

# Stack Wars: JAX vs. PyTorch

Albert Cao, Eric Zhao, John Xie, Vishnu Kannan

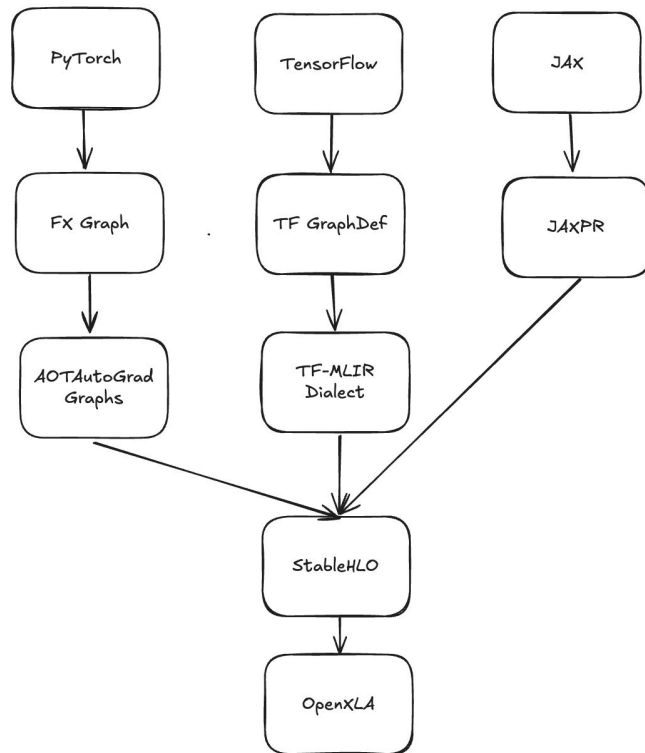
# Problems + Motivation

- The major deep learning frameworks (PyTorch, JAX, and TensorFlow) 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

# Context: Compilation Across Frameworks

- PyTorch, TensorFlow, and JAX functions can also be compiled for better performance
  - PyTorch -> `torch.compile()`
  - JAX -> `@jax.jit`
  - TensorFlow -> `@tf.function`
- Compilation transforms a function into a device-agnostic static computation graph, applies optimizations, and generates a device-specific executable
- In compilation mode, they all lower their high-level Python models into the same IR, StableHLO, before execution which enables:
  - Operation fusion and global optimizations for faster, more efficient execution.
  - Device-optimized code that maximizes GPU/TPU utilization and portability across hardware.

# Compilation Across Frameworks Diagram



# Progress

- For JAX and PyTorch:
  - Tested 4 common model components (GEMMs, FFNs, Multi-Head Attention, Convolutions)
  - Captured StableHLO intermediate representations and observed high-level differences
  - Implemented profiling functions for each framework that collect performance metrics (latency and memory usage) and capture timeline of operations on CPU and TPU/GPU

# MLIR Comparisons

## JAX: StableHLO for GEMM

```
module @jit_gemm attributes {mhlo.num_partitions = 1 : i32, mhlo.num_replicas = 1 : i32} {  
  func.func public @main(%arg0: tensor<128x256xf32>, %arg1: tensor<256x1024xf32>) ->  
  (tensor<128x1024xf32> {jax.result_info = "result"}) {  
    %0 = stablehlo.dot_general %arg0, %arg1, contracting_dims = [1] x [0], precision =  
    [DEFAULT, DEFAULT] : (tensor<128x256xf32>, tensor<256x1024xf32>) -> tensor<128x1024xf32>  
    return %0 : tensor<128x1024xf32>  
  }  
}
```

## PyTorch: StableHLO for GEMM

```
module @IrToHlo.5 attributes {mhlo.cross_program_prefetches = [], mhlo.input_output_alias = [],  
mhlo.is_dynamic = false, mhlo.use_auto_spmd_partitioning = false} {  
  func.func @main(%arg0: tensor<256x1024xf32>, %arg1: tensor<128x256xf32>) -> tensor<128x1024xf32> {  
    %0 = stablehlo.dot_general %arg1, %arg0, contracting_dims = [1] x [0], precision = [DEFAULT,  
    DEFAULT] : (tensor<128x256xf32>, tensor<256x1024xf32>) -> tensor<128x1024xf32>  
    return %0 : tensor<128x1024xf32>  
  }  
}
```

