Stack Wars: JAX vs. PyTorch

Albert Cao, Eric Zhao, John Xie, Vishnu Kannan

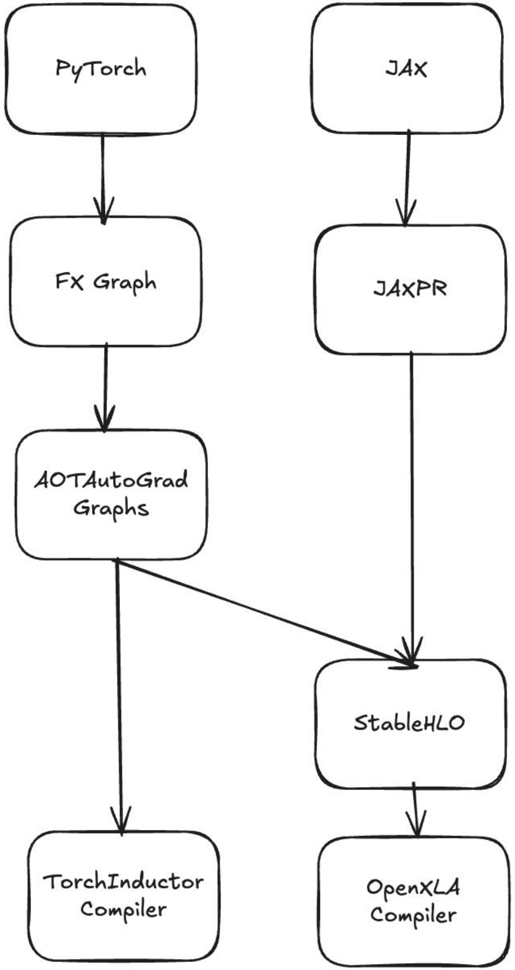
Research Problem/Motivation

- PyTorch and JAX have reached feature parity over the last few years
- However, despite this apparent feature parity, their real-world performance still varies widely across workloads, hardware, and compiler modes
- These frameworks differ in how they lower high-level Python code to IRs and how they interact with optimizing compilers such as OpenXLA
- Understanding why these differences exist, even with similar compiler backends, is both academically interesting and practically impactful

How and why does the performance of PyTorch and JAX differ on the same operations even when compiling to the same backend (Open XLA)?

System Diagram

- By default, JAX compiles directly to StableHLO, which is passed into the OpenXLA Compiler, while PyTorch is compiled using TorchInductor
- But PyTorch can be compiled to StableHLO using torch_xla
- To ensure a fair comparison, we chose to evaluate both libraries on the same IR/compiler (from StableHLO down)
    - But this gives an advantage to JAX, more on this later …



Experimental Setup

- Hardware: v6e-1 TPU
  - 32 GB HBM (TPU equivalent of VRAM)
- Software:
  - Language: Python
  - Libraries: PyTorch, JAX
  - Compilers: XLA
- Basic Operations:
  - GEMM, FFN, MHA
- End-To-End-Models:
  - ResNet-50, Bert-Base, GPT-2 Small
  - ViT-B/16, GPT-Neo 1.3B, LLaMA-3.1 8B

Evaluations - Latency for Basic Operations

- In general, the first run is significantly slower than the steady state
  - Makes sense since the compilation happens on the first run
- JAX is slower than PyTorch for GEMM, but (increasingly) faster for FFN/MHA
  - As the complexity of the operation increases, JAX seems to shine
- JAX reaches steady state performance after the the first run, while PyTorch needs 2 runs to achieve steady state
  - Because JAX is functional with fixed shapes before compilation, XLA can optimize the entire training step, whereas PyTorch creates optimizer state lazily during the first step, causing the computation graph to change.

