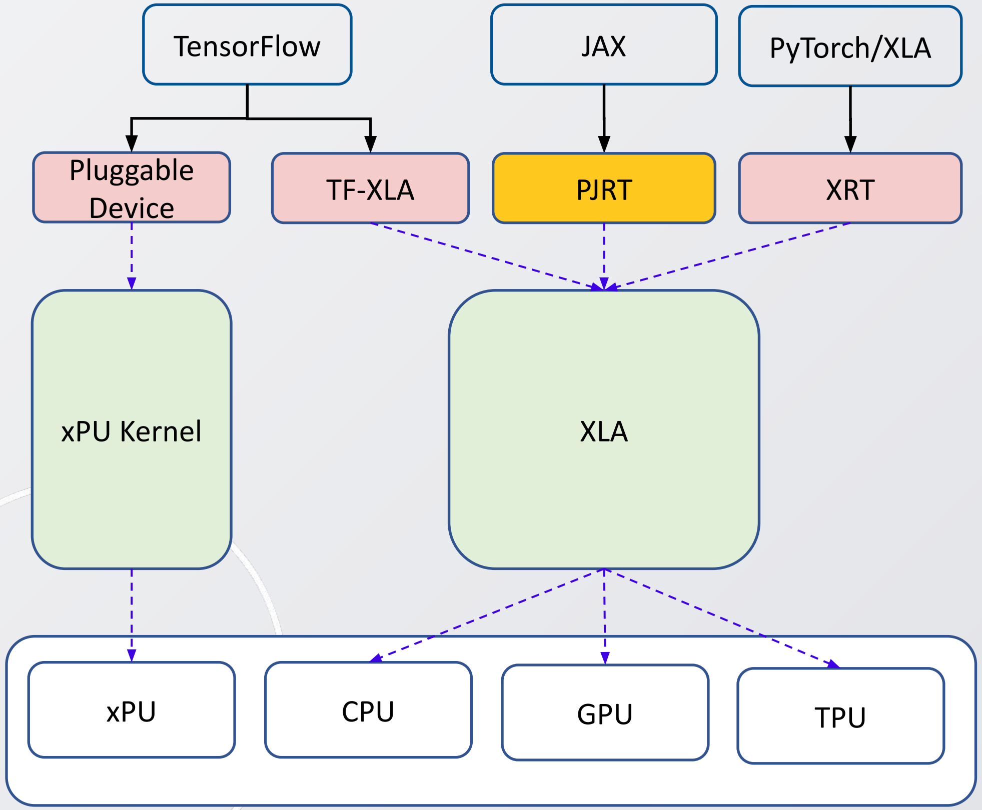
# Accelerate JAX on Intel GPUs via PJRT


Xiao Yu, Google

Yiqiang Li, Intel

# Problem

- Fragmented stack for different ML frameworks
- No way to add new hardware support in JAX
- TensorFlow has its own way for adding new hardware support
    - Cannot leverage compiler technologies

