Stack Wars: An Empirical and IR-Level Comparison of JAX and PyTorch via OpenXLA

1st Albert Cao
University of Michigan
Ann Arbor, Michigan
caoalb@umich.edu

2nd Vishnu Kannan
University of Michigan
Ann Arbor, Michigan
vishnuka@umich.edu

3rd John Xie
University of Michigan
Ann Arbor, Michigan
johnxie@umich.edu

4th Eric Zhao
University of Michigan
Ann Arbor, Michigan
zhaoeric@umich.edu

***Abstract*—JAX and PyTorch are two popular deep learning frameworks built on fundamentally different execution models. JAX adopts a functional, side-effect-free paradigm for ahead-of-time tracing and compilation, where as PyTorch uses dynamic eager execution with compilation layered on top. While PyTorch traditionally compiles through TorchInductor and JAX through XLA, both can target the same OpenXLA compiler backend via the StableHLO intermediate representation (IR). This shared backend enables us to perform an empirical and IR-level comparison of both frameworks that isolates the impact of frontend design choices from backend compiler behavior. We evaluate both frameworks on a set of microbenchmarks and end-to-end models, measuring compilation time and steady-state performance. We complement these metrics with a qualitative analysis of the emitted StableHLO graphs, in an effort to understand how IR differences translate to performance differences. Overall, we find that JAX achieves faster steady-state performance and better memory efficiency on complex workloads, while PyTorch incurs longer warm-up phases and higher memory usage. While JAX's superior performance is somewhat expected given its native XLA integration, our findings emphasize the importance of frontend and IR design on end-to-end performance.**

I. Introduction

Deep learning workloads rely heavily on software frameworks that translate high-level model definitions into efficient execution on specialized hardware such as GPUs and TPUs [1], [2]. Among these frameworks, PyTorch and JAX have emerged as the popular choices for research [3], [4].

Although PyTorch and JAX expose similar functionality to the user, their internal execution models differ substantially. PyTorch executes operations eagerly by default, with compilation introduced as an additional layer to recover static structure from dynamic execution [3], [5]. In contrast, JAX is designed around pure functions and ahead-of-time tracing that produces representations well-suited for compiler optimization [4], [6]. Recent developments in the OpenXLA ecosystem now enable both frameworks to target StableHLO, a common IR for high-level tensor operations. This convergence raises the following question: if JAX and PyTorch can compile to the same IR and run on the same hardware, why do performance differences persist?

This paper seeks to answer this question by presenting an empirical and IR-level comparison of JAX and PyTorch under controlled conditions using the OpenXLA compiler stack. We make the following contributions: (1) we establish a controlled experimental methodology that isolates frontend effects by compiling both frameworks to identical StableHLO IR, (2) we measure and compare compilation overhead, steady-state performance, and memory efficiency across microbenchmarks and end-to-end models, and (3) we analyze the structural differences in emitted StableHLO graphs to explain observed performance variations and potentially understand how frontend and IR design decisions influence downstream optimization.

II. Background and Related Work

A. PyTorch Compilation

PyTorch was originally designed around eager execution, executing operations immediately as they are called in Python. PyTorch 2.0 introduced torch.compile and TorchInductor [5], which leverage Python's frame evaluation API to capture computation graphs dynamically and compile them. TorchInductor generates Triton or C++ code for CUDA and CPU targets, focusing on kernel fusion and memory optimization.

B. JAX and XLA

JAX was designed from the ground up for high-performance numerical computing with compilation as a first-class citizen [6]. The core abstraction in JAX is the pure function transformation: jax.jit traces a Python function to produce an intermediate representation that is directly lowered to XLA HLO (High-Level Operations). XLA is Google's domain-specific compiler for linear algebra that performs aggressive optimizations including operation fusion, layout optimization, and memory planning. Because JAX enforces functional purity and uses ahead-of-time tracing, it naturally produces static graphs that expose optimization opportunities to XLA. This tight integration has made JAX the framework of choice for many large-scale training workloads, particularly on TPUs.

