PART III · CHAPTER 9

Operator Fusion & the Compiler Stack

How Compilers Turn Naive PyTorch into Near-Peak Hardware Performance

Book: AI/ML Infrastructure from Silicon to Scale | **Author:** Venkat Vinjam | **Cross-ref:** *Roofline & regimes (Ch 1) · HBM bandwidth (Ch 4) · CUDA kernels/Triton (Ch 7) · Quantization (Ch 8) · Training (Ch 10) · KV cache (Ch 11)*

> *"The most important optimization in deep learning compilers is operator fusion. Every other optimization is either a special case of fusion or a precondition for it."*
>
> — Horace He, "Making Deep Learning Go Brrrr From First Principles"

A naive PyTorch model executes each operation — every LayerNorm, every activation, every matrix multiply — as a separate GPU kernel. Each kernel launch carries ~5μs of CPU overhead. Each intermediate result makes a full round-trip through HBM at 3.35 TB/s. A single transformer layer in eager mode executes approximately 80 kernel launches and 120 HBM round-trips, even though the computation theoretically requires only 2 (input → output). The result: models running at 10–20% of theoretical peak despite perfect algorithmic design.

The compiler stack exists to close this gap. Through operator fusion, graph optimization, and code generation, modern DL compilers — TorchInductor, XLA, and TensorRT — routinely close 60–80% of the gap between naive dispatch and hardware limits. Applied together (torch.compile + FP8 + FlashAttention + TRT-LLM), the cumulative gain reaches 5× throughput over PyTorch eager on identical hardware. This chapter explains how — from first principles of HBM round-trips through the full four-layer compilation pipeline.

FOUNDATIONAL

- Explain why separate GPU kernels waste HBM bandwidth even for correct computation
- Describe the three performance regimes and which compilers target which
- Understand what torch.compile does in plain terms (no compiler theory needed)
- Know the factory metaphor for why fused cos×64 takes the same time as cos×1

INTERMEDIATE

- Explain the four-layer PyTorch 2.x compilation pipeline with each layer's role
- Quantify HBM traffic saved by fusing LayerNorm+GeLU+Dropout with exact numbers
- Diagnose and fix the six most common torch.compile graph break causes
- Compare XLA vs TorchInductor vs TensorRT for a given deployment scenario

PRINCIPAL LEVEL

- Design the full compiler optimization strategy for a new LLM serving system
- Quantify the cumulative speedup stack and ROI for each compiler investment
- Explain min-cut recomputation and how fusion changes activation checkpointing economics
- Evaluate KernelEvolve-style AI-driven optimization for custom operators

9.1 The Three Performance Regimes — Diagnose Before Optimizing

Every GPU kernel and every workload falls into one of three performance regimes. Applying the wrong optimization wastes engineering time and can hurt performance. Identifying the regime is the prerequisite for every decision in this chapter.

```
THE THREE PERFORMANCE REGIMES (from Ch 1 Roofline):

REGIME 1: COMPUTE-BOUND
  Symptom:  Tensor Cores fully saturated; HBM bandwidth underused
  Diagnosis: Arithmetic Intensity > Ridge Point (~295 FLOP/byte H100)
  Examples: Large-batch prefill GEMM, training forward pass
  Compiler impact: MINIMAL — computation is the bottleneck, not overhead
  What TO do: FP8 quantization, larger tile sizes, TC alignment (Ch 7+8)

REGIME 2: MEMORY-BANDWIDTH-BOUND
  Symptom:  HBM BW saturated; Tensor Cores mostly idle
  Diagnosis: Arithmetic Intensity < Ridge Point
  Examples: Decode at batch=1, LayerNorm, activations, RMSNorm, GEMV
  Compiler impact: HIGH — operator fusion eliminates HBM round-trips
  What TO do: Fusion (this chapter), quantization to reduce bytes/token

REGIME 3: OVERHEAD-BOUND (Launch/CPU-Bound)
  Symptom:  GPU idle between tiny kernels; CPU dispatch is bottleneck
  Diagnosis: SM util <30% despite non-trivial work; GPU time << CPU dispatch time
  Examples: batch=1 decode without CUDA Graphs; many small ops
  Compiler impact: VERY HIGH — CUDA Graphs + torch.compile eliminate launch overhead
  What TO do: CUDA Graphs, torch.compile, C++ runtime

QUICK DIAGNOSIS:
  Step 1: nvidia-smi dmon -s u → watch SM utilization during inference
  Step 2: If SM util <50%: likely overhead-bound or memory-bound
  Step 3: Nsight Compute Roofline:
           - Below memory ceiling → memory-bandwidth-bound
           - Below compute ceiling → compute-bound
           - Below BOTH but neither saturated → overhead-bound
  Step 4: DCGM_FI_PROF_SM_ACTIVE vs DCGM_FI_PROF_SM_OCCUPANCY:
           - Active high, occupancy low → compute or memory bound
           - Both low → overhead/launch bound
```

9.2 Operator Fusion — The Foundation of DL Compilation

9.2.1 The HBM Round-Trip Problem

Every unoptimized neural network layer executes as a sequence of separate kernel launches, each with its own HBM read–compute–write cycle. The waste is not in the arithmetic — it is in the truck trips between warehouse and factory floor.

