Gaps in Operator Fusion for Deep Learning Systems: The Trade-off Between Compilation and Execution of Unfused Operations

Leonardo Yukio Macedo Kamei
PUC-MG
Belo Horizonte, Brazil
leokamei1@gmail.com

Pedro Henrique Ramos Costa
PUC-MG
Belo Horizonte, Brazil
phramoscosta@gmail.com

Abstract

Operator fusion can reduce steady-state latency, but compiler work becomes relevant in short-lived inference workloads. We characterize native TVM, XLA, and PyTorch 2 (*TorchInductor*) stacks on ResNet-18/50, BERT, and GPT-2 using $K = 5$ cold processes per model and input, 10 warmups, and 50 timed executions per process. We use the amortized cost model $T(n) = T_C + n\,T_X$, where T_C is the backend-specific compilation-cost proxy, T_X the steady-state latency, and n the number of executions, to expose the regimes favored by each measured stack. For TorchInductor, an inference-only *Conv–BatchNorm* parameter fold applied as offline preprocessing reduced the mean compiler-cost proxy by 29.5–41.1% for ResNet-18 and 1.9–28.6% for ResNet-50; in nine of the ten inputs the 95% confidence intervals remain non-overlapping after adding the measured 100–120-ms preprocessing cost. Execution effects were smaller and model-dependent. An applicability audit of the BERT *LayerNorm–Linear* fold finds no legal instance in the evaluated post-LayerNorm encoder: the structural matcher performs zero rewrites, and an isolated cold-process control shows overlapping confidence intervals in every input. The benchmark reports measured system-level trade-offs, while neither the native-stack comparison nor the valid ResNet fold establishes a universal compiler pass ordering.

Keywords

Operator Fusion, Machine Learning Compilers, Compilation Time, Kernel Fusion, Performance Benchmarking

1 Introduction

The *operator fusion* technique has direct roots in classical compiler optimizations, such as *loop* fusion, historically used to increase parallelism and improve data locality [12]. In modern compilers for Machine Learning, particularly on GPUs, this technique plays a central role: by combining multiple operations into a single *kernel*, memory traffic is reduced, intermediate tensors are avoided, and the number of dynamic launches during execution is decreased [4, 8]. Systems such as XLA, TVM, TensorRT, and the TorchInductor backend adopt aggressive fusion to amortize CPU–GPU communication costs and consolidate operator sequences into larger, more efficient kernels [1, 4, 7].

Fusion is performed over computational/dataflow graphs or compiler intermediate representations. Deciding *what* and *how* to fuse is challenging: specific graph-partitioning formulations of operator fusion can be mapped to NP-complete problems, while classical loop-fusion formulations are likewise computationally difficult [3, 5]. For this reason, compilers use legality rules, heuristics, and cost models. Compilation cost is especially important in dynamic workloads and scenarios with few repetitions, where fixed compiler work can dominate perceived performance.

Many compiler evaluations emphasize steady-state latency, even though deployment decisions may also depend on compilation cost. Reporting both quantities under repeated cold processes makes explicit when additional fixed cost can be amortized. XLA [11], TVM [4], and TorchInductor (PyTorch 2) [1] use different frontend contracts, optimization pipelines, and code-generation strategies, resulting in different compilation- and execution-time profiles.

We evaluate this trade-off for native TVM, XLA, and TorchInductor implementations of ResNet-18/50, BERT, and GPT-2 under a common outer sampling protocol. For n executions, we use $T_b(n) = a_b n + b_b = T_C + n\,T_X$, where $b_b = T_C$ is the compilation-cost proxy and $a_b = T_X$ is steady-state latency (Section 4). Minimizing the measured lines gives a descriptive deployment envelope for this hardware and software stack. Because the backends use native model implementations, layouts, parameter-binding contracts, and compilation APIs, the comparison is system-level rather than an isolation of compiler quality. We additionally evaluate a legal `Conv--BatchNorm` fold as offline TorchInductor preprocessing.

Contributions:

- A repeated cold-process protocol and the amortized model $T(n) = T_C + n\,T_X$ for reporting compilation–execution trade-offs;
- A $K = 5$ system-level characterization of native XLA, TorchInductor, and TVM stacks on ResNet-18/50, BERT, and GPT-2;
- Artifact analysis (TVMScript, HLO, and Triton), a correctness-audited TorchInductor `Conv--BatchNorm` fold, and a BERT fold-applicability audit that finds no legal `LayerNorm--Linear` pair in the evaluated encoder.

2 Related Work

Operator fusion can be seen as the computation-graph analogue of classical *peephole optimization*. Fraser's machine-independent peephole optimizer characterized valid local rewrites over a sequence of machine instructions by describing each instruction's effect as a Register Transfer List, replacing a window of instructions by a single equivalent one whenever such a description existed [6]. Operator fusion plays the same role one level up: where peephole optimization rewrites a window of three-address-code instructions into an equivalent instruction, fusion rewrites a window of computation-graph operators into an equivalent operator that preserves the observable semantics.

Among graph-level approaches, *Optimus* minimizes memory accesses over a computation DAG through an offline dynamic-programming algorithm and an online buffer-budget variant [2]. DNNFusion combines data-flow analysis with fusion templates and a cost model, illustrating that legality and profitability are distinct concerns [8]. The XNNC study maps its directed-graph partitioning formulation to an NP-complete problem and reports that a backtracking solver did not finish the largest evaluated networks within a one-hour limit, whereas its greedy clustering alternatives terminated on the full suite [3]. Production compilers similarly combine restricted search, pattern rules, and backend-specific code generation [1, 4].

3 Theoretical Background

Consider a sequence of operators $x_{k+1} = f_k(x_k)$ and $x_{k+2} = f_{k+1}(x_{k+1})$, in which each f_i acts on tensors of a computation graph. We say that f_k and f_{k+1} are *fusible* when there exists a single operator h such that $h(x_k) = f_{k+1}(f_k(x_k))$ and the execution of h preserves the observable semantics of the original sequence. In this case, the compiler performs the transformation $(f_k, f_{k+1}) \Rightarrow h$, eliminating the materialization of intermediate tensors, reducing the number of kernel launches, and improving memory locality [4, 8].

As discussed in Section 2, this notion of fusibility is the computation-graph analogue of classical *peephole optimization*: h plays, for a window of graph operators, the role that an equivalent instruction plays for a window of three-address-code instructions. In ML models, however, operators handle multi-dimensional tensors and their definitions are often complex. We therefore distinguish kernel fusion from parameter folding. Equations (III) and (VIII) in Appendix A describe `BatchNorm+ReLU` fusion and inference-only `Conv--BatchNorm` reparameterization followed by ReLU. Equation (X) gives a conditional affine `LayerNorm--Linear` reparameterization; our BERT audit found no legal instance of this last pattern.