GEMM	First Run	Second Run	Avg Steady St.
JAX	41.243 ms	0.451 ms	0.192 ms
PyTorch	0.198 ms	0.086 ms	0.055 ms

FFN	First Run	Second Run	Avg Steady St.
JAX	1081.837 ms	0.474 ms	0.225 ms
PyTorch	256.368 ms	2.729 ms	0.347 ms

MHA	First Run	Second Run	Avg Steady St.
JAX	1118.373 ms	0.619 ms	0.363 ms
PyTorch	987.797 ms	15.758 ms	0.946 ms

Evaluations - Latency for End-To-End Models

- Again, it's much clearer here that PyTorch needs two runs to reach steady state while Jax only needs one run
- PyTorch's steady state is much slower than Jax's steady state overall
  - This follows the trend that when the complexity increases, JAX seems to shine
- ResNet-50 and GPT-2 Small have simple, regular computation graphs that JAX can trace and compile efficiently, so JAX's first-run latency is lower, but BERT-Base introduces more complex patterns, which take longer for JAX's tracing and XLA compilation.

ResNet-50	First Run	Second Run	Avg Steady State
JAX	7328.458 ms	1.442 ms	0.931 ms
PyTorch	7659.820 ms	7271.575 ms	10.562 ms

Bert-Base	First Run	Second Run	Avg Steady State
JAX	9103.082 ms	1.351 ms	0.913 ms
PyTorch	4098.641 ms	3774.402 ms	21.723 ms

GPT-2 Small	First Run	Second Run	Avg Steady State
JAX	2128.444 ms	1.061 ms	0.708 ms
PyTorch	3545.754 ms	3256.955 ms	27.896 ms

Evaluations - Latency for End-To-End Models (OOM)

- There were many models where JAX was able to handle the memory load but PyTorch was not.
- Seems to imply that JAX is more memory efficient than PyTorch at least when both are compiled using OpenXLA.
- JAX uses a static, side effect free computation model that lets XLA reuse memory globally.
- JAX compiles each full training step into one fused XLA graph, while PyTorch ends up creating multiple smaller graphs.

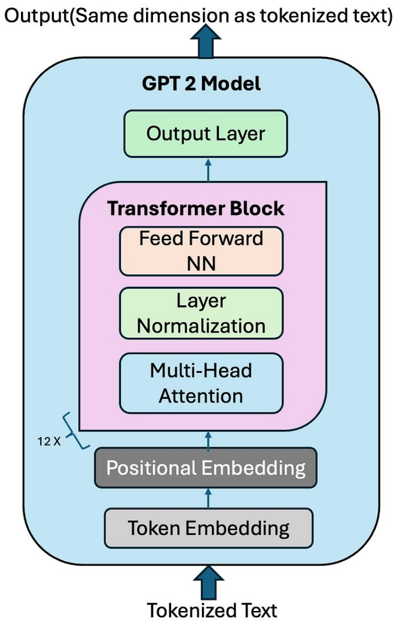
ViT-B/16	First Run	Second Run	Avg Steady State
JAX	36453.13 ms	2.313 ms	1.836 ms
PyTorch	OOM	OOM	OOM

GPT-Neo 1.3B	First Run	Second Run	Avg Steady State
JAX	8011.992 ms	4.596 ms	4.883 ms
PyTorch	OOM	OOM	OOM

LLaMA-3.1 8B	First Run	Second Run	Avg Steady State
JAX	OOM	OOM	OOM
PyTorch	OOM	OOM	OOM

Evaluations - StableHLO Analysis GPT-2 Small

- Captured StableHLO of GPT-2 Small forward pass for JAX and PyTorch
- AI assisted analysis
  - Both StableHLO representations are ~1500-2000 lines
  - Embedding lookup → 12 repeated transformer blocks
  - Each block consists of:
    - Layer Normalization → Multi-Head Attention → MLP/FFN



Evaluations - StableHLO Analysis (Initial Embeddings)

- First step: adding the token embeddings and positional embeddings for each token in the batch