C. OpenXLA and StableHLO

OpenXLA is an open-source initiative to make XLA compilation infrastructure available across frameworks and hardware backends. A key component of OpenXLA is StableHLO, a portability layer that defines a stable, versioned operation set for expressing high-level tensor programs. StableHLO serves as a common IR that frameworks can target, decoupling frontend evolution from backend compiler development. Both

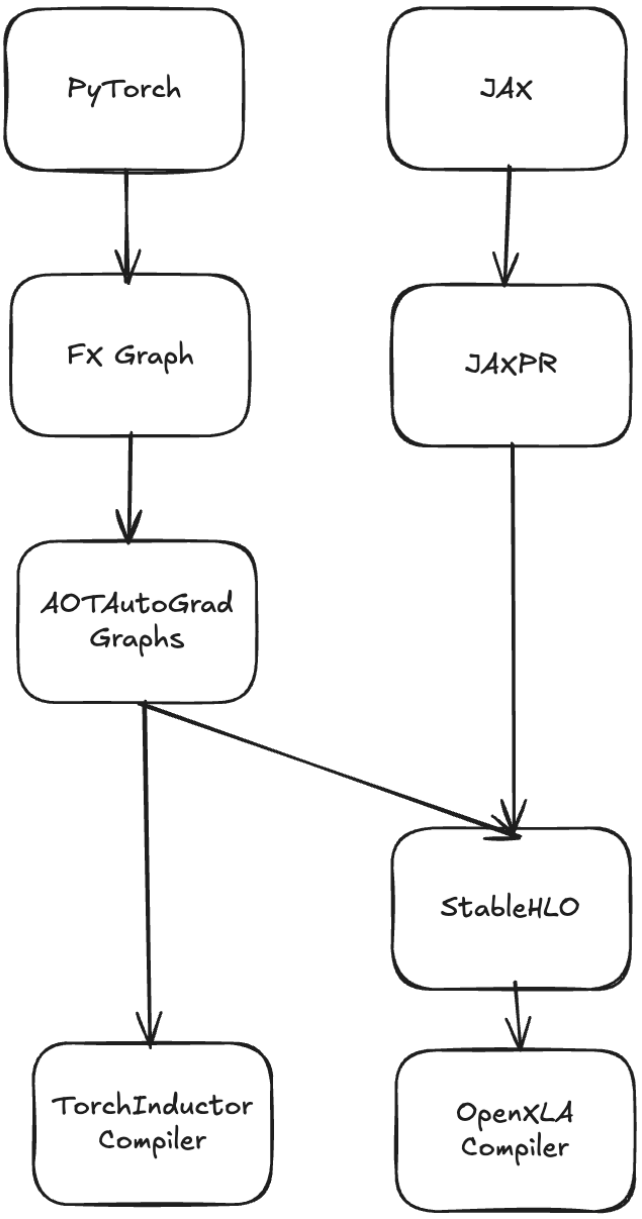


Fig. 1: This diagram shows the **native** compiler stacks for PyTorch and JAX. PyTorch compiles via TorchInductor, while JAX compiles via OpenXLA.

PyTorch (via torch_xla) and JAX can now compile to StableHLO, enabling the controlled comparison we perform in this work (see Figure 1). By using StableHLO as a common compilation target, we can isolate the impact of frontend design decisions from backend optimization strategies.

D. Framework Comparisons

Prior work has compared deep learning frameworks along various dimensions such as ease of use and performance. Most comparisons treat frameworks as black boxes, measuring end-to-end performance without examining the intermediate representations. Our work differs by explicitly analyzing the StableHLO IR emitted by each framework, allowing us to connect observable performance differences to structural properties of the compiler input.

III. Approach and Methodology

As mentioned previously, our methodology is designed to isolate the effects of frontend compiler design while holding backend compilation and hardware constant. We focus on JAX and PyTorch and compile both frameworks to StableHLO using the OpenXLA toolchain. We evaluate two categories of workloads designed to isolate fundamental compiler behaviors while also replicating realistic end-to-end execution.

The first consists of microbenchmarks that represent the dominant computations in modern neural networks. We evaluate general matrix multiplication (GEMM) as the core linear algebra primitive underlying fully connected layers and attention projections, using matrix sizes representative of common hidden dimensions in modern models. We include a transformer feed-forward network (FFN) block that is of the form linear layer to GELU activation to a final linear layer, as this is another common building block of many models. Finally, we evaluate multi-head attention (MHA), as this is a common, but computationally more irregular dataflow pattern, involving QKV projections, tensor reshaping and transposes, batched matrix multiplications, softmax, and output projection.