Kernel sequences may be left unfused because of legality, profitability, or backend restrictions [1, 8]:

- **Training mode.** A forward-only `BatchNorm` parameter fold is invalid in training because batch statistics and state updates remain dynamic. Training-time kernel fusion remains possible when an implementation preserves the required backward values and state semantics.
- **Graph uses.** A local reparameterization must account for every consumer of the rewritten value. Multiple consumers do not universally forbid kernel fusion, but they can make a single-consumer fold illegal.
- **Numerical precision.** Mixed-precision rewrites must preserve conversion and accumulation semantics within the accepted numerical tolerance.
- **Shapes and indexing.** Broadcasting, layout conversion, reindexing, or dynamic shapes can make a candidate illegal or unprofitable.
- **Operation type.** Differing memory-access patterns or synchronization requirements may prevent profitable execution in one generated kernel.
- **Backend restrictions.** External-library calls such as cuDNN or cuBLAS constrain which surrounding operations a compiler can incorporate into the same call [1, 11].

In every case, the fusion decision involves an explicit trade-off: investing more compilation time to reduce execution time, or compiling quickly with fewer fusions. This balance underlies the cost model adopted in this work and guides the comparison between backends.

4 Methodology

We compare TVM [4], XLA [11], and TorchInductor (PyTorch 2) [1] in a controlled environment, measuring a **backend-specific compilation-cost proxy**, the **average latency per execution**, and the **static structure of the generated artifacts**.

4.1 Experimental Environment

The repeated campaigns were run on Fedora 41, an AMD Ryzen 5 4600G CPU, 15.4 GiB of RAM, and an NVIDIA RTX 3050 with 8 GB of VRAM, using driver 580.105.08. The host `nvcc` compiler was CUDA 12.6 (V12.6.77). Transformer measurements ran in the archived container on host kernel 6.17.10; every JSON records the effective Python, framework, and platform versions.

Per-framework configuration. For the **TVM** experiments, we used Python 3.11.13, cuDNN version 9.1 (via PyTorch), TVM 0.22.dev0 (commit 3b60f1c), LLVM 20.1.8, and CUDA 12.6 integrated into the TVM backend.

For **XLA/TorchInductor**, the ResNet environment used Python 3.10.18. XLA used JAX/jaxlib 0.6.2 and the PJRT GPU runtime plugin (`jax-cuda12-pjrt` 0.6.2). Its CUDA plugin was built against cuDNN 9.8 and explicitly loaded the host cuDNN 9.11.0 at runtime; the packaged cuDNN 9.10.2 had produced intermittent convolution-plan failures on this machine. The repeated ResNet campaign uses XLA's default autotuner, records both build/runtime cuDNN versions, and retains a failed process only as retry metadata, never as a timing observation. The Transformer container uses Python 3.12.3 and Transformers 4.56.2: TorchInductor uses PyTorch 2.9.0+`cu128`, XLA uses JAX 0.6.2, and TVM 0.22.dev0 imports the PyTorch 2.5.1+`cu124` graph. The BERT fold-control audit in Table 7 uses the same pinned TorchInductor environment. Because compiler and model-library versions affect compilation cost and generated graphs, absolute times are compared only within the corresponding campaign and table.

The common outer conditions are FP32 tensor storage, the same GPU, inference mode, 10 warmups, 50 timed executions, explicit synchronization, and isolated cold processes. Backend-native layouts are retained: TorchInductor uses logical NCHW tensors in `channels_last` memory format, XLA uses NHWC, and TVM uses NCHW. The TVM environment is pinned to version `0.22.dev0`.

Accumulation precision. On Ampere, FP32 storage does not by itself determine the arithmetic used inside convolution and matrix multiplication, so Table 1 states the regime of every model/backend pair instead of assuming a single one. Both Transformer campaigns compute in true FP32: TorchInductor keeps PyTorch's default `float32_matmul_precision` of `highest` and XLA is pinned with `jax_default_matmul_precision` set to `highest`. The ResNet campaign

Table 1: Accumulation precision of each model/backend pair. FP32 is the only regime the three stacks can share, because the pinned TVM build has no TF32 tensor-core path.

Campaign	Model	TorchInductor	XLA	TVM
Native stacks	ResNet-18/50	TF32	TF32	FP32
	BERT, GPT-2	FP32	FP32	FP32
Fold (Conv–BN)	ResNet-18/50	TF32	—	—
Fold (LN–Linear)	BERT	FP32	—	—

instead leaves TorchInductor and XLA on the TF32 tensor-core path, which cuDNN enables for convolution by default. TVM is FP32 everywhere, and not by configuration: the pinned build exposes 83 CUDA tensor-core intrinsics, none of which takes TF32 or FP32 inputs, and no convolution rule at all in `dlight`. A uniform TF32 regime is therefore unreachable on the evaluated TVM path, which makes FP32 the only regime the three stacks can share; Section 6 states what this implies for the ResNet comparison.

Pinning XLA's precision required re-measuring its seven Transformer cells, so the XLA column of Tables 3 and 4 comes from a later session than the other two. Re-measuring the superseded XLA cell at `(1,64)` with the pin removed reproduced its previous means within 1%, which attributes the changes below to precision rather than to the session; re-measuring the TorchInductor cell unchanged reproduced execution within 1% but the compilation proxy only within 12%, an uncertainty small against the fourfold gap the table reports.

4.2 Models and Inputs

We evaluated **ResNet-18/50**, **BERT**, and **GPT-2**. Each ResNet input is written as `(N,C,H,W)`: batch size N, number of channels C, and spatial height and width $H \times W$. Inputs for ResNets (CNNs): `(1,3,224,224)`, `(16,3,224,224)`, `(64,3,224,224)`, `(1,3,512,512)`, `(1,3,1024,1024)`. BERT uses `(batch,seq_len)` pairs `(1,64)`, `(1,128)`, and `(8,128)`; GPT-2 uses `(1,16)`, `(1,128)`, `(8,128)`, and `(8,256)`.

TorchInductor and TVM use the same nominal `torchvision` ResNet definitions with `weights=None` and seed 0; the original and folded TorchInductor children start from identical state dictionaries, while TVM imports the PyTorch FX graph and binds its parameters. XLA uses the native `flaxmodels` ResNet definitions with `pretrained=None`, no embedded input normalization, logits output, PRNG seed 0, and variables closed over the JIT-compiled function. Inputs are seeded random tensors for TorchInductor/TVM and ones for XLA. Consequently, the experiment compares complete native software stacks for the same named architecture, shape, and hardware, under the per-campaign accumulation-precision regimes stated above; it does not establish numerical model equivalence or isolate the compiler while holding frontend graph, parameters, layout, and parameter-binding semantics constant.