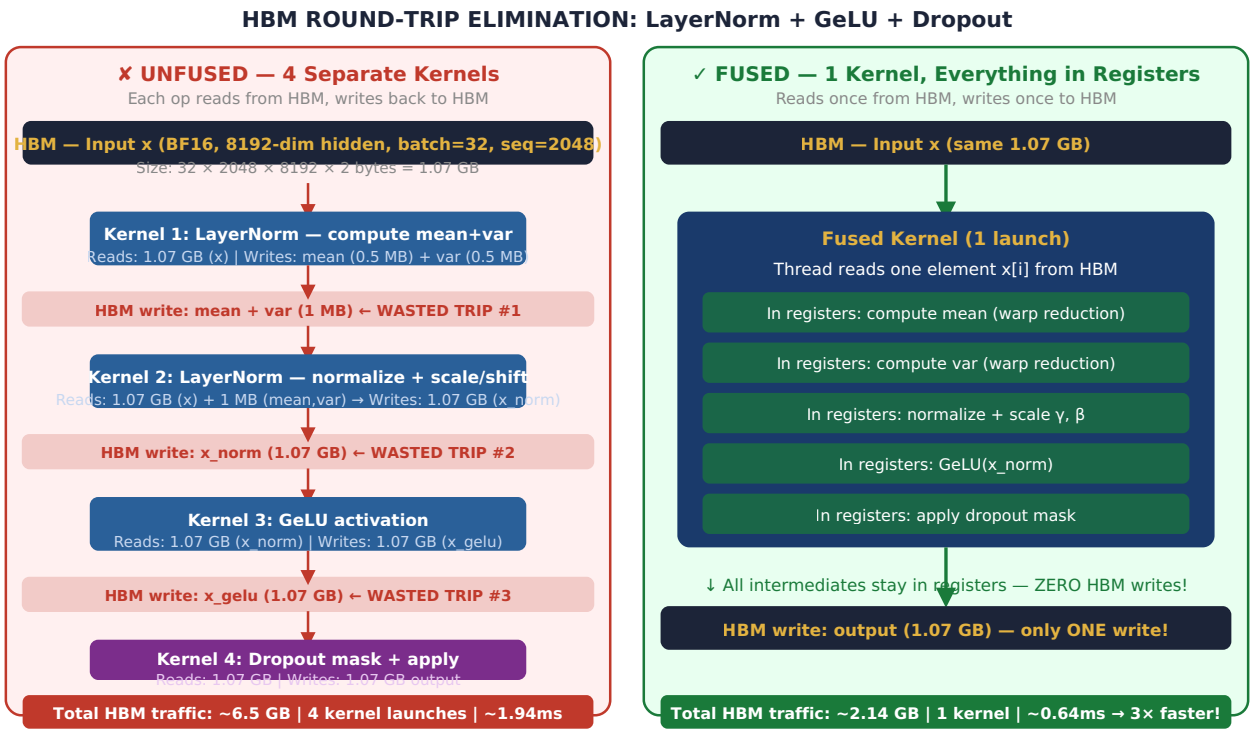


Figure 9.1 *— HBM round-trip elimination via operator fusion. The unfused sequence (LayerNorm mean/var → LayerNorm normalize → GeLU → Dropout) requires 4 kernel launches and ~6.5 GB HBM traffic for batch=32, seq=2048, d=8192. The fused kernel performs all four operations in registers between one HBM read and one write — total traffic drops to ~2.14 GB (~3× reduction) and latency falls from ~1.94ms to ~0.64ms. The intermediate activations (mean, var, x_norm, x_gelu) never touch HBM.*

9.2.2 The Factory Metaphor — Why Fusion Works

The clearest mental model for fusion: HBM is a warehouse, SMEM/registers are a factory floor, and HBM transactions are delivery trucks. Separate kernels truck results back to the warehouse between every operation. Fusion keeps data on the factory floor until all work is done.

```
WITHOUT FUSION — truck the result back between every operation:
  Warehouse (HBM) → Truck → Factory (registers): x.cos()
  Factory → Truck → Warehouse: write cos_x
  Warehouse → Truck → Factory: read cos_x.cos()
  Factory → Truck → Warehouse: write cos_cos_x
  ... × 64 applications: 128 truck trips total

  99% of time is truck transport, not computation.

WITH FUSION — do all operations while the truck is here:
  Warehouse → Truck → Factory: x
  Factory: x.cos().cos().cos()...×64  ← all in registers
  Factory → Truck → Warehouse: result

  Cost: 2 truck trips TOTAL regardless of N operations.

MATHEMATICAL CONSEQUENCE:
  fused x.cos() × 64 times  ≈  same runtime as  fused x.cos() × 1

  Both limited by: 2 HBM accesses ÷ 3.35 TB/s ≈ 0.3µs (at 1 GB tensor)
  NOT by:          64 cos() computations ≈ 0.001µs (registers)

  The 63 extra cos() operations happen at "zero marginal cost."

CRITICAL IMPLICATION: GELU is NOT slower than ReLU in a fused kernel.
  GELU: ~8× more FLOPs than ReLU
  BUT: Both are memory-bandwidth-bound elementwise ops
  BOTH cost exactly 2 HBM accesses — the same bottleneck
  Extra FLOPs vanish into HBM latency.
```

9.2.3 Fusion Categories and Impact

Type	Pattern	Example	Mechanism	Speedup	Auto-Fused By
Type 1 Pointwise+Pointwise	Horizontal chain	RMSNorm + Scale + Dropout	All ops in registers; 1 read, 1 write	2–10× for long chains	torch.compile, NVFuser, XLA
Type 2 Pointwise+Reduction	Vertical	Subtract → Square → Mean (MSE)	Pointwise in registers; reduction via warp shuffle	3–5×	NVFuser, XLA; sometimes torch.compile
Type 3 GEMM+Epilogue	Output fusion	Linear + Bias + GELU	GEMM output stays in SMEM; epilogue before HBM write	1.3–2×	cuBLASLt epilogue, CUTLASS, TRT-LLM
Type 4 Tiled Self-Fusion	SMEM tiling	Flash Attention	SMEM tiling prevents N^2 attention matrix from touching HBM	2–5× end-to-end	Flash Attention v3, cuDNN, TRT-LLM
Type 5 Cross-Layer	Vertical graph	GEMM → RMSNorm → GEMM	Intermediate tensor eliminated between layers	1.2–1.8×	TensorRT, some torch.compile patterns

9.2.4 The Min-Cut Recomputation Problem

Fusion interacts with activation checkpointing in a non-obvious and economically important way. When an operation is fused, its recomputation cost collapses to nearly zero — changing the optimal checkpoint placement dramatically.

```
MIN-CUT RECOMPUTATION ECONOMICS:

UNFUSED RMSNorm recomputation cost:
  Forward: 5 HBM round-trips (5 kernels)
  Recomputing in backward: another 5 HBM round-trips
  Net recompute cost: HIGH — prefer to store activation

FUSED RMSNorm recomputation cost:
  Forward: 1 HBM round-trip (1 fused kernel)
  Recomputing in backward: another 1 HBM round-trip
  Net recompute cost: ~ZERO (bandwidth cost = 1 HBM read)
  Memory saved: entire intermediate tensor (~16 GB across 80 layers LLaMA-3 70B)

OPTIMAL POLICY:
  Fused ops: RECOMPUTE (free recompute, saves memory)
  Attention:  STORE (recomputing flash attention still reads K,V from HBM)
  GEMM:       STORE (compute-expensive to recompute)

AOTAutograd MIN-CUT ALGORITHM:
  Models graph as flow network; finds minimum-cost partition into
  {save set} and {recompute set} subject to memory budget.
  Result: 60% memory reduction with <5% extra compute
  vs naive recompute-all: 60% memory reduction with 30% extra compute

Enabling in PyTorch:
  from torch.utils.checkpoint import checkpoint_wrapper, CheckpointImpl
  model = checkpoint_wrapper(model, checkpoint_impl=CheckpointImpl.NO_REENTRANT)
```