While microbenchmarks isolate individual computational patterns, they do not fully capture the interactions between layers, memory layout, and compiler optimizations that arise in production workloads. This motivates our second category of benchmarks, which is vision and language models. For vision, we include ResNet-50, a CNN with residual connections, and ViT-B/16, a Vision Transformer that uses transformer-style attention and repeated blocks. For language, we analyze BERT-Base, a 12-layer bidirectional transformer, and GPT-2 Small, a 12-layer autoregressive transformer with causal attention masking. To study scalability and memory behavior, we also include the larger GPT-Neo 1.3B and LLaMa-3.1 8B language models.

For each workload, we measure first-run latency, which includes tracing and compilation, as well as steady-state latency and throughput. In addition, we extract the StableHLO graphs prior to backend optimization and analyze their structure to determine how each framework encodes the same computation and whether these differences explain observed performance gaps.

IV. Implementation

All experiments were conducted on identical hardware to ensure fairness. We used a v6e-1 TPU with 32 GB of high-bandwidth memory, providing a controlled environment for evaluating both latency performance and memory behavior. We also used the Python implementations of JAX and PyTorch, along with OpenXLA as the compiler backend.

JAX programs were compiled using jax.jit, which traces the entire function execution to produce a static representation before lowering to StableHLO. This process captures the

full computation graph, including control flow and shape information, prior to execution.

For PyTorch, we used the torch_xla library to perform compilation via XLA instead of PyTorch's default backend. This ensured that both frameworks were lowered to StableHLO and optimized by the same compiler.

Finally, StableHLO graphs were captured and inspected to analyze structural differences in how each framework represents computation.

V. Evaluation

Our evaluation metrics are shown in Tables 1, 2, and 3. The next three sections contain a detailed analysis of these results.

A. Overall Latency

Our evaluation reveals consistent performance differences between JAX and PyTorch across both microbenchmarks and end-to-end models. For simple operations such as general matrix multiplication, PyTorch often exhibits lower initial compilation overhead which results in faster first-run latency. However, as computation complexity increases (moving from GEMMs to FFNs or MHAs), JAX consistently achieves lower steady-state latency. These trends are summarized quantitatively in Table 1, which reports latency measurements for the evaluated basic operations.

This can be immediately seen when comparing the two frameworks on feed-forward networks and multi-head attention. JAX reaches optimal performance after a single execution, while PyTorch typically requires two runs to stabilize. This behavior is explained in detail in the next section. Once steady state is reached, JAX often outperforms PyTorch by a significant margin. The steady-state advantages observed for feed-forward networks and multi-head attention are reflected in the measured latencies reported in Table 1.

TABLE I: Latency for Basic Operations (ms)

Operation	Framework	First Run	Second Run	Steady State
GEMM	JAX	41.243	0.451	0.192
	PyTorch	0.198	0.086	0.055
FFN	JAX	1081.837	0.474	0.225
	PyTorch	256.368	2.729	0.347
MHA	JAX	1118.373	0.619	0.363
	PyTorch	987.797	15.758	0.946

TABLE II: End-to-End Model Latency (ms)

Model	Framework	First Run	Second Run	Steady State
ResNet-50	JAX	7328.458	1.442	0.931
	PyTorch	7659.820	7271.575	10.562
BERT-Base	JAX	9103.082	1.351	0.913
	PyTorch	4098.641	3774.402	21.723
GPT-2 Small	JAX	2128.444	1.061	0.708
	PyTorch	3545.754	3256.955	27.896

B. Initial Compile Time

A consistent pattern across both microbenchmarks and end-to-end models is that PyTorch typically requires two executions to reach steady-state performance, where as JAX stabilizes after a single run. On the first execution, PyTorch incurs additional overhead beyond compilation, even when routing through OpenXLA. This overhead arises from PyTorch's dynamic execution model, in which portions of the computation graph and optimizer state are initialized lazily during the first iteration. As a result, the graph observed by the compiler during the first run does not fully reflect the steady-state computation.

During the second execution, PyTorch reuses cached structures and avoids repeated initialization, allowing the compiler to operate on a more stable graph. Only after this second run does PyTorch consistently achieve its best steady-state performance. This two-run stabilization behavior is visible in the latency measurements across both microbenchmarks and end-to-end workloads in Table 1 and Table 2.

