loss = criterion(outputs, labels)

        # Backward with gradient scaling
        scaler.scale(loss).backward()
        scaler.step(optimizer)
```

Expected speedup: 1.5-3x on V100/A100/H100 GPUs with Tensor Cores.

AMP Best Practices

When to use AMP:

- ✅ Training CNNs, Transformers on modern GPUs (V100+)
- ✅ Large batch sizes (better Tensor Core utilization)
- ✅ Models with lots of matrix multiplications

When NOT to use AMP:

- ❌ Small models on old GPUs (no Tensor Cores)
- ❌ Models with numerical instability
- ❌ When accuracy drops significantly (rare)

Debugging AMP issues:

Memory Optimization: Gradient Checkpointing

Problem: Storing all activations for backprop uses O(N) memory.

Example (4-layer network):

```
Forward:  Input → Act1 → Act2 → Act3 → Act4 → Loss
Backward: ∇Loss ← ∇Act4 ← ∇Act3 ← ∇Act2 ← ∇Act1
          ▲       ▲       ▲       ▲       ▲
          Need to store all activations!
```

Memory usage: Batch_size × Num_layers × Hidden_dim

Solution: Gradient Checkpointing (Recomputation)

- Store only subset of activations (checkpoints)
- Recompute others during backward pass
- **Trade**: 20-30% slower for 50%+ memory savings

Gradient Checkpointing in PyTorch

```python
import torch.utils.checkpoint as checkpoint

class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(1024, 1024)
        self.layer2 = nn.Linear(1024, 1024)
        self.layer3 = nn.Linear(1024, 1024)

    def forward(self, x):
        # Checkpoint layer1 and layer2
        x = checkpoint.checkpoint(self._forward_layers, x)
        x = self.layer3(x)
        return x

    def _forward_layers(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return x
```

Use case: Train larger models/batches that otherwise OOM.

Gradient Accumulation

Problem: Limited GPU memory → small batch size → poor convergence.

Solution: Accumulate gradients over multiple steps.

```
accumulation_steps = 4  # Effective batch size = 32 * 4 = 128

optimizer.zero_grad()
for i, batch in enumerate(dataloader):
    outputs = model(batch)
    loss = criterion(outputs, labels)

    # Normalize loss by accumulation steps
    loss = loss / accumulation_steps
    loss.backward()

    # Only step optimizer every N batches
    if (i + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

Effect: Simulates large batch training with limited memory.

Compute Optimization: torch.compile

PyTorch 2.0+ feature: JIT compilation for speedups.

```python
import torch

model = MyModel()
model = torch.compile(model)  # Compile the model

# Training loop unchanged
for batch in dataloader:
```

What it does:

- **Graph capture**: Traces model operations
- **Operator fusion**: Merges ops (e.g., Conv+BN+ReLU → 1 kernel)
- **Memory optimization**: Reuses buffers
- **CUDA graph**: Reduces kernel launch overhead

Expected speedup: 10-50% for free!

torch.compile Modes

```python
# Default mode (balanced)
model = torch.compile(model)

# Maximum performance (slower compile time)
model = torch.compile(model, mode="max-autotune")

# Reduce memory usage
model = torch.compile(model, mode="reduce-overhead")

# Debug mode (disable optimizations)
model = torch.compile(model, mode="default", dynamic=True)
```

Caveats:

- First run is slow (compilation overhead)
- Not all operations supported (fallback to eager)
- Dynamic shapes can trigger recompilation

Operator Fusion Example

Without fusion (3 kernel launches):

```
x = conv(input)     # Kernel 1: Convolution
x = bn(x)           # Kernel 2: Batch norm
x = relu(x)         # Kernel 3: ReLU
```

With fusion (1 kernel launch):

```
x = conv_bn_relu(input)  # Single fused kernel
```

Benefits:

- Fewer kernel launches (less overhead)
- Reduced memory bandwidth (no intermediate writes)
- Better cache locality

torch.compile does this automatically!

Flash Attention

Problem: Standard attention has $O(N^2)$ memory complexity.

Standard attention:

```python
# Materialize full N×N attention matrix
scores = Q @ K.T  # (N, N) matrix
attn = softmax(scores)  # (N, N) matrix
```

Flash Attention (Dao et al., 2022):

- Tiled computation (never materialize full matrix)
- Fused kernel (attention + softmax in one pass)
- **Result**: 2-4x speedup, $O(N)$ memory instead of $O(N^2)$

Usage:

```python
from torch.nn.functional import scaled_dot_product_attention

# PyTorch 2.0+ uses Flash Attention automatically!
```

System-Level Optimization

CPU affinity (bind processes to cores):

```
taskset -c 0-7 python train.py  # Use cores 0-7
```

NUMA awareness (multi-socket systems):

```
numactl --cpunodebind=0 --membind=0 python train.py
```

PCIe optimization (multi-GPU):

```
# Use GPUs on same PCIe switch
os.environ["CUDA_VISIBLE_DEVICES"] = "0,1"  # Same switch
```

Storage I/O:

- Use SSD over HDD for datasets
- Use RAM disk for small datasets (`tmpfs`)

Benchmarking Best Practices

1. Warmup runs (JIT compilation, cache warming):

```python
for _ in range(10):
    model(dummy_input)  # Warmup

# Now measure
start = time.time()
for _ in range(100):
    model(input_data)
```

2. Multiple runs (reduce variance):

```python
import numpy as np

times = []
for _ in range(100):
    start = time.perf_counter()
    model(input_data)
    times.append(time.perf_counter() - start)

print(f"Mean: {np.mean(times)*1000:.2f} ms")
```

Benchmarking Checklist

Environment control:

- [ ] Disable CPU frequency scaling (`performance` mode)
- [ ] Close background applications
- [ ] Fix random seeds (`torch.manual_seed(42)`)
- [ ] Use same device (GPU vs CPU)