# MLIR Comparisons

## JAX: StableHLO for FFN - $(w_2 * \text{RELU}(w_1 * x + b_1) + b_2)$

```
module @jit_ffn attributes {mhlo.num_partitions = 1 : i32, mhlo.num_replicas = 1 : i32} {
  func.func public @main(%arg0: tensor<128x64x256xf32>, %arg1: tensor<256x1024xf32>, %arg2: tensor<1024xf32>, %arg3: tensor<1024x256xf32>,
    %arg4: tensor<256xf32>) -> (tensor<128x64x256xf32> {jax.result_info = "result"}) {
    %0 = stablehlo.dot_general %arg0, %arg1, contracting_dims = [2] x [0], precision = [DEFAULT, DEFAULT] : (tensor<128x64x256xf32>,
    tensor<256x1024xf32>) -> tensor<128x64x1024xf32>
    %1 = stablehlo.broadcast_in_dim %arg2, dims = [2] : (tensor<1024xf32>) -> tensor<1x1x1024xf32>
    %2 = stablehlo.broadcast_in_dim %1, dims = [0, 1, 2] : (tensor<1x1x1024xf32>) -> tensor<128x64x1024xf32>
    %3 = stablehlo.add %0, %2 : tensor<128x64x1024xf32>
    %4 = call @relu(%3) : (tensor<128x64x1024xf32>) -> tensor<128x64x1024xf32>
    %5 = stablehlo.dot_general %4, %arg3, contracting_dims = [2] x [0], precision = [DEFAULT, DEFAULT] : (tensor<128x64x1024xf32>,
    tensor<1024x256xf32>) -> tensor<128x64x256xf32>
    %6 = stablehlo.broadcast_in_dim %arg4, dims = [2] : (tensor<256xf32>) -> tensor<1x1x256xf32>
    %7 = stablehlo.broadcast_in_dim %6, dims = [0, 1, 2] : (tensor<1x1x256xf32>) -> tensor<128x64x256xf32>
    %8 = stablehlo.add %5, %7 : tensor<128x64x256xf32>
    return %8 : tensor<128x64x256xf32>
  }
  func.func private @relu(%arg0: tensor<128x64x1024xf32>) -> tensor<128x64x1024xf32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %0 = stablehlo.broadcast_in_dim %cst, dims = [] : (tensor<f32>) -> tensor<128x64x1024xf32>
    %1 = stablehlo.maximum %arg0, %0 : tensor<128x64x1024xf32>
    return %1 : tensor<128x64x1024xf32>
  }
}
```