For Transformers, TorchInductor and TVM start from the same Hugging Face base configurations, deterministic random weights, and input IDs; XLA uses the corresponding native Flax model. GPT-2 receives a fixed four-dimensional causal mask, avoiding a nested `vmap` masking path that PyTorch 2.5 cannot export. For TVM import only, the exported `aten._unsafe_view` alias is normalized to the equivalent supported `aten.reshape`; no tensor values or shapes change.

The three models cover complementary architectural regimes: the ResNets contribute the recurring `Conv--BatchNorm--ReLU` chains that are the canonical case for fusion in CNNs; BERT contributes an encoder graph dominated by matrix multiplications, linear projections, normalizations, and activations; and GPT-2 contributes the repetitive decoder-only flow of autoregressive generation.

4.3 Experimental Pipeline

The repeated workflow adopts six outer stages shared by TVM, XLA, and TorchInductor for both convolutional and Transformer models.

(1) Compilation. For each (model, input, backend) configuration, we perform $K = 5$ cold compilations in independent processes. TorchInductor's original and folded ResNet variants are also compiled in separate processes. TVM and XLA children compile only the optimized/JIT variant reported in the comparison, so an auxiliary variant cannot warm in-process compiler state first. Each process receives private TorchInductor, Triton, and JAX compiler-cache directories; persistent TorchInductor and JAX compilation caches are disabled. Model-weight files and the operating system page cache remain shared, and model loading is outside the timed interval. We report mean, sample standard deviation, and two-sided 95% confidence intervals over the five process means, and we only claim a difference between two configurations when their 95% confidence intervals do not overlap. The independently compiled process—not an execution from an already compiled process—is the experimental unit. The separate BERT fold audit follows the same $K = 5$ cold-process protocol.

(2) Warmup and execution + graph capture. Execution occurs in two phases: (i) a warmup phase for runtime state and autotuning; and (ii) synchronized timed executions. The benchmark additionally exports backend IR artifacts—optimized HLO (XLA), TVMScript/TIR (TVM), and generated Triton/Python wrappers (TorchInductor)—outside the steady-state timing window and collects structural call metadata when available.

(3) Analysis of kernel quantity and quality. From the captured IRs, a Python script parses the generated program tree (TVMScript, HLO, and Triton wrappers) and quantifies the call-like units of each backend, separating units generated internally from external library calls. This analysis describes, within each stack, the static fusion structure and the degree of delegation to optimized libraries, providing structural context for interpreting the performance results.

(4) Measurements and algebraic model. The timing metrics are a backend-specific compilation-cost proxy and the average synchronized latency T_X. In the ResNet campaign, each of the $K = 5$ isolated processes performs 10 warmups followed by 50 timed executions, and execution confidence intervals use Student's t distribution over the five process means. For **TVM**, the proxy times graph passes and `relax.build` after FX tracing/import. For **TorchInductor** and

XLA, it is estimated by

$$T_C = T_{\text{first}} - T_X,$$

where T_{first} is the backend's first compiled execution and T_X is steady-state latency. The estimator is applied within each process before aggregation. These operational windows follow the available backend APIs: they share the cold-process sampling design but do not include identical frontend work. Cross-backend T_C values and envelopes must therefore be interpreted as native-stack measurements under this harness, not as an isonomic comparison of compiler passes alone.

Algebraic model. We use the following amortized model for compilation followed by repeated execution:

$$T_b(n) = a_b\, n + b_b,$$

where b_b is the fixed cost of compilation and initial overhead, and a_b is the average latency, allowing us to infer the inflection point for different backends as a function of the number of repetitions n. Equivalently, writing $b_b = T_C$ (the compilation-cost proxy) and $a_b = T_X$ (per-execution latency), the model reads $T_b(n) = T_C + n\,T_X$; this is the framework we use throughout to compare backends.

The model $T_b(n) = a_b n + b_b$, together with its *lower envelope*—the curve that, for each value of n, selects the smallest cost among all backends—helps explain the crossover points observed between the different backends. Thus, the intersections of these functions indicate from which number of executions n each backend becomes more advantageous.

(5) Applying the fold and comparing regimes. In addition to the standard models, we evaluate a fold as offline preprocessing before TorchInductor graph capture. At inference, the valid ResNet transformation absorbs `BatchNorm` into the preceding `Conv`; the conditional Transformer derivation absorbs the affine part of `LayerNorm` into a subsequent `Linear`. The reparameterizations are algebraically exact up to floating-point rounding (Appendix A), but require all affected consumers to remain semantically compensated. A `LayerNorm` value also used by an identity residual edge cannot be absorbed only into one `Linear`.

Unlike operator fusion, which combines operations during code generation, the fold reparameterizes the model before graph capture. The fold tables report compiler cost after preprocessing and record transformation time separately; the deployment envelope adds the measured fold time to the folded intercept. Model loading remains outside both intervals, so neither corresponds to a full application-startup latency. On fixed model copies with nontrivial BatchNorm statistics, the folded outputs pass `allclose(rtol=1e-3,atol=1e-4)` with maximum absolute errors 6.676×10^{-6} (ResNet-18) and 2.441×10^{-4} (ResNet-50), and no BatchNorm modules remain.

For the folded configuration, $T_f(n) = a_f n + b_f$, where the envelope sets b_f to measured fold preprocessing plus the subsequent compiler-cost proxy. The crossover point for which $\Delta T = T_f(n) - T_b(n) = 0$, where $T_b(n)$ is the cost without fold, is

$$n_{\text{cross}} = \frac{b_b - b_f}{a_f - a_b}, \qquad \text{provided } a_f \neq a_b.$$

We report a positive value only when the two point-estimate lines cross for $n > 0$. When $b_f < b_b$ and $a_f \leq a_b$, fold dominates for every non-negative n and no finite crossover exists.

(6) `JSON` *construction.* For each (model, input, backend) configuration, including the folded models, we generate a `JSON` that consolidates the benchmark state: experiment identification, timing metrics, generated call-like units, and version metadata.

5 Evaluation

All timing tables below come from the repeated $K = 5$ cold-process campaigns described in Section 4.

Throughout, *internal units* are call-like units generated by the backend (TIR calls for TVM, Triton wrapper calls for TorchInductor, and HLO fusions for XLA), whereas *external units* are calls delegated to libraries such as `cuDNN` or `cuBLAS`. These static IR categories provide structural context but are not a common hardware-kernel metric: an HLO fusion, a TIR function, and a Python-wrapper call need not map one-to-one to dynamic GPU launches.

PyTorch and XLA load models from native libraries; TVM receives an FX import of the PyTorch model. Frontend representation and parameter binding affect the resulting census. We therefore use the IR figures to describe each stack internally and do not interpret a smaller count as proof of a better compiler.