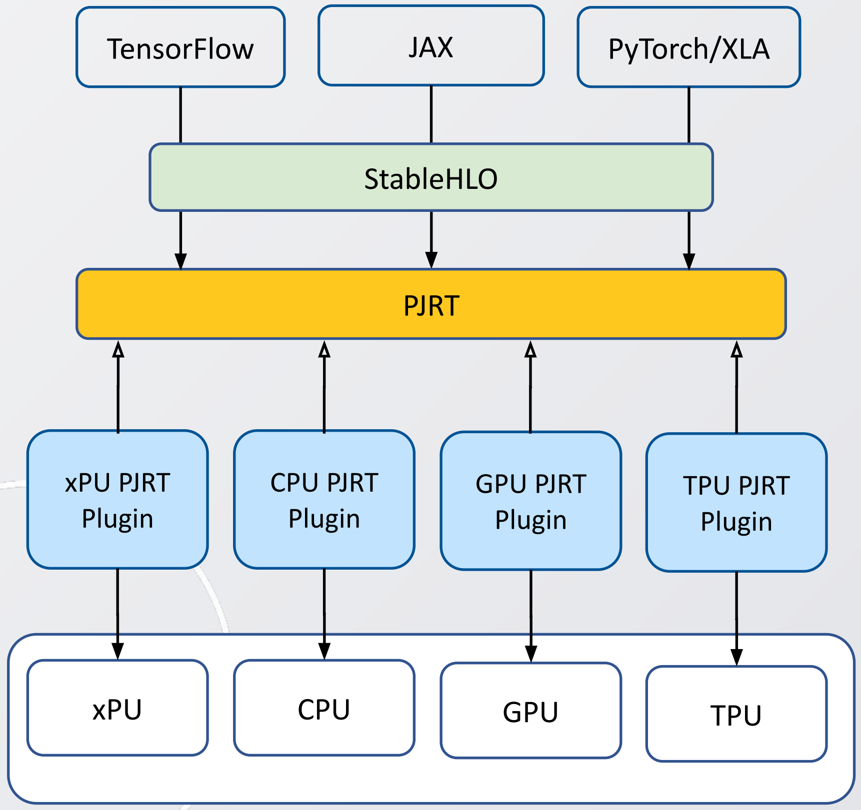


# PJRT

- Unified pluggable device API --- **PJRT**

- A PJRT Plugin contains:
  - Hardware-specific compiler, which takes StableHLO as standard input IR.
  - Hardware runtime

- Decouple ML Framework and device PJRT plugin
  - Plugin are discovered and loaded as dynamic library

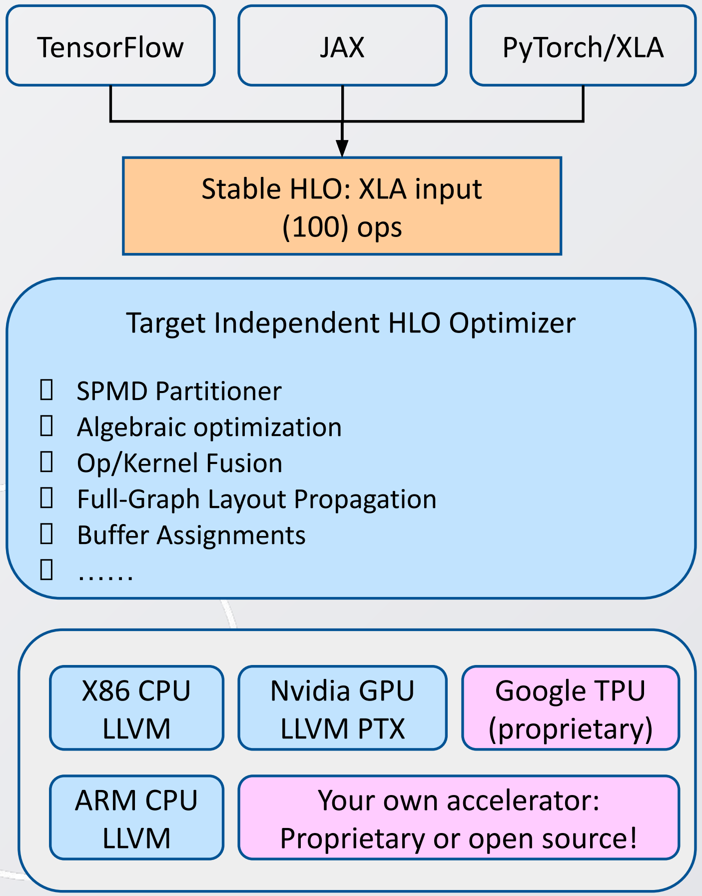


# PJRT: high level overview

- Scope: "**Compile and Execute a StableHLO program**"
- A PJRT Plugin will implement following APIs:
  - **Compile:** StableHLO -> PjrtLoadedExecutable
    - Trigger HW specific compiler to compile StableHLO into a HW executable
  - **H2D transfer:** host buffer, PjrtDevice -> PjrtBuffer
    - Prepare input by transferring data from host to HW
  - **Execute:** PjrtLoadedExecutable, PjrtDevice, PjrtBuffer -> PjrtBuffer
    - Execute the program on HW
  - **D2H transfer:** PjrtBuffer -> host buffer
    - Transfer output back to host