In contrast, JAX's compilation model enforces a static, fully specified computation graph prior to execution. When a function is wrapped with jax.jit, the entire computation, including parameter shapes, control flow, and optimizer logic, is traced ahead of time and compiled as a single unit. Consequently, JAX reaches steady state immediately after the first execution, as there is no additional graph refinement or state materialization required at runtime.

While JAX generally exhibits faster convergence to steady state, we observe an exception in the case of BERT-Base, where JAX incurs a longer initial compilation time compared to the other models.

We attribute this behavior to the structure of BERT-Base, which introduces more complex control flow and shape interactions than the other evaluated models. In particular, BERT's use of layer normalization, residual connections, and attention masking leads to a larger and more complex traced graph. Because JAX traces and compiles the entire computation upfront, this added structural complexity directly increases compilation cost. This increased first-run latency for BERT-Base under JAX is shown in Table 2, which reports end-to-end model latencies across frameworks.

Importantly, this effect is limited to the first execution. Once compiled, JAX's steady-state performance on BERT-Base remains superior to PyTorch's, reinforcing the distinction between one-time compilation overhead and long-term execution efficiency. This observation highlights a trade-off inherent in JAX's design. Aggressive ahead-of-time tracing, as done in JAX, can increase upfront cost for complex models, but enables stronger optimization and stability during steady-state execution.

C. Memory Comparison

In addition to latency differences, we observe meaningful qualitative differences in memory behavior between JAX and PyTorch when executing large models under identical hardware constraints. While we do not report precise peak memory

TABLE III: Out-of-Memory Behavior on Large Models

Model	Framework	First Run	Second Run	Steady State
ViT-B/16	JAX	36453.13	2.313	1.836
	PyTorch	OOM	OOM	OOM
GPT-Neo 1.3B	JAX	8011.992	4.596	4.883
	PyTorch	OOM	OOM	OOM
LLaMA-3.1 8B	JAX	OOM	OOM	OOM
	PyTorch	OOM	OOM	OOM

usage values, out-of-memory (OOM) behavior provides a coarse, but informative signal of how effectively each framework manages memory during compilation and execution (see Table 3 for these results).

On a v6e-1 TPU with 32 GB of high-bandwidth memory (HBM), JAX was able to successfully execute ViT-B/16 and GPT-Neo 1.3B, whereas PyTorch encountered OOM failures for both models under the same conditions. For LLaMA-3.1 8B, neither framework was able to complete execution, indicating that this model exceeds the memory capacity of the device regardless of frontend choice. The differing outcomes for ViT-B/16 and GPT-Neo 1.3B suggest that JAX makes more effective use of available memory.

We attribute this difference primarily to how each framework exposes memory allocation opportunities to the compiler. JAX's ahead-of-time tracing produces a single, static computation graph for the entire execution step, enabling XLA to perform global memory optimization and aggressively reuse buffers across operations. Because the graph is fully known prior to execution, temporary tensors can often be overlapped or eliminated through fusion, reducing peak memory usage.

In contrast, PyTorch's dynamic execution model and lazy graph construction result in a sequence of smaller graphs that are compiled and executed incrementally. This fragmentation limits the compiler's ability to reason globally about memory lifetimes, increasing the chance of transient memory allocation spikes that lead to OOM failures (even when the total model size might otherwise fit within memory). Additionally, PyTorch's reliance on higher-level primitives (as we discuss in Section V.D) can mask opportunities for buffer reuse that are visible in JAX's more explicit IR.

While OOM behavior is an imprecise metric, these results show that frontend design choices significantly affect memory efficiency, even when both frameworks target the same backend compiler.

D. StableHLO Analysis

Analyzing the StableHLO emitted by both frameworks gives us some insight into the reasons behind these performance differences. At a high level, we find that JAX-generated graphs tend to preserve user-defined batch and sequence dimensions and enumerate mathematical operations for complex operations like LayerNorm. Because JAX exposes more fine-grained details to the compiler, the compiler has more freedom to perform complex optimizations/fusions. In contrast, PyTorch-generated graphs frequently flatten dimensions and rely on higher-level primitives, which can obscure program structure and limit the compiler's ability to optimize across operations.

In the next few paragraphs, we provide a detailed analysis of the differences in the StableHLO between JAX and PyTorch for two building blocks of transformer-based models: Multi-Head Attention (MHA) and LayerNorm.