Measurement:

- [ ] Warmup before timing (10+ iterations)
- [ ] Measure multiple runs (100+)
- [ ] Report mean, std, percentiles (p50, p95, p99)
- [ ] Synchronize CUDA ops (`torch.cuda.synchronize()`)

Comparison:

Common Performance Anti-Patterns

1. Implicit CPU-GPU synchronization:

```python
# BAD: Forces sync every iteration
for i, batch in enumerate(dataloader):
    loss = train_step(batch)
    print(f"Loss: {loss.item()}")  # .item() syncs!

# GOOD: Batch logging
losses = []
for i, batch in enumerate(dataloader):
    loss = train_step(batch)
    losses.append(loss.detach())  # No sync
if i % 100 == 0:
    print(f"Avg loss: {torch.stack(losses).mean()}")
```

2. Small batch sizes (underutilize GPU):

- Batch size 1-8: Poor GPU utilization
- Batch size 32-128: Better (saturate GPU)

Common Performance Anti-Patterns (2)

3. Unnecessary data transfers:

```
# BAD: Transfer to GPU every iteration
for batch in dataloader:
    batch = batch.cuda()  # Slow!

# GOOD: Use pin_memory + non_blocking
dataloader = DataLoader(..., pin_memory=True)
for batch in dataloader:
    batch = batch.cuda(non_blocking=True)  # Faster!
```

4. Inefficient tensor operations:

```
# BAD: Python loop
result = []
for i in range(len(tensor)):
    result.append(tensor[i] * 2)

# GOOD: Vectorized operation
result = tensor * 2  # Much faster!
```

Case Study: Training Speedup

Baseline ResNet-50 on ImageNet:

- Batch size: 32
- Time per epoch: 120 minutes
- GPU utilization: 45%

Optimization steps:

Optimization	Speedup	Cumulative Time
Baseline	1.0x	120 min
+ num_workers=8	1.4x	86 min
+ Mixed precision (AMP)	1.9x	45 min
+ Larger batch (32→128)	2.3x	37 min
+ torch.compile	2.8x	31 min

Final result: 2.8x speedup, 74% faster!

Case Study: Memory Optimization

Problem: Training LLaMA-7B on single A100 (40GB VRAM) OOMs.

Optimization steps:

Technique	Memory Usage	Batch Size
Baseline FP32	52 GB	OOM
FP16	26 GB	1
+ Gradient checkpointing	18 GB	2
+ Gradient accumulation	18 GB	8 (effective)
+ Flash Attention	14 GB	4

Result: Fits on single GPU with effective batch size of 16!

Profiling Workflow Summary

Step 1: Establish baseline

- Measure throughput, latency, memory
- Profile with PyTorch Profiler
- Identify bottleneck category (CPU/GPU compute/GPU memory/I/O)

Step 2: Apply targeted optimization

- CPU bottleneck → `num_workers`, prefetching
- GPU compute → AMP, `torch.compile`, algorithmic improvements
- GPU memory → gradient checkpointing, smaller batch, model parallelism
- I/O → faster storage, caching, data format (HDF5, LMDB)

Step 3: Measure impact

- Re-run profiling

Optimization Priority

Quick wins (do first):

1. ✅ Enable AMP (5 min, 1.5-2x speedup)
2. ✅ Tune `num_workers` (10 min, 1.2-1.5x speedup)
3. ✅ Use `torch.compile` (1 line, 1.1-1.5x speedup)
4. ✅ Enable `pin_memory=True` (1 parameter, 1.1x speedup)

Medium effort (if needed):

5.
⚙️
Gradient accumulation (if memory-limited)

6.
⚙️
Larger batch size (if hardware allows)

7.

Tools Ecosystem Summary

Profiling:

- `nvidia-smi` : GPU monitoring
- `cProfile` : Python function profiling
- `line_profiler` : Line-level profiling
- `memory_profiler` : Memory usage
- PyTorch Profiler: Deep PyTorch profiling
- TensorBoard: Visual profiling
- Nsight Systems/Compute: Expert CUDA profiling

Optimization:

- `torch.cuda.amp` : Mixed precision
- `torch.compile` : Graph optimization
- `torch.utils.checkpoint` : Gradient checkpointing

Lab Preview

Today's mission:

1. **Part 1**: Profile ResNet-18 training and identify bottlenecks
2. **Part 2**: Optimize data loading (num_workers, pin_memory)
3. **Part 3**: Apply mixed precision training (AMP)
4. **Part 4**: Use gradient checkpointing to fit larger batch
5. **Part 5**: Apply torch.compile and measure speedup
6. **Part 6**: Create comprehensive performance comparison

Deliverable: Optimization report showing 2-3x speedup!

Key Takeaways

1. **Always profile before optimizing** - measure, don't guess
2. **Focus on the critical path** - optimize what matters (training loop)
3. **Quick wins first** - AMP, num_workers, torch.compile are easy
4. **Memory vs speed trade-offs** - gradient checkpointing, accumulation
5. **Benchmark properly** - warmup, multiple runs, synchronization
6. **Iterative process** - profile → optimize → measure → repeat

Remember: A 2x speedup means 2x more experiments, faster iteration, and cheaper costs!

Additional Resources

Documentation:

- PyTorch Profiler: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html
- PyTorch Performance Tuning: https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html
- torch.compile: https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html

Papers:

- Mixed Precision Training (Micikevicius et al., 2018)
- Flash Attention (Dao et al., 2022)
- Gradient Checkpointing (Chen et al., 2016)

Tools:

- TensorBoard: https://www.tensorflow.org/tensorboard
- Nsight Systems: https://developer.nvidia.com/nsight-systems

[illegible]