Optimizing LLM Inference: Triton Kernels and Sparse Attention

Alif Ilham Madani

April 22, 2026

1 Baseline Walkthrough

1.1 Baseline 1 (Naive PyTorch)

The objective of Baseline 1 is to implement a functional, unoptimized sparse attention mechanism using native PyTorch operations. This is done by creating a sparse block mask based on specific causal, local, and dilation rules and applying it correctly to the attention scores before the softmax reduction.

1.1.1 Sparse Keep Mask Logic

The `_build_sparse_keep_mask` function constructs a boolean mask of shape `[1, num_heads, target_len, source_len]`.

```
    def _build_sparse_keep_mask(
        self,
        *,
        input_pos: torch.Tensor,
        num_heads: int,
        source_len: int,
        block_size: int,
        dilation: int,
        local_blocks: int,
    ) -> torch.Tensor:
        T = int(input_pos.numel())
        S = int(source_len)
        q_pos = input_pos.view(1, 1, T, 1)
        kv_pos = torch.arange(S, device=input_pos.device).view(1, 1, 1, S)
        causal = q_pos >= kv_pos
        q_block = q_pos // block_size
        kv_block = kv_pos // block_size
        block_delta = q_block - kv_block
        local_keep = block_delta <= local_blocks - 1
        head_id = torch.arange(num_heads, device=input_pos.device).view(1, num_heads, 1, 1)
        long_keep = kv_block % dilation == head_id % dilation
        keep_mask = causal & (local_keep | long_keep)
        return keep_mask
```

We first establish absolute positions and enforce the strict causal condition (`q_pos >= kv_pos`). We then map token positions to block IDs using integer division to enforce the assignment rules: queries can always attend to blocks within the `local_keep` delta, and for older blocks, heads apply a modulo stripe pattern (`long_keep`) dictated by their `head_id` and the `dilation` factor. Finally, these constraints are combined using logical AND/OR operations to return the boolean mask.

1.1.2 Masked Attention Logic

The `_masked_attention` function applies the computed keep mask to the dense attention matrices.

```
    def _masked_attention(
        self,
        *,
        q: torch.Tensor,
        k: torch.Tensor,
        v: torch.Tensor,
        keep_mask: torch.Tensor,
    ) -> torch.Tensor:
        scores = torch.matmul(q.float(), k.float().transpose(-2, -1)) * self.scale
        scores = scores.masked_fill(~keep_mask, float("-inf"))
        probs = torch.softmax(scores, dim=-1).to(q.dtype)
        output = torch.matmul(probs, v)
        return output
```

To maintain numerical stability, the `q` and `k` projections are cast to `float32` before the scaled dot-product. Positions marked `False` in the `keep_mask` are filled with `-inf` so they become zero during the softmax operation. Finally, probabilities are cast back to the native datatype (`float16`) to compute the weighted sum of `v`.

Fig. 1 shows the verification of the Baseline 1 evaluation script passing the correctness checks.

```
=== Evaluation Result ===
Correctness: PASS
Correctness details:
  tolerances: atol=0.200, rtol=0.100
  fail_rate_tol: 0.05
  global: max_abs_diff=0.0723, ref_at_max_abs_diff=-2.4434, pred_at_max_abs_diff=-2.3711
  per regime:
    Reg Verdict    Mismatch  Elements  FailRate   MaxAbs   Ref@Max  Pred@Max
    R1  PASS              0    151936    0.0000   0.0195   -2.4355   -2.4551
    R2  PASS              0    151936    0.0000   0.0596    0.7202    0.6606
    R3  PASS              0    151936    0.0000   0.0527   -0.6953   -0.7480
    R4  PASS              0   1215488    0.0000   0.0723   -2.4434   -2.3711

Memory:
  max_memory_allocated: 7.92 GB
  max_memory_reserved: 10.65 GB
  per-regime peak memory:
    R1: allocated=7.92 GB, reserved=10.60 GB
    R2: allocated=7.92 GB, reserved=10.60 GB
    R3: allocated=7.92 GB, reserved=10.65 GB
    R4: allocated=7.92 GB, reserved=10.60 GB

Per-regime metrics:
    Reg Metric            Value      Unit    Better
    R1  Throughput        24.78     tok/s    higher
    R2  Throughput        17.56     tok/s    higher
    R3  TTFT            1885.93        ms     lower
    R4  TTFT            1470.32        ms     lower

Runtime:
  setup_time: 57.27 s
  total_timed_run: 15.81 s
```

Figure 1: Output logs confirming PASS for correctness and recording initial Baseline 1 metrics.