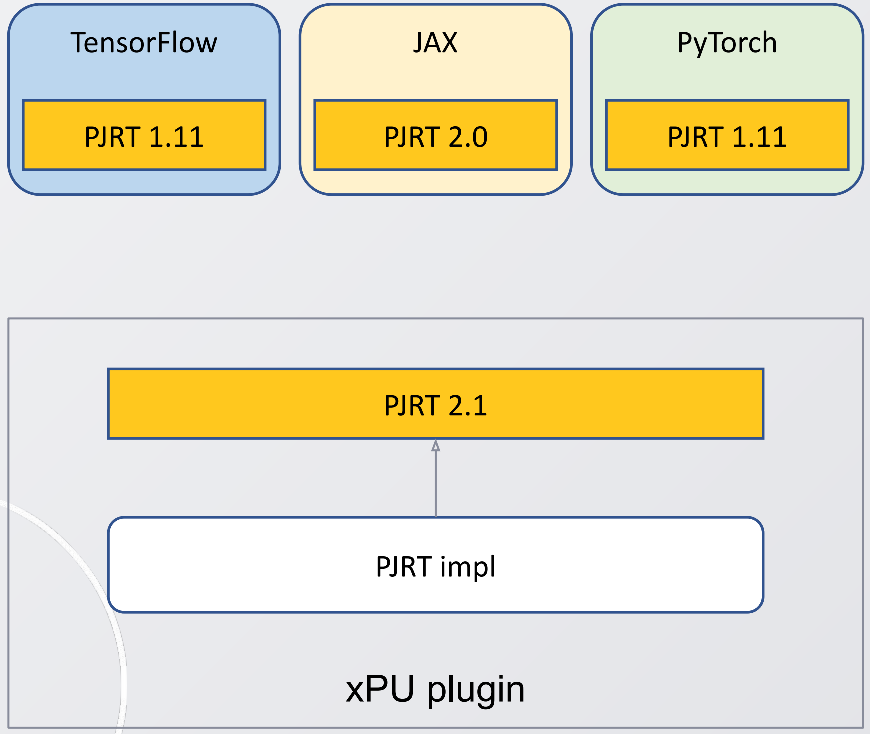
# OpenXLA

- It is challenging to build a new device compiler and runtime from scratch
- OpenXLA: Open, state-of-the-art ML compiler, using the best of XLA & MLIR
- Reusable target independent optimizer