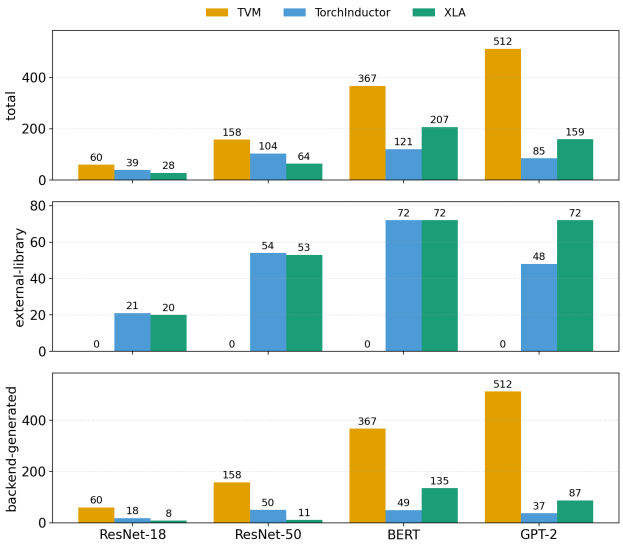

Figure 1: Static call-like units in the optimized artifacts of the four models, at the representative input of each one: (`1,3,224,224`) for the ResNets and (`1,128`) for BERT and GPT-2. Top: total, the sum of the two panels below. Middle: external-library units—`cuDNN`/`cuBLAS` calls for TorchInductor and custom calls for XLA. Bottom: backend-generated units—TIR calls for TVM, Triton launches for TorchInductor, and HLO fusions for XLA. The categories are backend-specific and are not a ranking or a direct count of dynamic GPU launches.

Figure 1 separates external-library calls from backend-generated units. The decomposition explains implementation structure—TVM generates every unit itself in the four models, whereas TorchInductor and XLA delegate the convolutions of the ResNets and the

matrix multiplications of the Transformers—but cannot establish fusion quality across unlike IRs. It also shows that the census grows with the architecture rather than with the compiler alone: TVM goes from 60 units on ResNet-18 to 512 on GPT-2, while TorchInductor stays between 39 and 121 in all four models. A normalized fusion comparison would require a shared input graph, parameter contract, and counting level.

5.1 Residual Networks

In ResNet-18/50, the three native stacks reveal structural differences in generated units and external-library delegation. Table 2 is generated directly from the complete repeated-run JSON grid; the artifact records the synchronization, cache-isolation, retry, and version metadata used for every cell.

TVM exhibits the largest static call-like census at the representative input. In its imported representation, convolution and batch normalization arrive as opaque units, so the lowering path keeps them as separate TIR functions in many blocks and has little visibility to combine them; even the `ReLU` frequently remains isolated, and only the shorter element-wise chains are collapsed. The outcome is a fragmented pipeline, with simple fusions and little structural reduction of the graph. This is a property of the evaluated import and lowering path, not a universal limitation of TVM.

TorchInductor relies on `cuDNN` for the convolutions, which prevents it from placing convolution, `BatchNorm`, and activation in a single generated kernel; instead, its inspected wrapper delegates the convolutions to external calls and fuses the surrounding `BatchNorm+ReLU` work into Triton kernels. This yields fewer launches and more compact graphs than TVM's, but relying on Triton for the element-wise operations and reductions makes code generation grow with the model: moving from ResNet-18 to ResNet-50 increases the generated code and the static wrapper-call census, which is associated with a larger compilation-cost proxy (Table 2).

In XLA's optimized HLO, convolution, bias, normalization, and activation appear together inside cuDNN custom calls. This more aggressive pattern yields a smaller HLO-level census and a more cohesive pipeline, and it is accompanied by the lowest compilation-cost proxy in all ten ResNet cells (Table 2). XLA also closes model variables over the JIT function, unlike TorchInductor's dynamic module-parameter contract. The timing difference is therefore a native-stack result and cannot be attributed solely to fusion or pass ordering.

5.2 Attention-Based Architectures (BERT)

The **BERT** model has a distinct fusion pattern that reflects its architecture centered on matrix multiplications [9], QKV (query–key–value) projections, normalizations, and activations.

TVM tends to produce smaller and more numerous units: its BERT artifacts contain 367 `call_tir` invocations at the imported Relax/TIR level. This is the same graph fragmentation observed in the ResNets, although BERT is more favorable to it than a CNN, because the encoder offers long element-wise chains that are easy to collapse. It remains the slowest stack in execution by a factor of at least two.

TorchInductor pursues fusions that minimize memory accesses: it delegates the matrix multiplications to external libraries such as cuBLAS [1]—72 external GEMM calls in the inspected wrapper, one per linear projection of the twelve encoder layers—and generates Triton code for what surrounds them. Only four distinct Triton kernels are emitted and then reused throughout the twelve layers—the embedding with its normalization, the residual addition with the following normalization, the projection with its activation, and the attention body—which reduces the number of launches at the cost of the highest compilation-cost proxy of the three. Generated-code summaries are identical across the five replicas of each cell.

XLA ends up with a larger static census than TorchInductor—207 call-like units at `(1,128)`—divided into 135 HLO fusions and 72 custom calls. Under the FP32 regime of this campaign, all 72 custom calls carry the target `__cublas$gemm`: the matrix multiplications that TorchInductor issues as external calls are delegated one-for-one here as well, so the two stacks make the same decision about the GEMMs and differ only in what they generate around them—12 custom fusions for the attention bodies, 25 reduction fusions for the LayerNorm chains, and 98 loop fusions for the element-wise work. XLA compiles in roughly a quarter of TorchInductor's time in all three inputs, but its execution does not reproduce that ordering: TorchInductor is 16.5% faster at `(1,64)` with non-overlapping process-level intervals, while at `(1,128)` and `(8,128)` the two are indistinguishable under the same criterion. As the batch grows, the GEMM cost comes to dominate execution in both stacks, so the difference in call-like census does not translate proportionally into latency. These orderings are results for the measured native stacks; different frontend representations and parameter contracts prevent attributing them solely to fusion. Table 3 reports variability over the five process means; all 250 per-iteration samples remain in the artifact.

5.3 Autoregressive Attention-Based Models (GPT-2)

The GPT-2 blocks follow a relatively linear decoder-only pattern—autoregressive attention and MLP, each preceded by LayerNorm and followed by GELU activation, repeated across all layers [10].

TVM does not incorporate the long epilogues of this architecture—softmax and LayerNorm—into larger units, because they arrive opaque in the imported IR; it does fuse the short ones, such as bias addition and, in some cases, GELU. Its imported GPT-2 IR consequently has a larger static `call_tir` census than its BERT import (512 against 367), and it is again the slowest in execution (Table 4). The fixed-shape import is executable for all four inputs after the export normalization of Section 4.

TorchInductor keeps the strategy it uses on BERT: it delegates the matrix multiplications and synthesizes Triton kernels that gather normalization, activation, and softmax into extended epilogues. This reduces the number of generated units, but the size and complexity of each kernel keep the compilation-cost proxy high, as on BERT. The five replicas in each cell produce identical structural summaries.