1.2 Baseline 2 (Compiled PyTorch)

The objective of Baseline 2 is to retain the mathematical correctness established in Baseline 1 while optimizing the execution overhead. Specifically, it applies `torch.compile` to the one-token decode hot path. Because decoding repeats a small step many times, minimizing Python and CUDA launch overhead causes significant speedups without changing the underlying model logic.

1.2.1 Compile Hookup Logic

The integration occurs within the `setup` function, targeting only the decode-heavy evaluation regimes.

```
regime = str(regime_name).upper()
if regime in {"R1", "R2"}:
    model.compiled_decode_one_token = torch.compile(
        model.decode_one_token,
        mode="reduce-overhead",
        fullgraph=True,
    )
else:
    model.compiled_decode_one_token = None

_warmup_setup(model, regime_name)
```

We restrict the compilation to regimes `R1` and `R2`, which heavily exercise the step-by-step decode path. We utilize `mode="reduce-overhead"` to specifically target and minimize the CPU/CUDA launch bottlenecks inherent in small, repeated operations. The `fullgraph=True` argument asserts that the entire one-token decode step can be captured as a single, unbroken graph without falling back to Python [2]. Finally, a warmup chunk is passed through the model via `_warmup_setup` to ensure that the heavy graph-capture and compilation costs are paid upfront, preventing them from skewing the timed evaluation metrics.

Fig. 2 shows the verification of the Baseline 2 evaluation script showing the improved decode throughput.

1.3 Baseline 3 (Fused Projections PyTorch)

The objective of Baseline 3 is to reduce execution overhead by fusing multiple independent linear projection calls that share the same input tensor into a single matrix multiplication. The output is then split back into the original required tensors. This strategy minimizes the number of individual operations dispatched to the GPU.

1.3.1 State Dictionary Fusion Logic

Before the model can run the fused forward pass, the checkpoint weights (which are stored as separate tensors) must be combined during the model setup phase. This is handled in the `_fuse_projection_keys_for_load` function.

```
def _fuse_projection_keys_for_load(
    state_dict: dict[str, torch.Tensor],
    cfg: dict,
    ) -> dict[str, torch.Tensor]:
    out = dict(state_dict)
    num_layers = int(cfg["num_hidden_layers"])
    attention_bias = bool(cfg["attention_bias"])

        for i in range(num_layers):
            prefix = f"layers.{i}"

            # Fuse Q, K, V weights
            q_weight = out.pop(f"{prefix}.attn.q_proj.weight")
            k_weight = out.pop(f"{prefix}.attn.k_proj.weight")
```

```
=== Evaluation Result ===
Correctness: PASS
Correctness details:
  tolerances: atol=0.200, rtol=0.100
  fail_rate_tol: 0.05
  global: max_abs_diff=0.1094, ref_at_max_abs_diff=-2.6016, pred_at_max_abs_diff=-2.7109
  per regime:
    Reg Verdict    Mismatch  Elements  FailRate   MaxAbs   Ref@Max  Pred@Max
    R1  PASS              0    151936    0.0000   0.0176    3.8105    3.7930
    R2  PASS              0    151936    0.0000   0.1094   -2.6016   -2.7109
    R3  PASS              0    151936    0.0000   0.0527   -0.6953   -0.7480
    R4  PASS              0   1215488    0.0000   0.0723   -2.4434   -2.3711

Memory:
  max_memory_allocated: 7.92 GB
  max_memory_reserved: 10.65 GB
  per-regime peak memory:
    R1: allocated=7.92 GB, reserved=10.60 GB
    R2: allocated=7.92 GB, reserved=10.60 GB
    R3: allocated=7.92 GB, reserved=10.65 GB
    R4: allocated=7.92 GB, reserved=10.60 GB

Per-regime metrics:
    Reg Metric             Value      Unit    Better
    R1  Throughput         73.69     tok/s    higher
    R2  Throughput         31.52     tok/s    higher
    R3  TTFT             1941.40        ms     lower
    R4  TTFT             1543.74        ms     lower

Runtime:
  setup_time: 121.69 s
  total_timed_run: 9.28 s
```