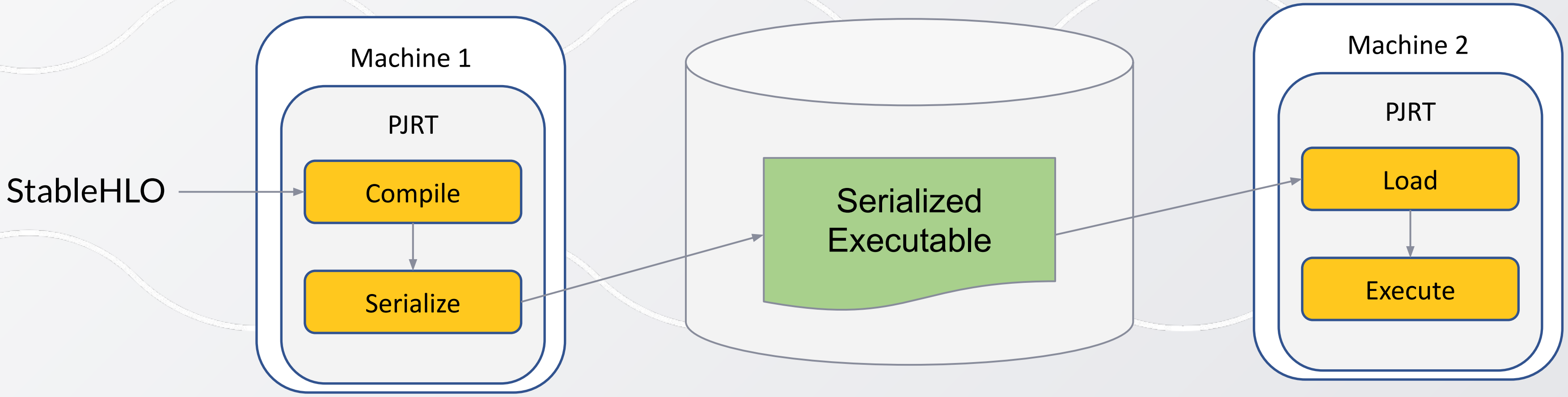
# Versioning

- ML Frameworks and PJRT Plugins can be released separately
    - ML Frameworks and PJRT Plugins may use different version of PJRT.
- Proposal:
    - The program works as long as the framework and the plugin have the same Major version.



# Ahead of Time Compilation

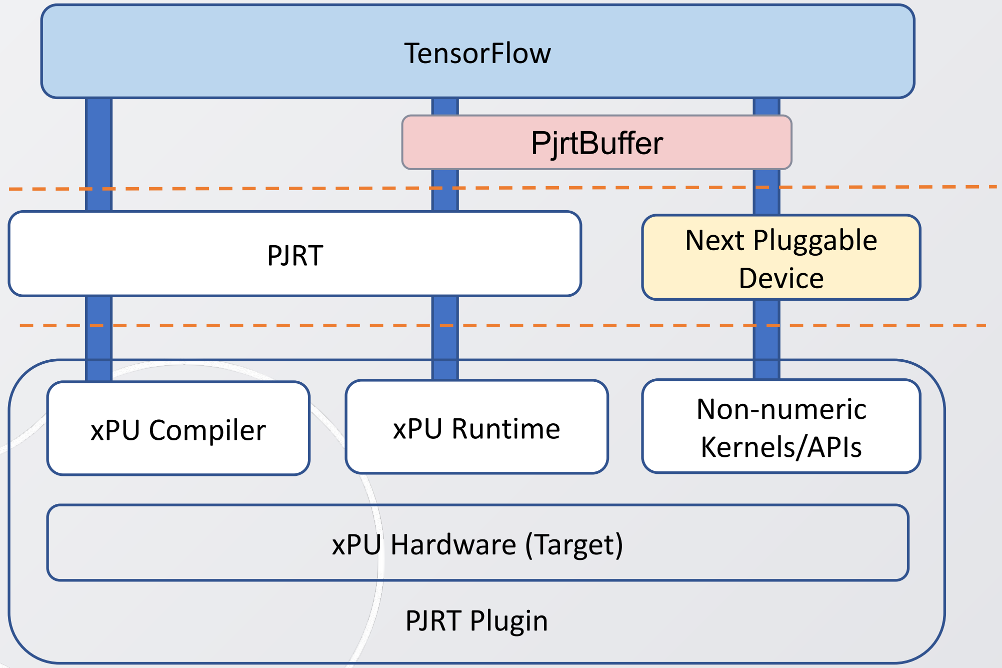
- Compile and execute StableHLO program on different machines
- Supported by Jax (done) and TensorFlow (WIP)
- PJRT Plugin is responsible for defining versioning and compatibility of serialized executable



# Custom Kernel support

- Custom Kernels are critical to support
  - Operation that cannot be represented in StableHLO.
  - Hand-written kernel

- Use PjrtBuffer as the concurrency to enable efficient (0 Copy) buffer sharing between PJRT and custom kernels
  - Implemented in TensorFlow as NextPluggableDevice API