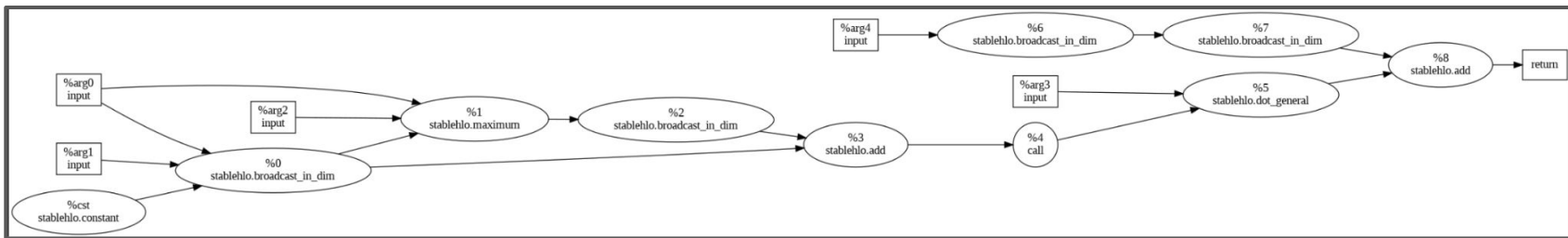
# MLIR Comparisons

## PyTorch: StableHLO for FFN

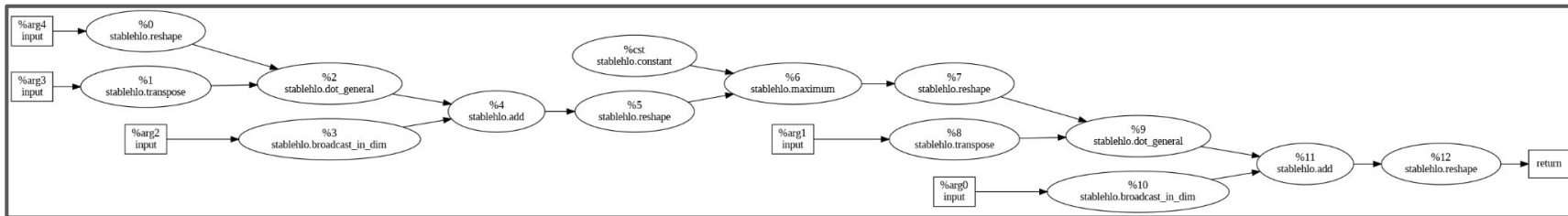
```
module @IrToHlo.28 attributes {mhlo.cross_program_prefetches = [], mhlo.input_output_alias = [], mhlo.is_dynamic = false,
mhlo.use_auto_spmv_partitioning = false} {
  func.func @main(%arg0: tensor<256xf32>, %arg1: tensor<256x1024xf32>, %arg2: tensor<1024xf32>, %arg3: tensor<1024x256xf32>, %arg4:
tensor<128x64x256xf32>) -> tensor<128x64x256xf32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<128x64x1024xf32>
    %0 = stablehlo.reshape %arg4 : (tensor<128x64x256xf32>) -> tensor<8192x256xf32>
    %1 = stablehlo.transpose %arg3, dims = [1, 0] {result_layout = dense<[0, 1]> : tensor<2xindex>, xla_shape = "f32[256,1024]{0,1}" } :
(tensor<1024x256xf32>) -> tensor<256x1024xf32>
    %2 = stablehlo.dot_general %0, %1, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] : (tensor<8192x256xf32>, tensor<256x1024xf32>)
-> tensor<8192x1024xf32>
    %3 = stablehlo.broadcast_in_dim %arg2, dims = [1] : (tensor<1024xf32>) -> tensor<8192x1024xf32>
    %4 = stablehlo.add %2, %3 : tensor<8192x1024xf32>
    %5 = stablehlo.reshape %4 : (tensor<8192x1024xf32>) -> tensor<128x64x1024xf32>
    %6 = stablehlo.maximum %5, %cst : tensor<128x64x1024xf32>
    %7 = stablehlo.reshape %6 : (tensor<128x64x1024xf32>) -> tensor<8192x1024xf32>
    %8 = stablehlo.transpose %arg1, dims = [1, 0] {result_layout = dense<[0, 1]> : tensor<2xindex>, xla_shape = "f32[1024,256]{0,1}" } :
(tensor<256x1024xf32>) -> tensor<1024x256xf32>
    %9 = stablehlo.dot_general %7, %8, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] : (tensor<8192x1024xf32>, tensor<1024x256xf32>)
-> tensor<8192x256xf32>
    %10 = stablehlo.broadcast_in_dim %arg0, dims = [1] : (tensor<256xf32>) -> tensor<8192x256xf32>
    %11 = stablehlo.add %9, %10 : tensor<8192x256xf32>
    %12 = stablehlo.reshape %11 : (tensor<8192x256xf32>) -> tensor<128x64x256xf32>
    return %12 : tensor<128x64x256xf32>
  }
}
```

# StableHLO Visualizations

## JAX: StableHLO Visualization for FNN



## PyTorch: StableHLO Visualization for FNN



# Next Steps

1. Complete profiling for MHA and CNNs
2. Collect additional metrics
3. Deeper analysis on StableHLO representations
4. Further analysis on more complex models