Figure 2: Output logs confirming PASS for correctness and recording compiled Baseline 2 metrics.

```
            v_weight = out.pop(f"{prefix}.attn.v_proj.weight")
            out[f"{prefix}.attn.qkv_proj.weight"] = torch.cat([q_weight, k_weight, v_weight], dim
    =0)

            # Fuse Q, K, V biases conditionally
            if attention_bias:
                q_bias = out.pop(f"{prefix}.attn.q_proj.bias")
                k_bias = out.pop(f"{prefix}.attn.k_proj.bias")
                v_bias = out.pop(f"{prefix}.attn.v_proj.bias")
                out[f"{prefix}.attn.qkv_proj.bias"] = torch.cat([q_bias, k_bias, v_bias], dim=0)

            # Fuse Gate and Up MLP weights
            gate_weight = out.pop(f"{prefix}.mlp.gate_proj.weight")
            up_weight = out.pop(f"{prefix}.mlp.up_proj.weight")
            out[f"{prefix}.mlp.gate_up_proj.weight"] = torch.cat([gate_weight, up_weight], dim=0)

    return out
```

We iterate over each decoder layer, popping the individual `q_proj`, `k_proj`, and `v_proj` weights and concatenating them along `dim=0`. The same operation is applied to the `gate_proj` and `up_proj` weights in the MLP block. If the configuration specifies that attention uses biases, those are also concatenated. The fused tensors are then written back to the state dictionary under the new fused module keys (`qkv_proj` and `gate_up_proj`).

1.3.2 Fused Forward Pass Logic

Once the weights are fused, the forward pass of the attention and MLP modules must be updated to execute the single projection and split the result.

Attention QKV Forward:

```
def forward(self, x: torch.Tensor, *, input_pos: torch.Tensor, freqs_cis: torch.Tensor, mask:
    BlockMask) -> torch.Tensor:
    bsz, seq_len, _ = x.shape

    # Run the fused projection
    x = self.qkv_proj(x)

    # Split the output back into query, key, and value tensors
    q_lin, k_lin, v_lin = x.split(self.qkv_out_splits, dim=-1)

    # ... rest of the attention logic ...
```

Decoder MLP Forward:

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    # Run the fused projection
    x = self.gate_up_proj(x)

    # Split the output back into gate and up tensors
    gate, up = x.split(self.gate_up_out_splits, dim=-1)

    return self.down_proj(torch.nn.functional.silu(gate) * up)
```

In both instances, the input tensor `x` is passed through the single fused linear layer. We then utilize `tensor.split()` along the last dimension (`dim=-1`) utilizing the pre-calculated split sizes (`self.qkv_out_splits` and `self.gate_up_out_splits`) to unpack the results. This achieves the exact same mathematical output as Baseline 1, but with significantly fewer discrete PyTorch calls.

Fig. 3 shows the verification of the Baseline 3 evaluation script passing the correctness checks.

1.4 Baseline 4 (Triton INT4 Decode Head)

The objective of Baseline 4 is to introduce custom GPU kernels using OpenAI's Triton language. This baseline targets the language model (LM) head during the decoding phase, implementing a custom INT4 Matrix-Vector (GEMV) multiplication kernel to directly read packed weights, unpack them on the fly, and accumulate the results without falling back to high-level PyTorch primitives.

1.4.1 Triton INT4 Unpacking Logic

The core of this optimization lies within the inner loops of the `_int4_gemv_kernel`. Because the weights are packed (8 INT4 weights per 32-bit unsigned integer), the kernel must unpack them at the bit level before multiplication.

```
    @triton.jit
    def _int4_gemv_kernel(...):
        # ... surrounding launch and grid setup ...
        for lane in tl.static_range(0, 8):
            k_idx = k0 + lane
            x_lane = tl.load(
                x_ptr + pid_m * stride_xm + k_idx * stride_xk,
                mask=valid_g & (k_idx < K),
                other=0.0,
            ).to(tl.float32)

```

```
=== Evaluation Result ===
Correctness: PASS
Correctness details:
  tolerances: atol=0.200, rtol=0.100
  fail_rate_tol: 0.05
  global: max_abs_diff=0.0723, ref_at_max_abs_diff=-2.6016, pred_at_max_abs_diff=-2.6738
  per regime:
    Reg Verdict    Mismatch  Elements  FailRate   MaxAbs   Ref@Max  Pred@Max
    R1  PASS              0    151936    0.0000   0.0215   -2.4707   -2.4922
    R2  PASS              0    151936    0.0000   0.0723   -2.6016   -2.6738
    R3  PASS              0    151936    0.0000   0.0527   -0.6953   -0.7480
    R4  PASS              0   1215488    0.0000   0.0723   -2.4434   -2.3711