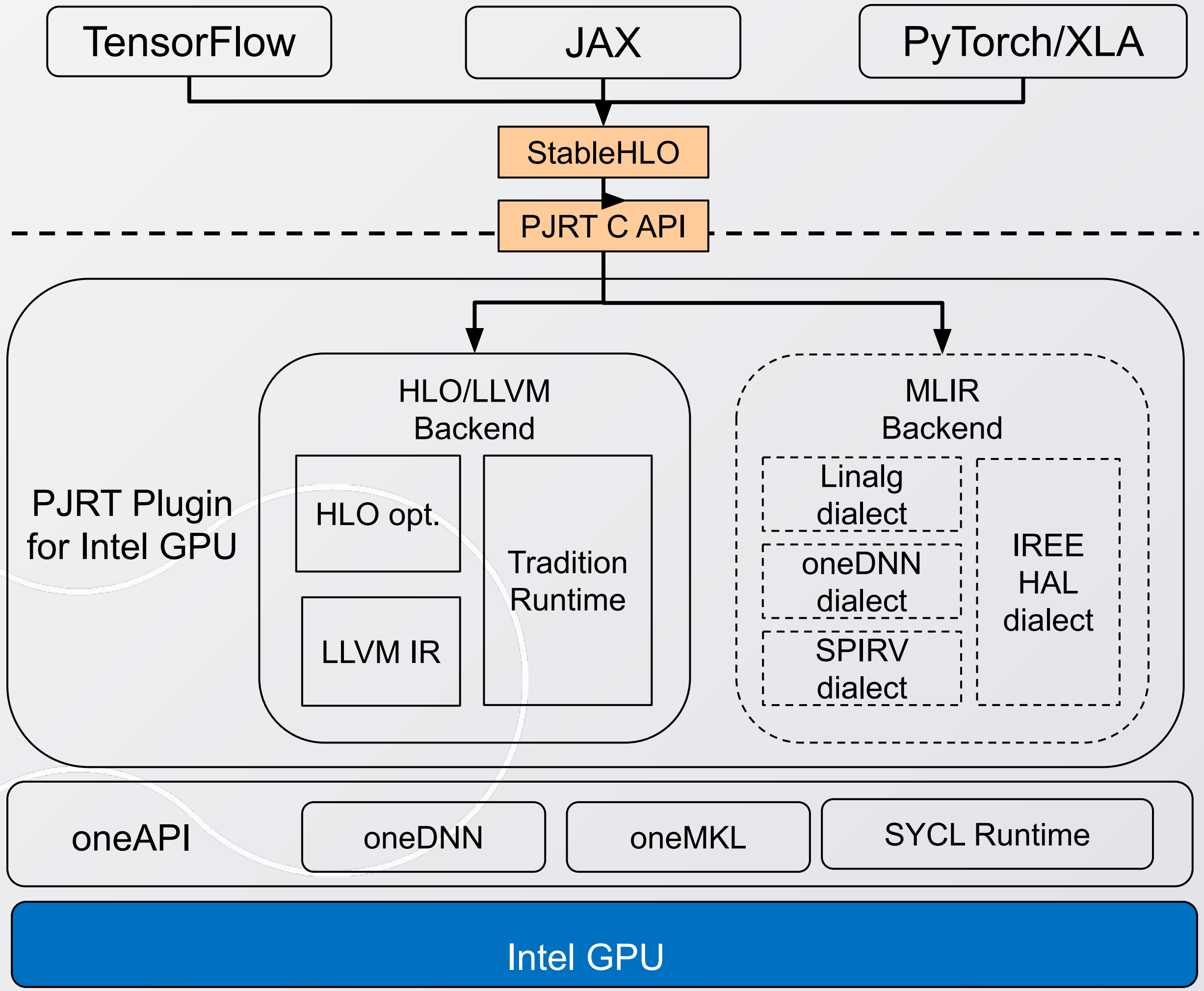


# PJRT - Future

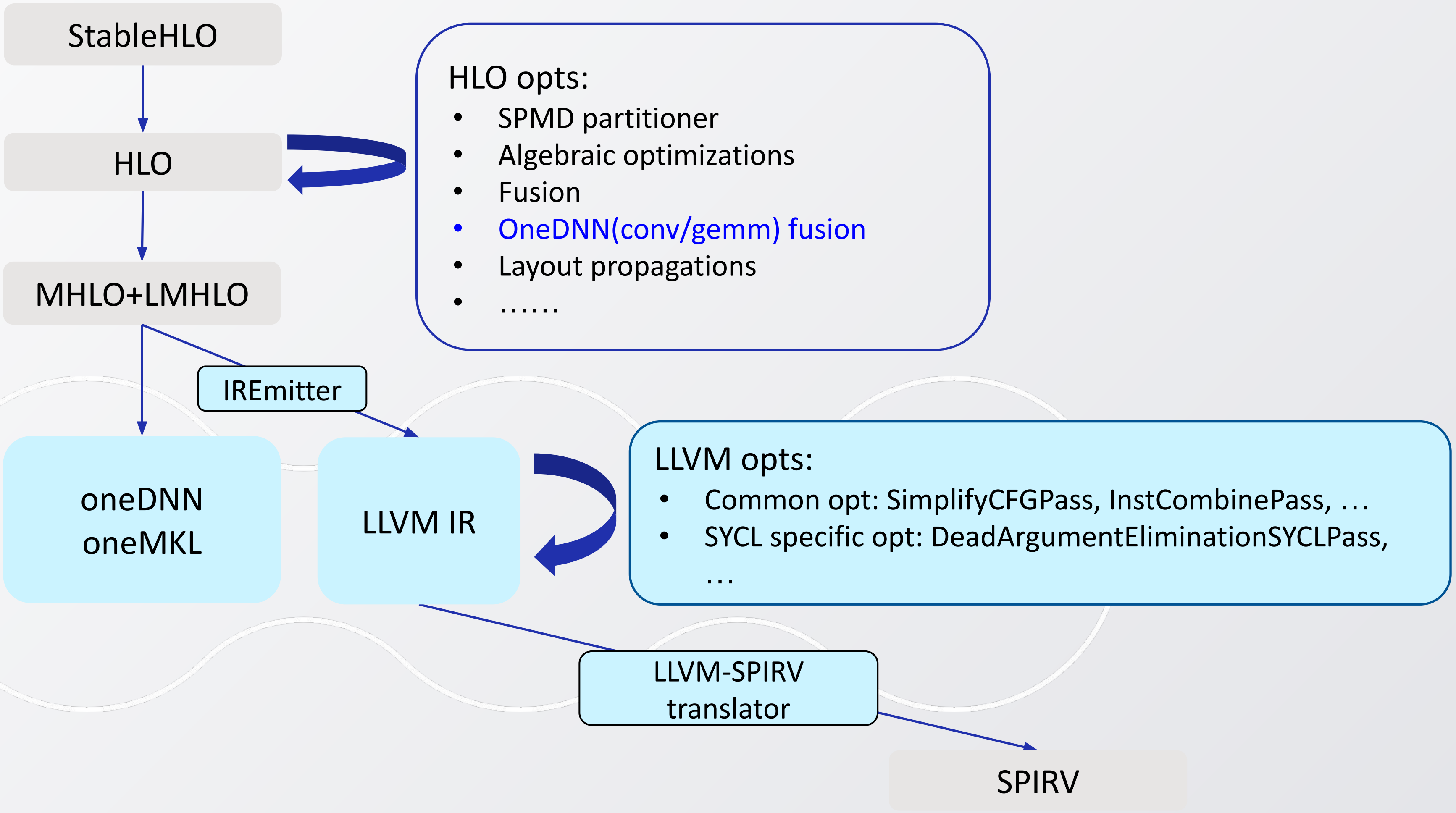
- Memory Space Support
    - Example use case: Nvidia Grace Hopper Superchip
- Sparsity Support
    - Enable ML compiler to optimize sparse computation
- MPMD Support
    - Enable more advanced Pipelining Parallelism