XLA again delegates every matrix multiplication—87 fusions against 72 `__cublas$gemm` custom calls at `(1,128)`, with the same division of fusions seen on BERT—although the two stacks do not decompose the GPT-2 projections into the same number of GEMMs:

Table 2: ResNet-18 and ResNet-50 native-stack compilation-cost proxy and execution time (mean ± sample sd over $K = 5$ independent process means, ms); each process uses 10 warmups and 50 synchronized executions. The lowest observed mean is marked in bold only when its marginal 95% process-level CI does not overlap either competitor. Compilation boundaries and native model contracts differ as described in Section 4.

		ResNet-18		ResNet-50	
Framework	Input (N,C,H,W)	Comp.	Exec.	Comp.	Exec.
TorchInductor	(1, 3, 224, 224)	11765 ± 1036	**1.88 ± 0.03**	24424 ± 1539	**3.78 ± 0.08**
	(16, 3, 224, 224)	14958 ± 907	11.86 ± 0.16	22519 ± 1094	31.09 ± 0.11
	(64, 3, 224, 224)	13116 ± 791	42.43 ± 0.13	21689 ± 586	112.51 ± 0.24
	(1, 3, 512, 512)	15544 ± 1364	4.85 ± 0.08	23221 ± 1411	11.85 ± 0.09
	(1, 3, 1024, 1024)	16502 ± 1110	15.03 ± 0.17	23834 ± 548	40.19 ± 0.14
TVM	(1, 3, 224, 224)	8104 ± 79	8.26 ± 0.07	13978 ± 28	20.01 ± 0.07
	(16, 3, 224, 224)	8333 ± 42	97.56 ± 0.06	14318 ± 130	270.32 ± 0.54
	(64, 3, 224, 224)	8216 ± 36	382.88 ± 1.19	14184 ± 47	1063.30 ± 1.27
	(1, 3, 512, 512)	7734 ± 34	33.01 ± 0.13	13564 ± 74	97.22 ± 0.22
	(1, 3, 1024, 1024)	7727 ± 44	128.08 ± 0.31	13577 ± 107	383.86 ± 0.37
XLA	(1, 3, 224, 224)	**1829 ± 12**	2.13 ± 0.04	**3503 ± 85**	4.19 ± 0.05
	(16, 3, 224, 224)	**2408 ± 27**	11.61 ± 0.05	**4800 ± 33**	**29.29 ± 0.13**
	(64, 3, 224, 224)	**4560 ± 100**	**38.71 ± 0.12**	**9352 ± 121**	**105.22 ± 0.26**
	(1, 3, 512, 512)	**2599 ± 54**	**4.23 ± 0.02**	**4013 ± 27**	**10.38 ± 0.04**
	(1, 3, 1024, 1024)	**2663 ± 76**	**12.94 ± 0.01**	**5011 ± 73**	**34.54 ± 0.02**

Table 3: BERT — compilation-cost proxy and execution time (mean ± sample sd over $K = 5$ independent process means, ms); each process uses 10 warmups and 50 synchronized executions.

Framework	Input	Comp.	Exec.
TorchInductor	(1, 64)	9076 ± 259	4.95 ± 0.06
	(1, 128)	9033 ± 262	8.07 ± 0.06
	(8, 128)	10466 ± 759	41.76 ± 0.03
TVM	(1, 64)	8120 ± 527	10.83 ± 0.12
	(1, 128)	7892 ± 48	22.25 ± 0.06
	(8, 128)	8259 ± 69	134.37 ± 0.35
XLA	(1, 64)	2401 ± 72	5.77 ± 0.09
	(1, 128)	2360 ± 44	8.02 ± 0.20
	(8, 128)	2779 ± 109	41.99 ± 0.31

Table 4: GPT-2 — compilation-cost proxy and execution time (mean ± sample sd over $K = 5$ independent process means, ms); each process uses 10 warmups and 50 synchronized executions.

Framework	Input	Comp.	Exec.
TorchInductor	(1, 16)	9389 ± 510	3.20 ± 0.11
	(1, 128)	8939 ± 187	7.29 ± 0.04
	(8, 128)	9677 ± 242	39.79 ± 0.15
	(8, 256)	9621 ± 148	82.78 ± 0.10
TVM	(1, 16)	6720 ± 250	5.55 ± 0.13
	(1, 128)	6687 ± 281	23.69 ± 0.37
	(8, 128)	6907 ± 228	149.13 ± 1.29
	(8, 256)	6540 ± 14	415.13 ± 0.34
XLA	(1, 16)	1729 ± 99	5.03 ± 0.10
	(1, 128)	1854 ± 29	7.65 ± 0.06
	(8, 128)	2555 ± 15	42.24 ± 0.05
	(8, 256)	2629 ± 11	86.24 ± 0.15

TorchInductor issues 48 external calls against XLA's 72. XLA compiles in roughly a quarter of TorchInductor's time in all four inputs, and here execution inverts that ordering outright: TorchInductor is faster in all four with non-overlapping intervals, by 57% at `(1,16)` and by 4–6% elsewhere. The static census therefore does not predict which stack runs faster—XLA emits more units and compiles them far more cheaply, yet matches TorchInductor's latency in no input. These process-level results characterize the measured stacks and do not establish a universal compiler ranking. Across BERT and GPT-2, all 21 backend/input cells contain five cold-process observations and 250 synchronized execution samples.

5.4 Fold

Table 5: ResNet-18 (TorchInductor) — effect of folding BN into Conv. Compilation is mean ± sample sd across $K = 5$ cold processes; execution entries are pooled means (ms).

Input	Comp. w/o fold	Comp. w/ fold	Δ Comp.	Exec. w/o fold	Exec. w/ fold	Δ Exec.
(1,3,224,224)	11765 ± 1036	6929 ± 235	−41.1%	1.88 ± 0.03	1.86 ± 0.03	−0.9%
(16,3,224,224)	14958 ± 907	9026 ± 214	−39.7%	11.86 ± 0.16	11.79 ± 0.10	−0.6%
(64,3,224,224)	13116 ± 791	9246 ± 215	−29.5%	42.43 ± 0.13	42.26 ± 0.14	−0.4%
(1,3,512,512)	15544 ± 1364	10201 ± 1132	−34.4%	4.85 ± 0.08	4.80 ± 0.10	−1.0%
(1,3,1024,1024)	16502 ± 1110	10187 ± 328	−38.3%	15.03 ± 0.17	14.89 ± 0.13	−0.9%

Table 6: ResNet-50 (TorchInductor) — effect of folding BN into Conv. Compilation is mean ± sample sd across $K = 5$ cold processes; execution entries are pooled means (ms).