Memory:
  max_memory_allocated: 7.91 GB
  max_memory_reserved: 10.66 GB
  per-regime peak memory:
    R1: allocated=7.91 GB, reserved=10.61 GB
    R2: allocated=7.91 GB, reserved=10.61 GB
    R3: allocated=7.91 GB, reserved=10.66 GB
    R4: allocated=7.91 GB, reserved=10.61 GB

Per-regime metrics:
    Reg Metric            Value      Unit    Better
    R1  Throughput        85.92     tok/s    higher
    R2  Throughput        31.41     tok/s    higher
    R3  TTFT            1981.92        ms     lower
    R4  TTFT            1608.17        ms     lower

Runtime:
  setup_time: 119.57 s
  total_timed_run: 9.16 s
```

Figure 3: Output logs confirming PASS for correctness and recording fused Baseline 3 metrics.

```
            # Extract 4-bit lane, cast to int8, and apply -8 offset
            w_lane = ((w_u32 >> (lane * 4)) & 0xF).to(tl.int8) - 8

            # Dequantize with scale and multiply by input activation
            acc += (w_lane.to(tl.float32) * s_vec) * x_lane
```

Inside the static, compile-time unrolled `lane` loop, the kernel isolates the specific 4-bit chunk of the packed `uint32` word (`w_u32`). It achieves this by bit-shifting right by `lane * 4` and masking with `0xF` (15). Because the quantization scheme stores the values with an offset (where 0 represents -8, and 15 represents 7), the unpacked value is cast to a signed 8-bit integer and shifted by subtracting 8. Finally, the true floating-point weight is reconstructed by casting to `float32` and multiplying by the group scale (`s_vec`), which is then multiplied by the scalar activation (`x_lane`) and added to the block accumulator (`acc`).

Fig. 4 shows the verification of the Baseline 4 evaluation script passing the correctness checks.

1.5 Baseline 5 (Custom Sparse Attention Triton Kernel)

The objective of Baseline 5 is to move the sparse attention computation entirely onto the GPU using a custom Triton kernel. Instead of generating a massive boolean mask and relying on PyTorch primitives (as in Baseline 1), this kernel computes the block-selection rules natively during the online softmax pass, discarding irrelevant key/value blocks on the fly to save VRAM bandwidth.

```
=== Evaluation Result ===
Correctness: PASS
Correctness details:
  tolerances: atol=0.200, rtol=0.100
  fail_rate_tol: 0.05
  global: max_abs_diff=0.0723, ref_at_max_abs_diff=-2.6016, pred_at_max_abs_diff=-2.6738
  per regime:
    Reg Verdict    Mismatch  Elements  FailRate   MaxAbs   Ref@Max  Pred@Max
    R1  PASS              0    151936    0.0000   0.0195    3.8105    3.7910
    R2  PASS              0    151936    0.0000   0.0723   -2.6016   -2.6738
    R3  PASS              0    151936    0.0000   0.0527   -0.6953   -0.7480
    R4  PASS              0   1215488    0.0000   0.0723   -2.4434   -2.3711

Memory:
  max_memory_allocated: 7.91 GB
  max_memory_reserved: 10.66 GB
  per-regime peak memory:
    R1: allocated=7.91 GB, reserved=10.61 GB
    R2: allocated=7.91 GB, reserved=10.61 GB
    R3: allocated=7.91 GB, reserved=10.66 GB
    R4: allocated=7.91 GB, reserved=10.61 GB

Per-regime metrics:
    Reg Metric            Value      Unit    Better
    R1  Throughput       118.53     tok/s    higher
    R2  Throughput        34.91     tok/s    higher
    R3  TTFT            1971.16        ms     lower
    R4  TTFT            1591.59        ms     lower

Runtime:
  setup_time: 122.09 s
  total_timed_run: 8.31 s
```

Figure 4: Output logs confirming PASS for correctness and recording Triton INT4 Baseline 4 metrics.

1.5.1 Triton Sparse Rule Translation

Inside the `_sparse_attention_kernel_single` function, one Triton program is responsible for a single query token. It loops over the key/value blocks and must evaluate the exact same causal, local, and dilation rules defined in Baseline 1 before loading the memory for that block.