# PJRT Plugin for Intel GPU

- Intel GPU plugin integrates via PJRT API using oneAPI, and works on two code-gen techniques in parallel:
  - HLO/LLVM IR, release, runs JAX models
  - MLIR/IREE, experimenting, runs simple cases
- Release to support JAX ([link](link))



# PJRT Plugin for Intel GPU



# PJRT Plugin for Intel GPU --- Library Opt

- oneDNN
  - Conv/GEMM
  - Supported fusion pattern
    - Conv + [bias + add + activation]
    - [alpha *] GEMM (A, B) + [beta * C]
  - Graph API integration is coming soon to have more fusion capabilities
- oneMKL: Cholesky, FFT, TriangularSolve

**Python code:**

```
@jax.jit
def lax_conv_example(lhs,rhs):
  out = jax.lax.conv_with_general_padding(
          lhs, rhs, (1,1), ((0,0),(0,0)), (1,1), (1,1))
  out = jax.nn.relu(out)
  return out

lhs = lhs = np.random.randn(2,1,9,9).astype(np.float32)
rhs = np.random.randn(1,1,4,4).astype(np.float32)
lax_conv_example(lhs, rhs)
```

**StableHLO:**

```
module @jit_lax_conv_example {
  func.func public @main(%arg0: tensor<2x1x9x9xf32>, %arg1: tensor<1x1x4x4xf32>) ->
(tensor<2x1x6x6xf32>) {
    %0 = stablehlo.convolution(%arg0, %arg1) : (tensor<2x1x9x9xf32>,
tensor<1x1x4x4xf32>) -> tensor<2x1x6x6xf32>
    %1 = call @relu(%0) : (tensor<2x1x6x6xf32>) -> tensor<2x1x6x6xf32>
    return %1 : tensor<2x1x6x6xf32>
  }
  func.func private @relu(%arg0: tensor<2x1x6x6xf32>) -> tensor<2x1x6x6xf32> {
    %0 = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %1 = stablehlo.broadcast_in_dim %0, dims = [] : (tensor<f32>) ->
tensor<2x1x6x6xf32>
    %2 = stablehlo.maximum %arg0, %1 : tensor<2x1x6x6xf32>
    return %2 : tensor<2x1x6x6xf32>
  }
}
```

**Optimized HLO:**

```
ENTRY main.10 {
  constant_3 = f32[1]{0} constant({0})
  Arg_1.2 = f32[1,1,4,4]{3,2,1,0} parameter(1)
  Arg_0.1 = f32[2,1,9,9]{3,2,1,0} parameter(0)
  onednn-conv-bias-activation.1 = (f32[2,1,6,6]{3,2,1,0}, u8[0]{0})
custom-call(Arg_0.1, Arg_1.2, constant_3), backend_config="{\"activation_mode\":\"2\"}"
-> conv+relu fusion
  ROOT get-tuple-element = f32[2,1,6,6]{3,2,1,0}
get-tuple-element(onednn-conv-bias-activation.1), index=0
}
```

# PJRT Plugin for Intel GPU --- LLVM IR

**Python code:**

```
@jax.jit
def func_jit(a):
  a =
jnp.abs(jnp.sqrt(x))
  return a
```

**StableHLO module:**

```
module @jit_func {
  func.func public @main(%arg0:
tensor<1024xf32> {jax.arg_info = "x", …}) {
    %0 = stablehlo.sqrt %arg0 :
tensor<1024xf32> loc(#loc2)
    %1 = stablehlo.abs %0 : tensor<1024xf32>
loc(#loc3)
    return %1 : tensor<1024xf32> loc(#loc)
  } loc(#loc)
} loc(#loc)
```