Input	Comp. w/o fold	Comp. w/ fold	Δ Comp.	Exec. w/o fold	Exec. w/ fold	Δ Exec.
(1,3,224,224)	24424 ± 1539	17427 ± 661	−28.6%	3.78 ± 0.08	3.71 ± 0.13	−1.8%
(16,3,224,224)	22519 ± 1094	19700 ± 475	−12.5%	31.09 ± 0.11	30.63 ± 0.14	−1.5%
(64,3,224,224)	21689 ± 586	21280 ± 188	−1.9%	112.51 ± 0.24	110.15 ± 0.08	−2.1%
(1,3,512,512)	23221 ± 1411	19435 ± 295	−16.3%	11.85 ± 0.09	11.39 ± 0.10	−3.9%
(1,3,1024,1024)	23834 ± 548	21031 ± 273	−11.8%	40.19 ± 0.14	38.91 ± 0.13	−3.2%

Fold (ResNets). Applying the offline fold reduces the mean compiler-cost proxy in all ten ResNet inputs (Tables 5 and 6): 29.5%–41.1% for ResNet-18 and 1.9%–28.6% for ResNet-50. The measured transformation itself takes 100–106 ms for ResNet-18

and 114–120 ms for ResNet-50. Under the criterion of Section 4, the 95% confidence intervals over the five process means do not overlap in nine of the ten inputs, and the same nine remain separated after adding the preprocessing cost. The exception is ResNet-50 at `(64,3,224,224)`, where the mean reduction is only 409 ms (−1.9%) and the intervals overlap. Execution effects are smaller: no ResNet-18 interval separates, whereas ResNet-50 has non-overlapping speedups at `(16,3,224,224)`, `(64,3,224,224)`, `(1,3,512,512)`, and `(1,3,1024,1024)`. These comparisons are between independently compiled variants; the 250 within-process timing samples are not treated as independent compilations.

The structural evidence explains the compilation result without implying fewer runtime launches. In ResNet-18, the generated wrapper retains 18–19 Triton and 21 external calls after folding, while generated-code size falls by 38–44% across repetitions. In ResNet-50, wrapper calls remain 50 Triton and 54 external calls, the number of distinct named Triton kernels falls from 21 to 19, and code size falls by about 37%. Fifteen additional anonymous ResNet-50 autotuning stubs are excluded from runtime-call counts. Thus, the supported mechanism is reduced code-generation volume; attributing the variation across inputs to a particular compiler subsystem would require an additional ablation.

Table 7: BERT (TorchInductor) fold-applicability audit. Base and fold-control are identical because the post-LayerNorm residual graph has no safe LayerNorm→Linear fold. Compilation is mean ± sample sd across $K = 5$ isolated cold processes; execution is pooled mean ± sd (ms).

Input	Comp. base	Comp. control	Δ Comp.	Exec. base	Exec. control	Δ Exec.
(1,64)	10445 ± 470	10305 ± 445	−1.3%	4.925 ± 0.313	4.978 ± 0.213	+1.1%
(1,128)	10053 ± 169	9938 ± 169	−1.1%	8.046 ± 0.239	8.049 ± 0.253	+0.0%
(8,128)	10546 ± 368	10622 ± 375	+0.7%	40.961 ± 0.810	41.049 ± 1.010	+0.2%

Fold applicability (BERT).. Equation (X) in Appendix A is exact for a true single-consumer `LayerNorm→Linear` chain. The standard Hugging Face `BertModel` used here is instead post-LayerNorm: each encoder LayerNorm output is consumed both by subsequent projections and by an identity residual path. Absorbing (γ, β) only into the projections would therefore change the residual and the model's inference function. No legal pair is present.

We audited the original BERT scripts and re-ran a conservative fold-control that deep-copies the same graph but performs zero rewrites. The structural matcher finds zero adjacent `LayerNorm--Linear` siblings in this model; nevertheless, the superseded harness compiled the base and the unchanged "folded" copy sequentially in one process, allowing the second compilation to reuse compiler state. In the corrected experiment, base and control reproduce the submitted BERT model/input protocol but are compiled in distinct cold processes with identical seeds. All five artifacts per variant have the same structural census and code size; the 95% CIs overlap for both compilation and execution in every input (Table 7). No value from that superseded run is used in this paper. Both variants load the pretrained `bert-base-uncased` checkpoint, whereas Table 3 reports the randomly initialized configuration of the Transformer campaign; since `BertConfig()` reproduces the `bert-base-uncased` dimensions exactly, the two differ in weight values but not in shapes, arithmetic volume, or emitted kernels. This audit is measured in the same FP32 regime as that campaign, and its execution means agree with the corresponding TorchInductor column to within 2%—a consistency check across two independently written harnesses. A future BERT fold experiment must target an architecture with a genuine single-consumer LayerNorm–Linear edge or explicitly compensate every residual consumer.

By contrast, the valid ResNet tests show that offline `Conv--BatchNorm` preprocessing lowers the TorchInductor compilation-cost proxy even after its own cost is added. This demonstrates a guarded frozen-weight specialization opportunity for the evaluated inference path, not that TorchInductor's internal passes execute in an incorrect order. Under the $K = 5$ point estimates, the folded variant has both a lower mean fixed cost—already including the measured preprocessing—and no higher mean execution slope in all ten inputs, so the crossover n_{cross} of Section 4 is not finite for any of them: fold dominates from $n = 0$ onward.

Execution-time effects are small and uncertain in ResNet-18, showing that **longer compilation does not, by itself, imply lower latency**. In ResNet-50, the folded point estimate is lower in every input, and in four of them the confidence intervals do not overlap. This motivates reporting both fixed cost and steady-state latency.

Taking the three architectures together, each stack keeps a recognizable strategy: TVM works at a finer granularity, with more units and more conservative fusions, while TorchInductor and XLA both delegate the heavy arithmetic and differ in what they generate around it. In the ResNets the contrast is sharpest: TorchInductor emits one unit for the convolution, delegated to the library, and separate Triton units for what follows it, whereas XLA keeps that chain inside a single delegated call. The offline fold acts precisely on the first half of that contrast, removing the `BatchNorm` from the graph that TorchInductor still has to generate code for. We analyzed a single transformation rather than the full pass pipeline, and the benchmark can be extended to other passes and models.

6 Proposed Benchmark

We developed a benchmark in which the user chooses the model and its input—(N, C, H, W) for the CNNs and (batch, sequence length) for the Transformers—and the system measures the backend-specific compilation-cost proxy and synchronized inference latency. It reports (i) the measured native stack with the lowest point-estimate total cost for a requested n, and (ii) the corresponding ranges. These recommendations are conditional on the recorded machine, versions, flags, and native model contracts.

The benchmark standardizes the outer sampling conditions—named architecture, input shape, GPU, inference mode, $K = 5$ isolated cold processes, 10 warmups, 50 synchronized iterations, and JSON serialization—and records the accumulation-precision regime of each campaign rather than forcing a single one, for the reason given in Section 4. It deliberately retains each backend's native model, layout, parameter binding, and compilation API. The resulting recommendation therefore applies to the measured software stacks; it must not be read as a controlled compiler-only comparison.