```
    @triton.jit
    def _sparse_attention_kernel_single(...):
        # ... setup and query loading ...
        for t0 in tl.range(0, KV_LEN, BLOCK_SIZE):
            kv_block = t0 // BLOCK_SIZE
            causal_block = kv_block <= q_block
            keep_block = causal_block

            if MODE_IS_SPARSE:
                block_delta = q_block - kv_block
                local_keep = (block_delta >= 0) & (block_delta < LOCAL_BLOCKS)
                long_keep = (block_delta >= LOCAL_BLOCKS) & ((kv_block % DILATION) == h_mod)
                keep_block = keep_block & (local_keep | long_keep)

            # ... proceed to load KV and compute softmax only if keep_block is True ...
```

This logic directly mirrors the naive PyTorch implementation but operates at the scalar/block level within the GPU registers. The distance between the current query block and the loop's current key/value block is calculated (`block_delta`). If the delta falls within the `LOCAL_BLOCKS` threshold, it is kept. If it is older, the kernel checks if the

key/value block's modulo aligns with the current head's modulo (`h_mod`), preserving the sparse dilation stripe. The final `keep_block` boolean dictates whether the surrounding online softmax logic processes that tile.

Fig. 5 shows the verification of the Baseline 5 evaluation script passing the correctness checks.

```
=== Evaluation Result ===
Correctness: PASS
Correctness details:
  tolerances: atol=0.200, rtol=0.100
  fail_rate_tol: 0.05
  global: max_abs_diff=0.1426, ref_at_max_abs_diff=-2.4434, pred_at_max_abs_diff=-2.3008
  per regime:
    Reg Verdict    Mismatch  Elements  FailRate   MaxAbs   Ref@Max  Pred@Max
    R1  PASS              0    151936    0.0000   0.0176   -1.8457   -1.8633
    R2  PASS              0    151936    0.0000   0.0586   -2.2188   -2.2773
    R3  PASS              0    151936    0.0000   0.0469    2.3613    2.3145
    R4  PASS              0   1215488    0.0000   0.1426   -2.4434   -2.3008

Memory:
  max_memory_allocated: 7.91 GB
  max_memory_reserved: 10.66 GB
  per-regime peak memory:
    R1: allocated=7.91 GB, reserved=10.61 GB
    R2: allocated=7.91 GB, reserved=10.61 GB
    R3: allocated=7.91 GB, reserved=10.66 GB
    R4: allocated=7.91 GB, reserved=10.61 GB

Per-regime metrics:
    Reg Metric             Value      Unit    Better
    R1  Throughput        125.96     tok/s    higher
    R2  Throughput         27.81     tok/s    higher
    R3  TTFT             2945.96        ms     lower
    R4  TTFT             2424.78        ms     lower