9.3 The PyTorch 2.x Compiler Stack

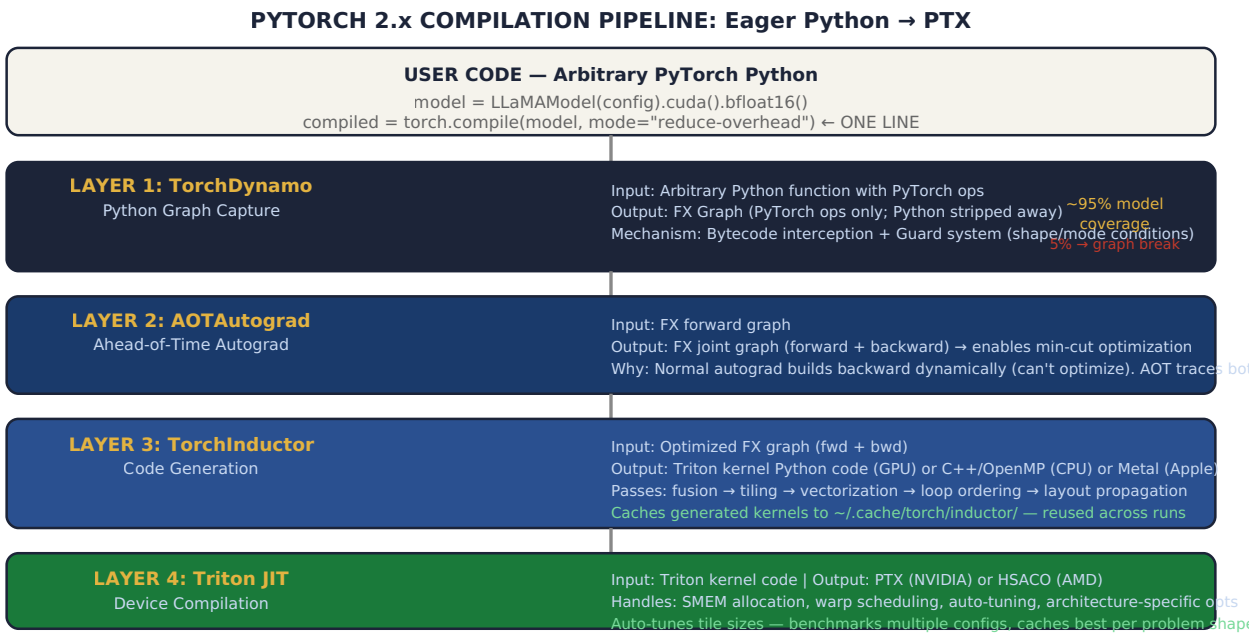


Figure 9.2 *— PyTorch 2.x four-layer compilation pipeline. TorchDynamo captures the computation graph from arbitrary Python. AOTAutograd creates a joint forward+backward graph for min-cut optimization. TorchInductor fuses operations and generates Triton kernels. The Triton JIT compiles to PTX/HSACO and auto-tunes tile sizes. The entire pipeline is triggered by one line:* `torch.compile(model)`.

9.3.2 torch.compile Modes — Tradeoff Chart

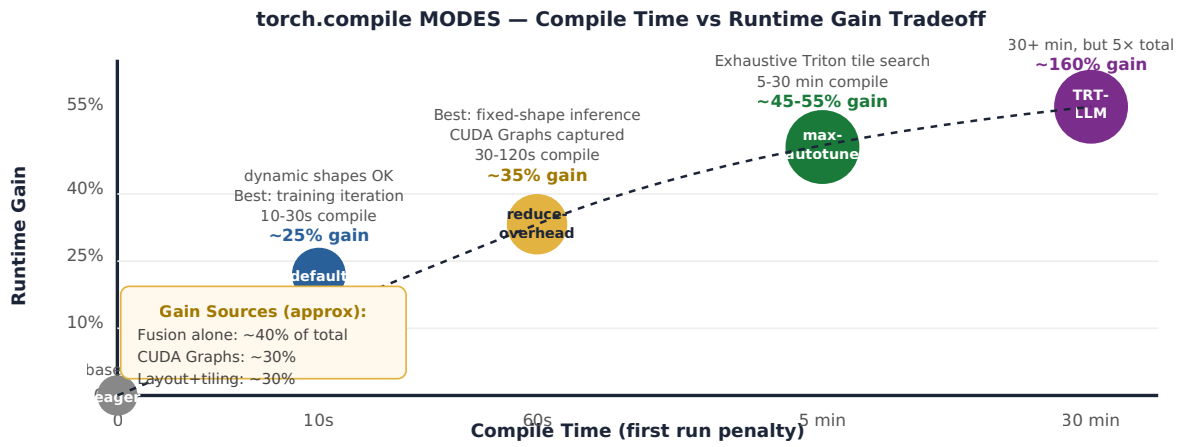


Figure 9.3 *— torch.compile modes: compile-time investment vs runtime gain tradeoff.* `default` *mode gives ~25% gain in 10–30 seconds and handles dynamic shapes.* `reduce-overhead` *captures CUDA Graphs for an additional ~10% at 30–120s compilation cost — best for fixed-shape inference.* `max-autotune` *exhaustively searches Triton tile configurations for another 10–20% gain at 5–30 minutes. TRT-LLM sits above the curve — it is not just "more torch.compile" but a fundamentally different approach with 2× the gain of max-autotune.*

```
torch.compile PERFORMANCE BENCHMARKS (PyTorch official):

Model               Mode             Speedup    Primary Source
──────────────────────────────────────────────────────────────
BERT-large          default          1.28×      Elementwise fusion
                    max-autotune     1.42×      + tile size tuning
LLaMA-2 7B (infer)  reduce-overhead  1.31×      CUDA Graphs dominant
LLaMA-2 7B (train)  default          1.24×      Min-cut recomputation
ViT-L/16            default          1.35×      Attention + norm fusion

WHEN torch.compile HURTS performance:
  Variable input shapes:  Each new shape → recompilation
    Fix: torch.compile(dynamic=True) — slightly slower but shape-agnostic
  Many graph breaks:      Overhead dominates; break count > 10 = problem
  Short sequences (<50 tokens): Compilation overhead not amortized
  Python-heavy control flow: Dynamo falls back to Python repeatedly
```

9.4 XLA — The Functional Compiler (JAX / TensorFlow)

XLA (Accelerated Linear Algebra) is Google's compiler for JAX and TensorFlow. Its design philosophy differs fundamentally: where TorchInductor works on traced Python graphs, XLA works on HLO (High-Level Operations) — a purely functional, side-effect-free IR with all shapes known at compile time.