MHA: For MHA, JAX's StableHLO (see Figure 3a in the appendix) preserves the high-dimensional structure of the computation throughout the forward pass. Specifically, JAX performs a single dot_general operation against a combined projection matrix that produces the query, key, and value tensors in one operation, followed by slicing to separate the three outputs. This design allows the compiler to load the projection weights in a single sequential sweep from memory, which avoids redundant kernel launches that would otherwise take up more resources. JAX applies the subsequent attention computations directly on four-dimensional tensors using dot_general with explicit batching and contracting dimensions. The contracting dimensions are those along which the dot product is applied. By expressing high-dimensional contractions directly in the IR, JAX enables XLA to generate custom kernel loop nests that do not require explicit reshaping or data movement. In contrast, in PyTorch (see Figure 3b in the appendix), the generated IR graph flattens the batch and sequence dimensions into a single dimension before the matrix multiplication, converting a tensor of shape $(Batch \times SequenceLength \times EmbeddingDimension) = 2 \times 64 \times 768$ into a 128×768 matrix. After the projection and bias addition, the tensor is reshaped back into its original dimensionality Although this approach performs well on optimized 2D matrix multiplication kernels, it has additional reshaping overhead and complicates the IR.

LayerNorm: We observe a similar pattern in the LayerNorm layers. JAX (see Figure 4a in the appendix) lowers layer normalization by explicitly enumerating the underlying arithmetic operations - it computes the mean and the variance, subtracts the mean, divides by the variance, and so on. This results in a longer but more transparent StableHLO graph. In addition, this exposes fine-grained computational structure to the compiler, allowing XLA to fuse operations and optimize memory reuse across the entire process. On the other hand, PyTorch (see Figure 4b in the appendix) maps layer normalization to a stablehlo.batch_norm_training primitive, which abstracts the operation as a single fused kernel. This trade-off suggests a broader distinction between the two stacks: PyTorch often favors higher-level fused primitives that align with existing kernel libraries, whereas JAX favors explicit mathematical structure that maximizes compiler visibility.

Takeaways: These differences in the emitted StableHLO align with the steady-state performance trends we discussed earlier. JAX usually achieves lower steady-state latency because its StableHLO exposes fine-grained details to the XLA compiler, allowing it to perform aggressive fusions and complex optimizations. In contrast, PyTorch's higher steady-latency can be explained by (1) its reliance on high-level primitives, which limit optimization opportunities across operations

and (2) its tendency to reshape the input tensors, which adds additional overhead.

VI. Discussion

In this section, we will discuss the limitations of our research and opportunities for future work.

A. Limitations

Our first limitation has to do with the fact that we route PyTorch through OpenXLA using torch_xla. We did this to ensure a fair comparison between the compiler frontends, but we realized this configuration does not represent PyTorch's native execution path using TorchInductor. Consequently, our results reflect how PyTorch behaves when forced into an XLA-centric workflow rather than its native ecosystem. This gives an unfair advantage to JAX in our evaluations.

Another limitation is that our memory analysis is qualitative rather than quantitative. Although out-of-memory failures provide a strong signal of relative memory efficiency, we do not report precise peak memory usage or allocation timelines. This limits our ability to attribute OOM behavior to specific tensors, buffer lifetimes, or fusion decisions. More fine-grained memory monitoring would allow us to make stronger conclusions.

A third limitation is that our StableHLO analysis focuses on qualitative structural differences, rather than systematic IR metrics. While manual inspection reveals clear patterns, such as dimension flattening versus preservation, and fine-grained math versus high-level primitives, we do not yet quantify graph size, fusion count, etc. A more systematic IR-level comparison could build on the StableHLO examples shown in Figures 2–4 of the appendix by extracting these types of metrics.

Finally, our benchmark set, while representative, is not exhaustive. We focus on widely used vision and transformer models, but do not evaluate emerging architectures such as mixture-of-experts models, sparse attention mechanisms, or models with highly irregular control flow. These workloads may stress the compiler in different ways.

B. Future Work and Next Steps

One first next step is to perform more comprehensive profiling, particularly around memory allocation. Analyzing peak memory usage and buffer allocations would allow us to directly correlate StableHLO structure with memory efficiency and get a more concrete understanding of why PyTorch runs OOM for certain models (see Table 3), while JAX does not.