JAX

```
# Create position indices [0, 1, 2, ..., 63] and broadcast to batch shape
%0 = stablehlo.iota dim = 0 : tensor<64xi32>
%1 = stablehlo.broadcast_in_dim %0, dims = [1] : (tensor<64xi32>) -> tensor<1x64xi32>
%2 = stablehlo.broadcast_in_dim %1, dims = [0, 1] : (tensor<1x64xi32>) ->
tensor<2x64xi32>

# Token embeddings: lookup from vocab (arg147 = wte weights)
%3 = call @_take(%arg147, %arg148) : (tensor<50257x768xf32>, tensor<2x64xi32>) ->
tensor<2x64x768xf32>

# Position embeddings: lookup from position table (arg146 = wpe weights)
%4 = call @_take_7(%arg146, %2) : (tensor<1024x768xf32>, tensor<2x64xi32>) ->
tensor<2x64x768xf32>

# Combined embeddings: token + position
%5 = stablehlo.add %3, %4 : tensor<2x64x768xf32>
```

JAX and PyTorch differ in how they:

- Generate the position indices [0…63]
- Do the position embeddings lookup

PyTorch

```
%0 = stablehlo.reshape %arg65 : (tensor<2x64xi64>) -> tensor<128xi64>
%1 = stablehlo.convert %0 : (tensor<128xi64>) -> tensor<128xui32>
%2 = "stablehlo.gather"(%arg66, %1) <{
  dimension_numbers = #stablehlo.gather
    ...
  >,
  indices_are_sorted = false,
  slice_sizes = array<i64: 1, 768>  // Take one vocab entry, all 768 dims
}> : (tensor<50257x768xf32>, tensor<128xui32>) -> tensor<128x768xf32>
%3 = stablehlo.reshape %2 : (tensor<128x768xf32>) -> tensor<2x64x768xf32>
%c_6 = stablehlo.constant dense<[0, 1, 2, ..., 63]> : tensor<64xui32>
%4 = "stablehlo.gather"(%arg64, %c_6) <{
  dimension_numbers = #stablehlo.gather
    ...
  >,
  indices_are_sorted = false,
  slice_sizes = array<i64: 1, 768>
}> : (tensor<1024x768xf32>, tensor<64xui32>) -> tensor<64x768xf32>
%5 = stablehlo.broadcast_in_dim %4, dims = [1, 2] :
     (tensor<64x768xf32>) -> tensor<2x64x768xf32>
%6 = stablehlo.add %3, %5 : tensor<2x64x768xf32>
```

Evaluations - StableHLO Analysis (1st Layer Normalization)

- Layer Norm: stabilization technique performed across embedding dimension to make training faster

JAX

```
%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
```

PyTorch

```
%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>
```

Differences

- PyTorch flattens batch and sequence dimensions
- JAX enumerates mathematical steps, while PyTorch uses high-level b**atch_norm_training** primitive

Evaluations - StableHLO Analysis (Multi-Head Attention)

- Multi-Head Attention: mechanism by which tokens in a sequence “communicate” with each other

JAX

```
%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]
```

PyTorch

```
%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>
```

Differences

- PyTorch flattens batch and sequence dimension again, while JAX uses the **dot_general** primitive

Comparison Takeaways

- JAX enumerates the mathematical steps for various components (like Layer Norm), while PyTorch relies on high level functions/primitives
  - JAX exposes more fine-grained details to the XLA compiler, which allows to perform more complex, custom fusions
- JAX maintains the sequence and batch dimension, while PyTorch flattens them
- JAX was designed specifically for OpenXLA, so it is somewhat unsurprising that its performance is better

Next Steps

- Compare native PyTorch and JAX stacks (for a more fair comparison)
  - Compare benchmarks of PyTorch compiled on Torch Inductor with current benchmarks
- Benchmark TensorFlow with XLA and native stack
- Stretch goal: Create a unified stack to get the best of both worlds

Thank you!