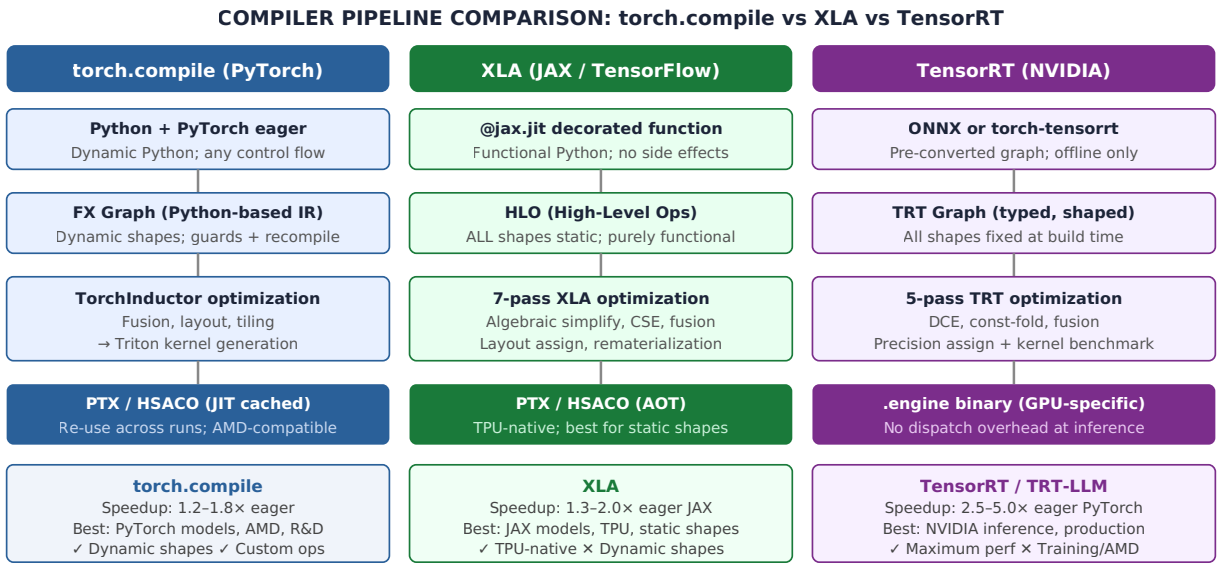


Figure 9.4 *— Three-compiler pipeline comparison. torch.compile is JIT, handles dynamic shapes, and works on AMD — the right choice for PyTorch models, training, and research. XLA is AOT, requires static shapes, and is the only compiler for TPU — the right choice for JAX models and Google infrastructure. TensorRT is the offline maximum-performance choice for NVIDIA inference production — 2.5–5× over eager with the highest engineering investment. All three converge to PTX/HSACO on the same hardware.*

9.5 TensorRT — Production Offline Compilation SHIPPED

TensorRT performs exhaustive offline compilation that produces a GPU-specific engine binary. Unlike JIT compilers, TRT benchmarks every available CUDA kernel for every operation shape in the model — a 5–30 minute investment that pays back over millions of inference calls.

```
TENSORRT 5-PASS OPTIMIZATION PIPELINE:

PASS 1: Graph Cleaning
  Dead code elimination:   Remove ops never contributing to output
  Constant folding:        Pre-compute all-constant ops (RoPE inv_freq, attn scale)
  Redundant elimination:   Remove unnecessary transpose/reshape ops

PASS 2: Layer Fusion
  Vertical chain fusion:   Conv → BN → ReLU → fused CBR kernel
  Attention fusion:        QKV proj + attention + output proj → one kernel
  QKV projection fusion:   3 separate Linear → 1 wider GEMM (3× width)
  Horizontal fusion:       Parallel ops merged into batched kernels

PASS 3: Precision Assignment
  Per-layer precision based on calibration data + quality thresholds
  Mixed: some layers FP16, others INT8/FP8 based on sensitivity

PASS 4: Kernel Selection (most expensive, most impactful)
  Benchmarks ALL cuBLAS GEMM algorithms for each problem shape
  Benchmarks ALL cuDNN attention variants
  Selects fastest kernel per op — result: O(1) dispatch at inference
  Time: 5–30 min for 70B model | Payback: amortized over millions of calls

PASS 5: Memory Planning
  Analyzes tensor lifetimes; assigns shared memory allocations
  In-place operations where safe
  Result: 15–25% HBM savings vs naive allocation
```

9.5.2 TRT-LLM vs torch.compile Performance Gap

```
WHY TRT-LLM BEATS torch.compile BY 2× (LLaMA-3 70B, H100, FP8, batch=32):

PyTorch eager:           ~850 tok/s/GPU
torch.compile default:  ~1,100 tok/s/GPU  (+29%)
TRT-LLM FP8:            ~2,200 tok/s/GPU  (+159% vs eager)

THREE MECHANISMS EXPLAINING THE GAP:

1. KERNEL SELECTION DEPTH:
   torch.compile: Triton auto-tunes ~5 tile configurations
   TRT-LLM:       Benchmarks 50+ cuBLAS algorithms per GEMM shape
                  Exhaustive search finds kernels Triton never considers

2. FUSED ATTENTION PLUGIN:
   torch.compile: Uses Flash Attention through operator dispatch
   TRT-LLM:       Uses gpt_attention_plugin — custom CUDA kernel fusing:
                  QKV GEMM + rotary embedding + masked attention + KV cache write
                  in one GPU kernel pass — eliminates multiple HBM round-trips

3. CUDA GRAPH CAPTURE:
   torch.compile (reduce-overhead): Captures graphs at first execution
   TRT-LLM:       Captures CUDA Graphs at startup for ALL decode batch sizes
                  + handles continuous batching within graph replay

PRODUCTION BUILD (trtllm-build):
  trtllm-build \
    --checkpoint_dir ./llama3-70b-trt-ckpt \
    --output_dir ./llama3-70b-engine \
    --gpt_attention_plugin float16 \   # Fused attention
    --quantization fp8 \
    --max_batch_size 64 \
    --tp_size 2
```