One consequence deserves to be stated explicitly. Because the evaluated TVM path offers no tensor-core route, the ResNet tables place TF32 TorchInductor and XLA against FP32 TVM, and their execution columns are not a precision-controlled comparison. The measured gap there—TVM is 4.4× to 11.1× slower than the faster of its two competitors across the ten cells—is well beyond what the TF32/FP32 difference alone can produce, so the ordering itself is not in question; but the ResNet discussion of Section 5 is deliberately confined to compilation cost and static structure, and no ResNet execution ratio should be read as a compiler-quality measurement. The Transformer tables carry no such caveat, since the three stacks share the FP32 regime there.

Report (`JSON`) and quick reading. The benchmark generates a `JSON` report with the experiment context, raw process/timing measurements, failed XLA attempts, IR metadata, and the point-estimate $T_b(n)$ recommendation. The artifact documentation gives the serialized field names.

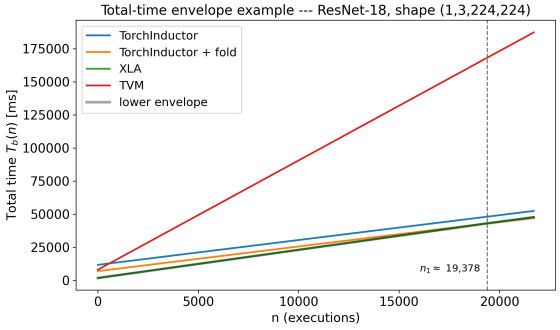

Figure 2: Example benchmark plot using the $K = 5$ point estimates: lines $T_b(n) = a_b n + b_b$ per measured configuration, with the lower envelope and crossover points. For ResNet-18 at `(1,3,224,224)`, XLA has the lowest point-estimate cost up to $n \approx 19{,}378$ and folded TorchInductor thereafter; TVM and the original TorchInductor variant never lie on this lower envelope. The folded intercept includes measured preprocessing.

How to read Figure 2. Each line starts at the backend-specific fixed-cost proxy b_b, and its slope is the measured per-execution latency a_b. The lower envelope highlights the lowest point estimate in each range of n; dashed lines mark crossovers. It is an illustration of the measured stacks, not an uncertainty-aware compiler ranking.

7 Conclusion

This work applied a repeated cold-process protocol to native TorchInductor, XLA, and TVM stacks on ResNet-18/50, BERT, and GPT-2, and used the lower envelope of $T_b(n) = a_b n + b_b$ as a deployment-oriented summary. The five-process estimates show that the preferred measured stack can change with n: fixed compilation cost matters at small n, whereas the smallest measured steady-state slope determines the outcome for sufficiently large n. Static fusion or call count alone does not predict that ordering. The cross-backend results characterize complete native stacks and do not isolate compiler quality because frontend graphs, layouts, parameter binding, and timed compilation boundaries differ.

Limitations. Although five independent cold compilations allow process-level uncertainty estimates, $K = 5$ remains small and all measurements come from one machine. The ResNet campaign also had to load the host cuDNN 9.11.0 to avoid the intermittent convolution-plan failures described in Section 4, and the repeated grid records any remaining failed attempt. For the 1024^2 inputs, XLA's autotuner also rejects candidate plans whose workspaces exceed the 8-GB GPU, so its selected plan is hardware-memory constrained. The Transformer campaign uses random initial weights and fixed shapes in the archived container; its absolute values therefore apply to that recorded stack. Future work should increase K, diversify hardware/software stacks, use trained checkpoints, and align frontend graphs and compilation boundaries.

Another limitation lies in the linearity assumption of $T_b(n) = a_b n + b_b$ itself: by assuming a constant per-batch cost and all of the fixed cost paid in the first batch, it abstracts away cache effects, autotuning, and recompilations triggered by dynamic shapes, which may make the effective behavior piecewise rather than linear. The coefficients should be read as summary statistics of the measured range of n, not as predictions for any usage regime. Future work can report goodness-of-fit metrics and investigate piecewise models.

Beyond the comparison between backends, this work also evaluated fold transformations in the **TorchInductor** graph. In ResNet-18/50, folding `BatchNorm` into `Conv` reparameterizes the convolution's weights and biases and removes the explicit `BatchNorm` in `eval` mode. This does not turn `Conv+BatchNorm+ReLU` into a single kernel—unlike XLA, which often delegates the whole chain to one custom call—but it reduces the volume of generated code, since reparameterizing the `Conv` is much cheaper than generating a separate Triton kernel for `BN+ReLU`.

For attention-based models, the analogous `LayerNorm+Linear` reparameterization is valid only when every use of the affine LayerNorm output can be absorbed or compensated. The post-LayerNorm BERT encoder evaluated here violates this precondition because the same value also follows an identity residual edge. Its isolated control experiment consequently performs no rewrite and shows overlapping confidence intervals. This negative result narrows the optimization opportunity: an automatic compiler pass must prove the consumer and residual conditions rather than matching only adjacent operator names.

A key observation is that the evaluated TorchInductor path does not perform the valid ResNet fold automatically under its current parameter contract. Offline folding produces a fixed-cost reduction with non-overlapping intervals in nine of ten ResNet inputs and comparable execution gains in four ResNet-50 inputs. This motivates guarded frozen-weight specialization with explicit legality checks, while leaving open whether automatic specialization is profitable under other contracts.

As a direct next step, a compiler pass could (i) specialize frozen `BatchNorm` parameters into `Conv` in `eval()` mode, (ii) request a convolution–activation epilogue when supported, and (iii) fall back safely otherwise. Its evaluation must include transformation overhead and dynamic launch profiling; the present result establishes lower code-generation volume, not fewer runtime launches.

Artifact Availability

The artifact provides the benchmark source code, pinned Python environments, a GPU-enabled Docker image recipe, correctness and smoke tests, the complete repeated cold-process ResNet and Transformer JSON grids, documented compiler-IR outputs, and scripts that regenerate the statistical analyses, tables, and plots. It is archived under the MIT License at https://doi.org/10.5281/zenodo.21892855; the development repository is mirrored at https://github.com/kameczera/compilers-benchmark. The README documents requirements, expected outputs, validation, resource costs, and the commands for both a short functional check and the complete experiment.

Acknowledgements

The authors used **ChatGPT**, a large language model from OpenAI, as a support tool during the preparation of this paper, restricted to the following purposes: refining the writing and checking textual coherence; reviewing the algebraic derivations of the kernel fusions in Appendix A; and assisting in writing and debugging portions of the benchmark code used in the experiments. All technical, methodological, and analytical content is the authors' own and was reviewed and validated by the authors. This disclosure follows the SBLP generative-AI policy and the corresponding ACM, IEEE, and Springer guidelines.