Runtime:
  setup_time: 116.68 s
  total_timed_run: 10.99 s
```

Figure 5: Output logs confirming PASS for correctness and recording custom Triton Attention Baseline 5 metrics.

2 Optimization Writeup

This section details two custom optimizations implemented beyond the standard guided baselines, focusing on memory bandwidth reduction and GPU occupancy scaling. Table 1 summarizes the effects of these optimizations across the four evaluation regimes. The evaluation was done on all examples of all regimes.

Table 1: Optimization Results Comparison

Commit / Configuration	**R1 (tok/s) ↑**	**R2 (tok/s) ↑**	**R3 (ms) ↓**	**R4 (ms) ↓**
`baseline4` (Base)	114.15	35.14	2028.50	1695.83
`baseline4+flex_attention`	126.54	40.30	1432.22	1423.92
`baseline4+autotune`	129.62	35.55	1973.61	1617.51
`baseline4+flex_attention+autotune`	**130.15**	**40.85**	**1416.96**	**1368.23**

2.1 Optimization 1: On-the-Fly Fused Sparse Attention (`flex_attention`)

To optimize the attention computation, I replaced the manual dense mask generation (`_build_sparse_keep_mask`) and the heavily dense masked-attention function (`_masked_attention`) with PyTorch's native `flex_attention` API [1]. Also, I added `model._forward_with_mask = torch.compile(model._forward_with_mask)` within the `setup()` function to ensure the entire forward pass (including prefill) is compiled, allowing PyTorch to generate a custom fused Triton kernel for the sparse block mask on the fly. I tried this because Baseline 4 evaluates sparse attention by calculating full $N \times N$ dense matrices and masking out dropped positions with `-inf`, wasting massive VRAM bandwidth and FLOPS computing discarded values during long contexts (e.g., 4096 tokens in R2 and R3).

The primary correctness risk introduced by `flex_attention` is its strict reliance on `torch.compile`. If the logical rules within the `BlockMask` are unsupported or overly complex, the compiler might silently fall back to eager execution (crashing performance) or compile a numerically unstable FlashAttention kernel, risking failure against the reference tolerances [2]. I verified correctness and runtime by executing the local evaluation script, ensuring the `Correctness: PASS` flag remained intact, and confirming via output logs that the compiler successfully compiled the graph without triggering the un-fused `UserWarning`. As shown in Table 1, this optimization drastically improved the prefill-heavy regimes (R3 and R4), dropping R3 TTFT by nearly 600 ms.

2.2 Optimization 2: Triton Launch Parameter Autotuning for INT4 GEMV

To scale GPU occupancy, I added the `@triton.autotune` decorator to the `_int4_gemv_kernel`, testing multiple configurations of `BLOCK_N` (64, 128, 256) and `num_warps` (4, 8), cached via `key=['N', 'K']` [3]. In the host launch function, I replaced the static grid tuple with a lambda function (`lambda META: (m, triton.cdiv(n, META['BLOCK_N']), split_g)`) so the number of launched blocks scales dynamically with the autotuner's selection. I implemented this because Baseline 4 hardcoded `BLOCK_N=64` and `num_warps=1`. A single warp (32 threads) severely underutilizes the GPU's Streaming Multiprocessors (SMs), causing threads to stall while waiting for global memory reads. Increasing `num_warps` allows the hardware scheduler to hide memory latency by executing math on active warps while others wait for weights to load.

This optimization carries the risk of out-of-bounds memory writes or skipped computations if the dynamic `BLOCK_N` is not properly linked to the grid calculation. Furthermore, picking a block size that is too large could overflow the SM's shared memory (SRAM), causing compilation failures. I verified correctness by monitoring the `max_abs_diff` to ensure varying block sizes did not alter the FP16 accumulation precision enough to breach the 0.200 absolute tolerance, alongside achieving a standard `Correctness: PASS`. Because the GEMV kernel is only utilized when sequence length $M = 1$, this optimization massively accelerated the memory-bound decode phase (R1) with negligible impact on prefill (Table 1).

2.3 Optimization 3: Flex Attention + Triton Autotuning

Finally, I integrated both the PyTorch `flex_attention` compilation for the prefill/attention paths (Optimization 1) and the Triton `@autotune` configurations for the INT4 GEMV decode path (Optimization 2) into a single, unified baseline. I hypothesized that since Optimization 1 targets the prefill compute bottleneck and Optimization 2 targets the decode memory bandwidth bottleneck, they dominate different phases of the generation lifecycle and should combine perfectly.

The main risk involved in combining compiler-level optimizations is the introduction of subtle state integration bugs. For instance, if the dynamically autotuned Triton GEMV kernel alters the memory layout or precision of the hidden states passed into the KV cache, the aggressively compiled `flex_attention` kernel might read corrupted values, causing cascading deviations across long generation contexts. Verification on the local evaluator confirmed a `Correctness: PASS` was maintained despite stacking both dynamic PyTorch compilation and Triton autotuning. As Table 1 confirms, the optimizations worked synergistically. The unified model achieved the highest throughput

and the lowest latency across all regimes, strictly dominating all previous isolated baselines without introducing overhead regressions.

References

[1] PyTorch. *FlexAttention: Flexible and Efficient Attention Kernels in PyTorch*. `https://pytorch.org/docs/stable/nn.attention.flex_attention.html`. Accessed: 2026-04-06. 2024.

[2] PyTorch. *torch.compile — PyTorch Documentation*. `https://docs.pytorch.org/docs/stable/generated/torch.compile.html`. Accessed: 2026-04-06. 2026.

[3] Triton Authors. *triton.autotune — Triton documentation*. `https://triton-lang.org/main/python-api/generated/triton.autotune.html`. Accessed: 2026-04-06. 2024.