9.6 NVFuser — NVIDIA's Runtime Fusion Engine SHIPPED

NVFuser is NVIDIA's production fusion compiler embedded directly in PyTorch, generating CUDA C++ rather than Triton. It is particularly effective for complex pointwise+reduction chains in transformer norms.

```
KEY NVFUSER PATTERNS FOR LLMs:

1. RMSNorm fusion (every LLaMA-family forward pass):
   Ops: sum(x^2), sqrt(sum/N+eps), divide, multiply by weight
   Unfused: 5 kernels, 10 HBM round-trips
   NVFuser: 1 kernel, 2 HBM round-trips  → ~4× speedup

2. SiLU + element-wise multiply (SwiGLU / LLaMA MLP):
   gate = x * sigmoid(x);  output = gate * up
   NVFuser: 1 read (x, up) → compute gate in registers → 1 write  → ~2×

3. Residual + RMSNorm:
   x = x + residual;  x = RMSNorm(x)
   NVFuser: reads x + residual, adds, normalizes, writes — ONE kernel
   Eliminates residual add as a separate HBM round-trip

NVFuser vs TorchInductor:
  NVFuser:       NVIDIA-only; CUDA C++ output; higher ceiling for complex norms
  TorchInductor: NVIDIA + AMD + CPU; Triton output; easier extension; better for training
```

9.7 Graph-Level Optimizations

```
FOUR CLASSIC GRAPH PASSES (applied before code generation):

1. DEAD CODE ELIMINATION (DCE):
   Training code has dropout, logging, debug stats — all dead in inference.
   Impact: 5–15% of training ops eliminated in inference graph.

   Before DCE: x2 = dropout(x1, p=0.1); x3 = x1 + x2; x4 = debug_stats(x3)
   After DCE:  output = x1 + x1  (dropout=0 at eval; debug removed)

2. CONSTANT FOLDING:
   inv_freq = 1.0 / (10000 ** (arange(0, d, 2) / d))  ← same every step!
   After folding: inv_freq = CONSTANT([...])  ← zero runtime compute
   Impact: RoPE positional encodings, attention scale factors → all become constants

3. COMMON SUBEXPRESSION ELIMINATION (CSE):
   Pre-norm transformers compute layer_norm(x) twice in naive code.
   CSE: compute once, share result → eliminates duplicate computation

4. LAYOUT PROPAGATION:
   PyTorch default: row-major [batch, seq, hidden]
   Tensor Cores prefer: specific A/B matrix layouts
   Layout pass: inserts one-time transpose rather than per-GEMM conversion
   Impact: 10–20% GEMM speedup when layout is non-optimal
```

9.8 Graph Break Diagnosis and Fixes

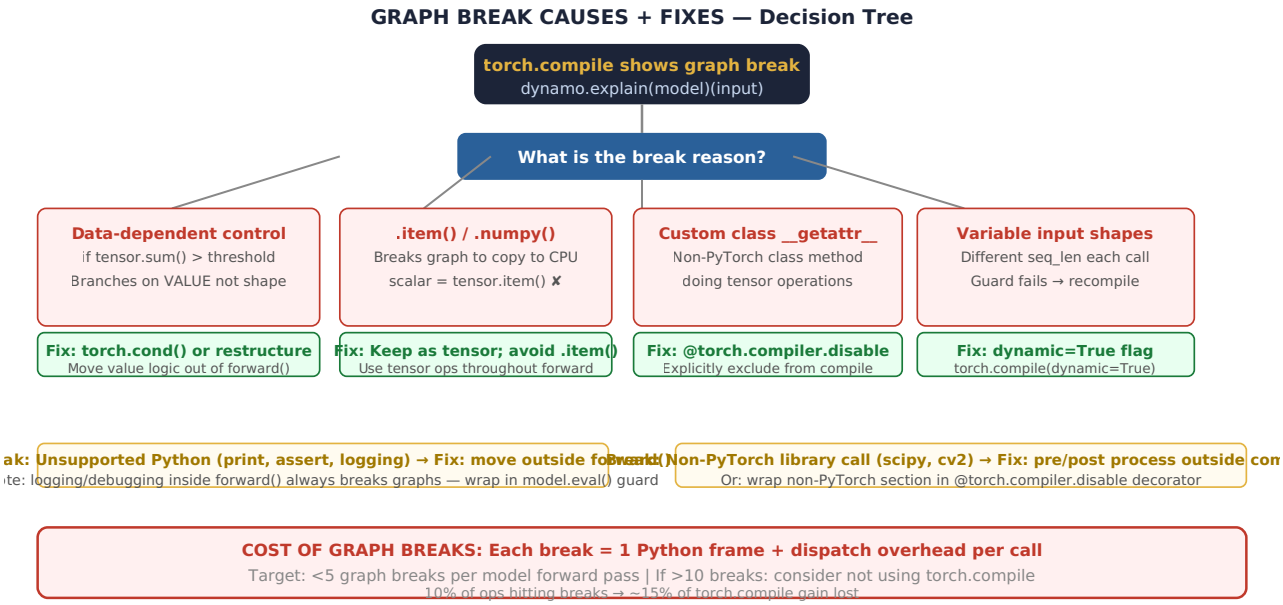


Figure 9.5 *— Graph break causes and fixes. TorchDynamo falls back to Python for six common patterns; each break is a missed fusion opportunity and adds per-call dispatch overhead. The fix for most breaks is structural: move value-dependent logic outside the compiled forward pass, avoid CPU-bound operations inside the graph, and use* `torch.compile(dynamic=True)` *for variable-shape workloads.*

9.9 Compiler Strategy Guide

Use Case	Recommended	Why	Expected Gain
LLM inference production (NVIDIA)	**TRT-LLM**	Maximum throughput; kernel benchmarking; fused attention plugin	2.5–5× vs eager
LLM inference development / AMD	**torch.compile**	Dynamic shapes; Triton on ROCm; rapid iteration	1.3–1.8×
LLM training (all)	**torch.compile + FSDP**	AOTAutograd min-cut; full gradient support; activation checkpointing	1.2–1.5×
JAX model / TPU	**XLA (@jax.jit)**	Only compiler for TPU; static shapes enable aggressive optimization	1.3–2.0×
Fixed-shape inference (NVIDIA)	**torch.compile reduce-overhead**	CUDA Graphs; fast compilation; good for batch servers	1.3–1.5×
Fine-tuning (LoRA)	**torch.compile default**	Gradient support; TRT has no gradient path	1.2–1.4×
Custom architecture / novel ops	**Triton + torch.compile**	Triton for custom kernel; compile handles fusion around it	Varies

Cumulative Compiler Optimization Stack (LLaMA-3 70B, H100)

PyTorch eager baseline: 850 tok/s/GPU. Apply in order: ❶ torch.compile: 1,100 tok/s (+29%). ❷ FP8 quantization (Ch 8): 2,200 tok/s (+100%). ❸ Flash Attention v3: ~2,900 tok/s (+32%). ❹ TRT-LLM migration: ~4,400 tok/s (+52%). Total: 5.2× throughput improvement on identical hardware through software and compiler optimization alone. No hardware upgrade required.