A Derivation of the Analyzed Fusion and Fold Transformations

This appendix distinguishes one kernel-fusion expression from two conditional parameter-folding derivations. Throughout, $\odot$ denotes the element-wise (Hadamard) product, broadcast along the channel dimension when the operands' shapes differ.

BatchNorm + ReLU

This widely known fusion combines a batch normalization (`BatchNorm`) with a following rectified linear unit (`ReLU`). Let $X \in \mathbb{R}^{N\times C\times H\times W}$ be the activation tensor (batch, channels, spatial dimensions); $\mu, \sigma^2 \in \mathbb{R}^C$ the per-channel estimated mean and variance; $\gamma, \beta \in \mathbb{R}^C$ the learnable scale and shift; and $\varepsilon > 0$ the numerical-stability constant. `BatchNorm` computes

$$\text{(I)} \qquad Y = \frac{X-\mu}{\sqrt{\sigma^2+\varepsilon}} \odot \gamma + \beta. \tag{1}$$

The `ReLU` is then applied element-wise to the output:

$$\text{(II)} \qquad Z = \max(0, Y). \tag{2}$$

An unfused realization may launch (I) and (II) separately, whereas a fused implementation computes both in one kernel:

$$\text{(III)} \qquad y = \max\left(0,\ \gamma \cdot \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} + \beta\right). \tag{3}$$

Modern deep-learning compilers cannot fuse every such pair profitably or legally, for the reasons discussed in Section 3.

Conv + BatchNorm + ReLU

In computer vision, the `Conv--BatchNorm--ReLU` chain contains two distinct opportunities: at inference, a fixed `BatchNorm` can be folded into the preceding convolution parameters $(\tilde{W}, \tilde{b})$, and, separately, a backend may implement `ReLU` as a convolution epilogue. The evaluated transformation performs the parameter fold only.

Formally, let $X \in \mathbb{R}^{N\times C_{in}\times H\times W}$ be the input and $W \in \mathbb{R}^{C_{out}\times C_{in}\times K_H\times K_W}$, $b \in \mathbb{R}^{C_{out}}$ the convolution parameters. The pre-activation output is

$$\text{(IV)} \qquad U_{n,c,h,w} = \sum_{c'=1}^{C_{in}} \sum_{r,s} W_{c,c',r,s}\, X_{n,c',h+r,w+s} + b_c. \tag{4}$$

`BatchNorm` at inference, with parameters $\mu, \sigma^2, \gamma, \beta \in \mathbb{R}^{C_{out}}$ and $\varepsilon > 0$, is

$$\text{(V)} \qquad Y_{n,c,h,w} = \frac{U_{n,c,h,w}-\mu_c}{\sqrt{\sigma_c^2+\varepsilon}}\,\gamma_c + \beta_c. \tag{5}$$

The `ReLU` (II) is then applied element-wise. We absorb the `BatchNorm` into the convolution parameters by defining

$$\text{(VI)} \quad \tilde{W}_{c,c',r,s} = \frac{\gamma_c}{\sqrt{\sigma_c^2+\varepsilon}}\, W_{c,c',r,s}, \tag{6}$$

$$\text{(VII)} \quad \tilde{b}_c = \frac{\gamma_c}{\sqrt{\sigma_c^2+\varepsilon}}\,(b_c-\mu_c) + \beta_c. \tag{7}$$

Substituting into Y and applying `ReLU` yields an equivalent expression that a backend may implement as one convolution–activation operation:

$$\text{(VIII)} \qquad Z_{n,c,h,w} = \max\left(0,\ \sum_{c'=1}^{C_{in}} \sum_{r,s} \tilde{W}_{c,c',r,s}\, X_{n,c',h+r,w+s} + \tilde{b}_c\right). \tag{8}$$

The benchmark guarantees only the reparameterization by $(\tilde{W}, \tilde{b})$; whether Equation (VIII) becomes one runtime kernel is backend-dependent.

LayerNorm (Affine Part) + Linear

Conditionally, the affine part of `LayerNorm` (LN) can be reparameterized into every subsequent `Linear` consumer. For an activation vector $h \in \mathbb{R}^d$, write the LN as $\tilde{h} = \gamma \odot \hat{h} + \beta$, where $\hat{h} = \mathrm{Norm}(h)$ is centered and normalized by the mean and variance of h, and $\gamma, \beta \in \mathbb{R}^d$ are the learnable scale and shift. For a `Linear` layer with weights $W \in \mathbb{R}^{m\times d}$ and bias $b \in \mathbb{R}^m$, expanding $y = W\tilde{h} + b = W(\gamma \odot \hat{h} + \beta) + b$ gives

$$\text{(IX)} \qquad y = W\,\mathrm{diag}(\gamma)\,\hat{h} + (W\beta + b). \tag{9}$$

Defining $W' = W\,\mathrm{diag}(\gamma)$ and $b' = W\beta + b$, the `LN+Linear` composition becomes

$$\text{(X)} \qquad y = W'\,\mathrm{Norm}(h) + b'. \tag{10}$$

Absorbing (γ, β) is thus legal only if every use of the affine output is removed or compensated, which the post-LayerNorm BERT evaluated here violates through its residual uses: Equation (X) is a conditional derivation, not a BERT optimization result.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, 929–947. doi:10.1145/3620665.3640366

[2] Xuyi Cai, Ying Wang, and Lei Zhang. 2022. Optimus: An Operator Fusion Framework for Deep Neural Networks. *ACM Transactions on Embedded Computing Systems* 22, 1 (2022), 26 pages. doi:10.1145/3520142

[3] Michael Canesche, Vanderson Martins do Rosario, Edson Borin, and Fernando Magno Quintão Pereira. 2025. Fusion of Operators of Computational Graphs via Greedy Clustering: The XNNC Experience. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. ACM, 117–127. doi:10.1145/3708493.3712689

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[5] Alain Darte. 1999. On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 149–157. Cat. No. PR00425.

[6] Christopher W. Fraser. 1979. A Compact, Machine-Independent Peephole Optimizer. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 1–6. doi:10.1145/567752.567753

[7] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, Compiled. In *TensorFlow Dev Summit*. https://www.tensorflow.org/xla

[8] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 883–898. doi:10.1145/3453483.3454083

[9] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. 2021. Demystifying BERT: Implications for Accelerator Design. In *IEEE International Symposium on Workload Characterization (IISWC 2021)*. IEEE, 109–120. doi:10.1109/IISWC53511.2021.00019

[10] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI Technical Report* (2019).

[11] Daniel Snider and Ruofan Liang. 2023. Operator Fusion in XLA: Analysis and Evaluation. https://arxiv.org/abs/2301.13062. arXiv:2301.13062.

[12] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 30–44.