Another area for future work is to analyze PyTorch in its native compilation stack. Comparing PyTorch compiled via TorchInductor against JAX compiled via XLA would provide insight into how much of the observed performance gap is attributable to frontend versus backend design and whether one stack is explicitly better than they other. Finally, given that it would also be interesting to explore if it is possible to create a unified/hybrid stack that combines the best properties of JAX, PyTorch, and other frameworks.

VII. Conclusion

In this paper, we presented a comprehensive comparison of JAX and PyTorch using the OpenXLA compiler stack. By compiling both frameworks to the same StableHLO intermediate representation and executing them on identical hardware, we isolated the impact of frontend framework design on performance and memory efficiency. Our results demonstrate that JAX consistently achieves superior steady-state performance and memory efficiency for complex workloads, largely due to how it exposes computation structure to the compiler. These findings emphasize the importance of frontend compiler design in modern ML systems and also reveal how different the JAX and PyTorch stacks are internally, despite providing the same functionality.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[2] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[4] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne *et al.*, "Jax: composable transformations of python+ numpy programs," 2018.

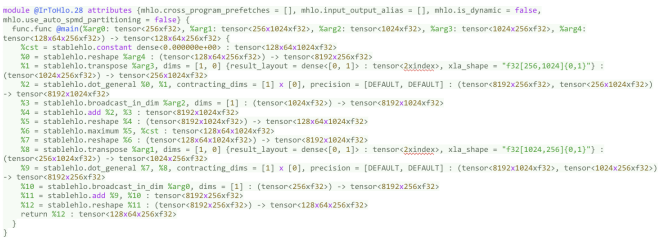
[5] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 929–947.

[6] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," in *SysML conference 2018*, 2019.

Appendix

The appendix, which contains the StableHLO representations referenced in Section V.D, is on the next page.

(a) JAX Feed-Forward Network StableHLO



(b) PyTorch Feed-Forward Network StableHLO

Fig. 2: Comparison of Feed-Forward Network StableHLO between JAX and PyTorch.

```
%40 = stablehlo.transpose %arg1, dims = [1, 0] : (tensor<2304x768xf32>) ->
tensor<768x2304xf32>
%41 = stablehlo.dot_general %31, %40, contracting_dims = [2] x [0],
precision = [DEFAULT, DEFAULT] : (tensor<2x64x768xf32>,
tensor<768x2304xf32>) -> tensor<2x64x2304xf32>
%42 = stablehlo.broadcast_in_dim %arg0, dims = [2] : (tensor<2304xf32>) ->
tensor<1x1x2304xf32>
%43 = stablehlo.broadcast_in_dim %42, dims = [0, 1, 2] :
(tensor<1x1x2304xf32>) -> tensor<2x64x2304xf32>
%44 = stablehlo.add %41, %43 : tensor<2x64x2304xf32>
%45 = stablehlo.slice %44 [0:2, 0:64, 0:768] : (tensor<2x64x2304xf32>) ->
tensor<2x64x768xf32>  # Query
%46 = stablehlo.slice %44 [0:2, 0:64, 768:1536] : (tensor<2x64x2304xf32>)
-> tensor<2x64x768xf32>  # Key
%47 = stablehlo.slice %44 [0:2, 0:64, 1536:2304] : (tensor<2x64x2304xf32>)
-> tensor<2x64x768xf32>  # Value
%48 = stablehlo.reshape %45 : (tensor<2x64x768xf32>) ->
tensor<2x64x12x64xf32>  # Q: [2, 64, 12, 64]
%49 = stablehlo.reshape %46 : (tensor<2x64x768xf32>) ->
tensor<2x64x12x64xf32>  # K: [2, 64, 12, 64]
%50 = stablehlo.reshape %47 : (tensor<2x64x768xf32>) ->
tensor<2x64x12x64xf32>  # V: [2, 64, 12, 64]
```