9.10 Meta's KernelEvolve — AI-Driven Kernel Optimization ANNOUNCED

KernelEvolve represents the frontier of compiler optimization: using AI-driven evolutionary search to discover kernel implementations that surpass both hand-written expert kernels and compiler output for custom operators at scale.

```
KERNELEVOLVE ARCHITECTURE (Meta, 2025-2026):

PROBLEM: Meta's production systems have thousands of CUSTOM operators
  Feature hashing, sparse attention, fused interaction layers, custom norms
  Compiler output: 30–50% below theoretical peak for these custom ops
  Manual expert optimization: weeks/op × thousands of ops = infeasible

APPROACH: Treat kernel optimization as a SEARCH problem
  1. SPECIFICATION: engineer defines input/output specs + correctness tests
  2. SEARCH SPACE: tile sizes × loop orderings × memory patterns × Triton/CUTLASS/CUDA
  3. EVOLUTIONARY SEARCH:
       Initial population: compiler-generated baseline + random mutations
       Fitness function: benchmark latency on real H100/MI300X/MTIA hardware
       Crossover: combine tile sizes from one candidate, loop order from another
       100 candidates evaluated in parallel on dedicated eval cluster
  4. VERIFICATION: promising candidates checked for numerical correctness
  5. DEPLOY: best kernel shipped to production

RESULTS (Meta production):
  Ads recommendation models:     60% inference throughput gain
  LLM fine-tuning kernels:       35–45% training speedup
  Custom attention variants:     2.1× vs compiler-generated code
  Time to optimize one operator: Hours vs weeks manual
  Targets: H100, MI300X, MTIA v2 simultaneously
```

9.11 Profiling Cookbook

🔬 **PROFILING COOKBOOK — COMPILER OPTIMIZATION DIAGNOSIS**

```
# ──────────────────────────────────────────
# SYMPTOM: Model runs at 15% SM utilization; target 65%+
# GOAL: Classify regime and apply correct compiler fix
# ──────────────────────────────────────────

# STEP 1: Identify regime with nvidia-smi
nvidia-smi dmon -s u -d 1  # Watch SM utilization every 1 second
# SM util <30% during inference → overhead-bound (CUDA Graphs fix)
# SM util 30-80% → memory-bandwidth-bound (fusion fix)
# SM util >80% → compute-bound (quantization / TC alignment fix)

# STEP 2: Count graph breaks before applying torch.compile
import torch._dynamo as dynamo
model = LLaMAModel(config).cuda().bfloat16()
explanation = dynamo.explain(model)(dummy_input)
print(f"Graphs: {explanation.graph_count}")
print(f"Graph breaks: {explanation.break_count}")
print(f"Break reasons: {explanation.break_reasons}")
# Target: <5 graph breaks. If >10: fix breaks before compiling
# Common reason: "data-dependent control flow" → restructure conditionals
# Common reason: "Unsupported: .item()" → remove .item() from forward()

# STEP 3: Benchmark properly (common mistake: no warmup)
import torch, time

def bench(model, inputs, n_warmup=10, n_bench=100, label=""):
    for _ in range(n_warmup):
        with torch.no_grad(): _ = model(*inputs)
    torch.cuda.synchronize()

    times = []
    for _ in range(n_bench):
        s = torch.cuda.Event(enable_timing=True)
        e = torch.cuda.Event(enable_timing=True)
        s.record()
        with torch.no_grad(): _ = model(*inputs)
        e.record()
        torch.cuda.synchronize()
        times.append(s.elapsed_time(e))

    times.sort()
    print(f"{label}: p50={times[n_bench//2]:.2f}ms p99={times[int(n_bench*.99)]:.2f}ms")

model_eager = LLaMAModel(config).cuda().bfloat16()
```

```
bench(model_eager, inputs, label="Eager")

model_compiled = torch.compile(model_eager, mode="reduce-overhead")
bench(model_compiled, inputs, label="Compiled")  # First call compiles
# IMPORTANT: Measure p50 AND p99. Compilers can improve mean but worsen p99.
# If compiled p99 > eager p99: check for recompilation on variable shapes

# STEP 4: Inspect generated Triton code
import torch._inductor.config as config
config.trace.enabled = True
config.debug.save_output_code = True
# Run one forward pass → generated kernels saved to /tmp/torchinductor_*/
# Look for: fused_ops_0.py, fused_ops_1.py etc.
# Verify fusion: each file should contain multiple operations

# STEP 5: Profile with torch.profiler to see kernel breakdown
from torch.profiler import profile, ProfilerActivity
with profile(activities=[ProfilerActivity.CUDA], record_shapes=True) as prof:
    with torch.no_grad():
        for _ in range(5):
            _ = model_compiled(*inputs)
print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=20))
# Look for: many small kernels → fusion opportunity missed
# Look for: kernel named "fused_*" → torch.compile fusion working
# Look for: "cudaLaunchKernel" overhead → consider CUDA Graphs

# STEP 6: Verify correctness after compilation
with torch.no_grad():
    out_eager = model_eager(*inputs)
    out_compiled = model_compiled(*inputs)
    max_diff = (out_eager - out_compiled).abs().max().item()
    print(f"Max output difference: {max_diff:.6f}")
    assert max_diff < 1e-3, f"Compilation introduced numeric error: {max_diff}"
# Typical compiled vs eager diff: 1e-5 to 1e-4 (numerical rounding only)
# If diff > 1e-3: check for FP16/BF16 precision issues in fused kernel
```

Worked Example: The Compiler Optimization Stack ROI

WORKED EXAMPLE 9.1 — CUMULATIVE COMPILER GAINS: PYTORCH EAGER → TRT-LLM

Setup

LLaMA-3 70B on 256× H100 SXM5 cluster (TP=2, 128 model replicas). Hardware cost: $2/GPU-hour owned. Starting point: PyTorch eager BF16. Goal: quantify the ROI of each compiler investment.

Step 1: Baseline Measurement

```
PyTorch eager BF16 baseline (measured):
  Decode throughput:       850 tok/s/GPU
  Prefill throughput:    85,000 tok/s/GPU
  SM utilization:            18% (overhead-bound during decode)
  GPU cost per 1M tokens: $2/(850 tok/s * 3600) * 10^6 = $0.653/1M tokens
```

Step 2: Apply torch.compile (1 line, 2 minutes engineering)

```
CHANGE: model = torch.compile(model, mode="reduce-overhead")
COMPILE TIME: 90 seconds (one-time; cached thereafter)

MEASURED RESULT:
  Decode: 850 → 1,100 tok/s (+29%)
  Prefill: 85K → 106K tok/s (+25%)
  SM util: 18% → 28% (CUDA Graphs eliminated most launch overhead)

GAIN BREAKDOWN:
  CUDA Graphs:         +12% (overhead regime eliminated)
  Operator fusion:     +10% (LayerNorm, activation chains fused)
  Layout optimization: + 7% (Tensor Core-friendly memory layouts)

COST: 2 minutes engineering | 90s compile time | ZERO hardware change
ROI: $0.653 → $0.504/1M tokens (23% cost reduction)
```

Step 3: Add FP8 Quantization (Chapter 8, 1 day engineering)

```
CHANGE: Enable Transformer Engine FP8 (te.Linear + fp8_autocast)
RESULT: GEMM throughput doubles on H100 Tensor Cores

  Decode: 1,100 → 2,200 tok/s (+100%)
  Weight memory: 140 GB → 70 GB (single H100 now sufficient for TP=1)

HARDWARE REDUCTION: TP=2 → TP=1 (256 GPUs → 128 GPUs for same capacity)
COST IMPACT: Cost per model replica: $2*2=$4/hr → $2*1=$2/hr (half the GPUs!)
  $/1M tokens = $2 / (2,200 * 3600) * 10^6 = $0.252/1M tokens
```

Step 4: Flash Attention v3 (already in vLLM, enable in config)

```
CHANGE: FlashAttention-3 enabled (H100-specific WGMMA+TMA pipeline)
RESULT: Attention throughput +30–50%; memory O(N^2) → O(N)

  Decode: 2,200 → 2,860 tok/s (+30%)
  Enables longer context (8K → 32K) without additional memory

$/1M tokens = $2 / (2,860 * 3600) * 10^6 = $0.194/1M tokens
```

Step 5: TRT-LLM Migration (1 week engineering, one-time build)

```
CHANGE: Full TRT-LLM build with FP8, fused attention plugin, CUDA Graphs
BUILD TIME: 45 minutes per target batch size configuration
RESULT: Exhaustive kernel benchmarking + fused attention plugin

  Decode: 2,860 → 4,400 tok/s (+54%)
  SM utilization: 28% → 72% (approaching memory-bandwidth limit for decode)

$/1M tokens = $2 / (4,400 * 3600) * 10^6 = $0.126/1M tokens
```

Final Summary

Configuration	Tok/s/GPU	$/1M Tokens	vs Baseline	Engineering Effort
PyTorch eager BF16	850	$0.653	1.00×	0 (baseline)
+ torch.compile	1,100	$0.504	1.29×	1 line + 2 min
+ FP8 (TE)	2,200	$0.252	2.59×	1 day
+ Flash Attn v3	2,860	$0.194	3.36×	Config change
+ TRT-LLM	4,400	$0.126	5.18×	1 week

```
TOTAL SAVINGS AT SCALE (256-GPU cluster, 50B tokens/day):

Annual tokens: 18.25 trillion (50B/day * 365)
Annual cost (eager BF16):  18,250,000 * $0.653 = $11.9M
Annual cost (TRT-LLM):     18,250,000 * $0.126 = $2.3M

ANNUAL SAVINGS: $9.6M from software + compiler optimization alone.
Hardware: unchanged — same 256 H100s.
Engineering investment: ~2 weeks total across the 4 optimization steps.
ROI: $9.6M savings / ~$30K engineering cost = 320× return.
```

CHAPTER 9 QUICK REFERENCE — NUMBERS TO KNOW COLD

Metric	Value	Context
torch.compile speedup (typical)	1.2–1.8×	PyTorch eager baseline; LLM inference
torch.compile default compile time	10–30 seconds	One-time; cached to ~/.cache/torch/inductor/
torch.compile max-autotune time	5–30 minutes	Exhaustive Triton tile search
TRT-LLM vs torch.compile	~2× more throughput	LLaMA-3 70B FP8 on H100
CUDA Graph overhead savings	2.4ms/step	480 kernels × 5μs eliminated (see Ch 7)
Graph break cost	~15% gain lost per 10%	10% of ops hitting breaks
Graph break target	<5 per forward pass	Run dynamo.explain() to count
LayerNorm unfused HBM trips	10 round-trips	5 kernels × 2 trips each
LayerNorm fused HBM trips	2 round-trips	1 read + 1 write; ~4× speedup
RMSNorm fused speedup	~4×	NVFuser; 5 kernels → 1
XLA speedup vs JAX eager	1.3–2.0×	Static shapes; TPU-native
KernelEvolve vs compiler output	35–60%	Meta custom ops; hours vs weeks
Total compiler stack gain	~5.2×	Eager → torch.compile + FP8 + FAv3 + TRT-LLM
DCE eliminates (inference)	5–15% of ops	Dropout, debug stats removed
Min-cut extra compute	<5% (selective)	vs 30% naive recompute-all

Key Takeaways

CORE CONCEPTS

- The three performance regimes — compute-bound, memory-bandwidth-bound, and overhead-bound — require different fixes. Compilers primarily target overhead (CUDA Graphs) and memory (fusion). Applying a compiler to a compute-bound workload yields minimal gain. Classify the regime first, always.

- Operator fusion eliminates HBM round-trips by keeping intermediate results in registers. A 4-op chain (LayerNorm mean/var, normalize, GeLU, Dropout) requires 6.5 GB of HBM traffic unfused but only 2.1 GB fused — a 3× traffic reduction that translates to proportional latency improvement for memory-bound ops.
- The factory metaphor captures why fused cos()×64 takes the same time as cos()×1: both are limited by 2 HBM accesses. The 63 extra operations happen at zero marginal cost in registers. This also explains why GeLU is not slower than ReLU in a fused context — both cost exactly 2 HBM transactions.
- Fusion changes the economics of activation checkpointing. Unfused RMSNorm is expensive to recompute (10 HBM trips). Fused RMSNorm is nearly free (2 HBM trips). AOTAutograd's min-cut algorithm exploits this: fused ops become the primary recomputation targets, yielding 60% memory reduction at <5% compute overhead vs 30% for naive recompute-all.