(a) JAX Multi-Head Attention Layer StableHLO

```
%arg70: tensor<768x2304xf32>  // Weight: 768 -> 2304 (768*3 for
Q, K, V)
%arg69: tensor<2304xf32>      // Bias
%13 = stablehlo.reshape %12 : (tensor<2x64x768xf32>) ->
tensor<128x768xf32>
%14 = stablehlo.dot_general %13, %arg70,
      contracting_dims = [1] x [0],  // Contract over hidden dim
      precision = [DEFAULT, DEFAULT] :
      (tensor<128x768xf32>, tensor<768x2304xf32>) ->
tensor<128x2304xf32>
%15 = stablehlo.broadcast_in_dim %arg69, dims = [1] :
      (tensor<2304xf32>) -> tensor<128x2304xf32>
%16 = stablehlo.add %14, %15 : tensor<128x2304xf32>
%17 = stablehlo.reshape %16 : (tensor<128x2304xf32>) ->
tensor<2x64x2304xf32>
```

(b) PyTorch Multi-Head Attention Layer StableHLO

Fig. 3: Comparison of Multi-Head Attention Layer StableHLO between JAX and PyTorch.

```
%6 = stablehlo.multiply %5, %5 : tensor<2x64x768xf32>  # x²
%cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
%7 = stablehlo.reduce(%5 init: %cst) applies stablehlo.add across dimensions
= [2] : (tensor<2x64x768xf32>, tensor<f32>) -> tensor<2x64xf32>  # sum(x)
%cst_0 = stablehlo.constant dense<7.680000e+02> : tensor<f32>  # 768.0
(hidden dim)
%8 = stablehlo.broadcast_in_dim %cst_0, dims = [] : (tensor<f32>) ->
tensor<2x64xf32>
%9 = stablehlo.divide %7, %8 : tensor<2x64xf32>  # mean = sum(x) / 768
%cst_1 = stablehlo.constant dense<0.000000e+00> : tensor<f32>
%10 = stablehlo.reduce(%6 init: %cst_1) applies stablehlo.add across
dimensions = [2] : (tensor<2x64x768xf32>, tensor<f32>) -> tensor<2x64xf32>  #
sum(x²)
%cst_2 = stablehlo.constant dense<7.680000e+02> : tensor<f32>
...
%27 = stablehlo.multiply %25, %26 : tensor<2x64x768xf32>  # (1/sqrt(var)) *
gain
%28 = stablehlo.multiply %20, %27 : tensor<2x64x768xf32>  # (x - mean) * gain
/ sqrt(var)
%29 = stablehlo.reshape %arg4 : (tensor<768xf32>) -> tensor<1x1x768xf32>
%30 = stablehlo.broadcast_in_dim %29, dims = [0, 1, 2] :
(tensor<1x1x768xf32>) -> tensor<2x64x768xf32>
%31 = stablehlo.add %28, %30 : tensor<2x64x768xf32>  # LayerNorm output
```

(a) JAX LayerNorm StableHLO

```
%7 = stablehlo.reshape %6 : (tensor<2x64x768xf32>) -> tensor<1x128x768xf32>
%cst_5 = stablehlo.constant dense<1.000000e+00> : tensor<128xf32>  // scale
%cst_4 = stablehlo.constant dense<0.000000e+00> : tensor<128xf32>  // offset
%output, %batch_mean, %batch_var = "stablehlo.batch_norm_training"(%7,
%cst_5, %cst_4) <{
  epsilon = 9.99999974E-6 : f32,
  feature_index = 1 : i64  // Normalize over dimension 1 (the 128 dimension)
}> : (tensor<1x128x768xf32>, tensor<128xf32>, tensor<128xf32>)
  -> (tensor<1x128x768xf32>, tensor<128xf32>, tensor<128xf32>)
%9 = stablehlo.reshape %output : (tensor<1x128x768xf32>) ->
tensor<2x64x768xf32>
%arg71: tensor<768xf32>  // gamma (scale)
%arg72: tensor<768xf32>  // beta (bias/offset)
%10 = stablehlo.broadcast_in_dim %arg71, dims = [2] :
      (tensor<768xf32>) -> tensor<2x64x768xf32>
%8 = stablehlo.broadcast_in_dim %arg72, dims = [2] :
     (tensor<768xf32>) -> tensor<2x64x768xf32>
%11 = stablehlo.multiply %9, %10 : tensor<2x64x768xf32>
%12 = stablehlo.add %8, %11 : tensor<2x64x768xf32>
```

(b) PyTorch LayerNorm StableHLO

Fig. 4: Comparison of LayerNorm StableHLO between JAX and PyTorch.