COST & PRODUCTIVITY IMPACT

- The compiler optimization stack delivers 5.2× throughput on identical hardware through four sequential changes: torch.compile (+29%), FP8 quantization (+100%), Flash Attention v3 (+30%), and TRT-LLM migration (+54%). Engineering investment: ~2 weeks. Annual savings on a 256-GPU cluster serving 50B tokens/day: $9.6M.
- torch.compile delivers 1.2–1.8× speedup with one line of code and 10–30 seconds compile time. This is the highest-ROI optimization available in AI infrastructure: the ratio of gain to engineering effort exceeds nearly any other optimization in this book. Apply it before anything else.
- TRT-LLM achieves ~2× more throughput than torch.compile on the same hardware for inference — not by being a "better torch.compile" but by exhaustively benchmarking 50+ cuBLAS kernel variants per GEMM shape (Kernel Selection pass) and using fused custom attention plugins. The 1-week migration investment pays back in days at production scale.

PRINCIPAL-LEVEL INSIGHT

- Graph breaks are the silent killer of torch.compile performance. Each break costs ~15% of total compiler gain per 10% of operations affected. Run `dynamo.explain()` before deploying any compiled model and fix every break source. The target is <5 graph breaks per forward pass — anything above 10 suggests torch.compile may not be worth applying at all.
- XLA and TorchInductor have fundamentally different compilation models, not just different APIs. XLA requires static shapes (all dimensions fixed at compile time); TorchInductor handles dynamic shapes via guard recompilation. For TPU and completely static workloads (fixed-batch inference), XLA's static analysis enables more aggressive optimization. For variable-length inference (the common LLM case), TorchInductor is the correct choice regardless of performance claims.
- KernelEvolve-style AI-driven search is not a research curiosity — it is the answer to a real scalability problem. When a production system has thousands of custom operators each needing per-hardware optimization, manual expert tuning is infeasible and compiler output is 30–50% below peak. Evolutionary search on dedicated eval clusters closes this gap in hours. Any infrastructure team at Meta/Google/Microsoft scale building custom operators should be building equivalent capability.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. A LLaMA-3 70B inference system shows 18% GPU SM utilization during batch=1 decode. Nsight Compute shows the system is not near either the compute or memory bandwidth ceiling. Which of the three performance regimes applies? What is the most likely root cause, and what is the single highest-impact fix?
2. Explain using the factory metaphor why a fused chain of 64 cos() operations takes the same time as a single cos() operation. Where do the 63 intermediate results live? What would break this equality and force the intermediate results back to HBM?
3. A team switches to torch.compile and observes 47 graph breaks in their model. What does each graph break mean for fusion and performance? Without knowing the specific break reasons, what two diagnostic steps would you take first?

INTERMEDIATE Apply and calculate — production context

1. A transformer layer applies LayerNorm (2 kernels: mean/var + normalize), GELU, Dropout, and Residual Add unfused. Input tensor: batch=32, seq=2048, d=8192, BF16 (2 bytes/element). (a) Calculate total HBM traffic for the unfused sequence. (b) Calculate HBM traffic after fusing all 4 ops into one kernel. (c) At H100 3.35 TB/s, what is the latency improvement? (d) How many such layers are in LLaMA-3 70B, and what is the total per-forward-pass savings?
2. You run torch.compile on a LLaMA model and measure: eager=18ms, compiled=13ms per forward pass. You then switch to mode="max-autotune" and measure 11ms. (a) What is the speedup of each mode vs eager? (b) What additional optimization does max-autotune perform that default does not? (c) If compilation took 20 minutes for max-autotune and the model runs 1M inferences per day, what is the break-even analysis in compute time recovered vs compilation cost?
3. A model that runs perfectly in PyTorch eager has three graph breaks in torch.compile: one from `if activations.sum() > threshold: clip_grads()`, one from `stats = activations.std().item()`, and one from a custom logging call inside forward(). (a) Classify the type of each break. (b) Provide the specific fix for each. (c) If the model has 400 total operations and 10% hit a break each step, approximately how much of the theoretical torch.compile gain is being lost?

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. **End-to-end compiler strategy for a new LLM serving system:** Your team is launching a production serving system for a custom 30B MoE model (8 experts, top-2 routing, novel sparse attention mechanism, variable sequence lengths 256–32K). Hardware: 64× H100 SXM5. Quality bar: MMLU within 0.5% of BF16. (a) For each compiler option (torch.compile, TRT-LLM, XLA), assess feasibility given the MoE routing and variable shapes. (b) Design a compilation strategy for the sparse attention: what can the compiler handle, what needs a custom Triton kernel, and what would KernelEvolve add? (c) Estimate the end-to-end throughput gain from your proposed stack vs PyTorch eager, and calculate the cost-per-1M-tokens at $2/GPU-hr.

2. **Min-cut recomputation design for training:** You are training LLaMA-3 70B on 1,024 H100s with FSDP. Memory is tight: 78 GB used of 80 GB HBM. Activation checkpointing is needed. (a) Without fusion, what is the extra compute overhead of checkpointing every transformer sub-layer? (b) After enabling torch.compile with full fusion, which operations become essentially free to recompute (give specific examples) and which remain expensive? (c) Design the min-cut partition for one transformer layer: which tensors should be saved, which should be recomputed, and what is your estimated memory savings and compute overhead? (d) How does this change if you also enable FP8 training via Transformer Engine?

3. **Compiler investment ROI analysis for a growing platform:** You operate a platform serving 3 models (LLaMA-3 70B, Code-Llama 34B, Mistral 7B) on 512 H100s at $2/GPU-hr. Current state: PyTorch eager BF16. Monthly serving volume: 200B tokens. You have three compiler investment options with the following profiles: (A) torch.compile for all models — 2 weeks, 1 engineer, +30% throughput; (B) TRT-LLM for 70B only (80% of volume) — 6 weeks, 2 engineers, +160% throughput on that model; (C) Custom Triton kernels for 7B model (5% of volume) — 4 weeks, 1 engineer, +80% on that model. (a) Calculate monthly cost before optimization. (b) Calculate monthly savings and ROI for each option individually. (c) Design the optimal investment sequence given an engineering team of 3 people and a 12-week horizon. Justify every priority decision quantitatively.