**HLO module:**

```
%fused_computation (param_0.1: f32[1024]) -> f32[1024] {
  %param_0.1 = f32[1024]{0} parameter(0)
  %sqrt.0 = f32[1024]{0} sqrt(f32[1024]{0} %param_0.1),
metadata={op_type="Sqrt" …}
  ROOT %abs.0 = f32[1024]{0} abs(f32[1024]{0} %sqrt.0),
metadata={op_type="Abs" …}
}
ENTRY %func.8 (arg0.1: f32[1024])->f32[1024] {
  %arg0.1 = f32[1024]{0} parameter(0), …
  ROOT %fusion = f32[1024]{0} fusion(f32[1024]{0} %arg0.1),
kind=kLoop, calls=%fused_computation,
}
```

**Lmhlo module:**

```
module attributes {hlo.unique_id = 0 : i32, mhlo.unique_id = 0 : i64} {
  func @func(%arg0: memref<4096xi8> {lmhlo.params = 0 : index}, %arg1:
memref<4096xi8> {lmhlo.output_index = dense<> : tensor<0xi64>})
attributes {result_xla_shape = "f32[1024]{0}"} {
    %c0 = arith.constant 0 : index
    %0 = memref.view %arg0[%c0][] : memref<4096xi8> to memref<1024xf32>
    %c0_0 = arith.constant 0 : index
    %1 = memref.view %arg1[%c0_0][] : memref<4096xi8> to
memref<1024xf32>
    "lmhlo.fusion"() ({
      %2 = bufferization.to_tensor %0 : memref<1024xf32>
      %3 = "mhlo.sqrt"(%2) : (tensor<1024xf32>) -> tensor<1024xf32>
      %4 = "mhlo.abs"(%3) : (tensor<1024xf32>) -> tensor<1024xf32>
      memref.tensor_store %4, %1 : memref<1024xf32>
      "lmhlo.terminator"() : () -> ()
    }) : () -> ()
    "lmhlo.terminator"() : () -> ()
  }
}
```

# PJRT Plugin for Intel GPU --- LLVM IR

Difference with NVVM:

- Target data layout and triple
- Address space
- SPIRV builtin function
  - get_global_id, …
  - Subgroup shuffle, barrier
  - Math function: sqrt, expm

| Address Space | NVPTX Memory Space | SPIR-V Memory Space |
|---|---|---|
| 0 | Generic | Private |
| 1 | Global | Global |
| 2 | *Internal Use | Constant |
| 3 | Shared | Workgroup |
| 4 | Constant | Generic |
| 5 | Local | |

**LLVM IR:**

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-xxxxx"
target triple = "spir64-unknown-unknown"

define spir_func void @fusion(i8 addrspace(1)* noalias nocapture
readonly align 16 dereferenceable(4096) %alloc0, i8 addrspace(1)*
noalias nocapture writeonly align 128 dereferenceable(4096)
%alloc1) local_unnamed_addr !intel_reqd_sub_group_size !1 {
entry:
  %0 = call i64 @_Z12get_group_idj(i32 0)
  %block_id = trunc i64 %0 to i32
  %1 = call i64 @_Z12get_local_idj(i32 0)
  %thread_id_x = trunc i64 %1 to i32
  %2 = shl nuw nsw i32 %block_id, 10
  %linear_index = add nuw nsw i32 %2, %thread_id_x
  %3 = bitcast i8 addrspace(1)* %alloc0 to float addrspace(1)*
  %4 = zext i32 %linear_index to i64
  %5 = getelementptr inbounds float, float addrspace(1)* %3, i64 %4
  %6 = load float, float addrspace(1)* %5, align 4, !invariant.load
!2
  %7 = call float @_Z4sqrtf(float %6)
  %8 = call float @llvm.fabs.f32(float %7)
  %9 = bitcast i8 addrspace(1)* %alloc1 to float addrspace(1)*
  %10 = getelementptr inbounds float, float addrspace(1)* %9, i64
%4
  store float %8, float addrspace(1)* %10, align 4
  ret void
}
declare spir_func float @_Z4sqrtf(float) local_unnamed_addr #0
```

# Flax/JAX Stable Diffusion

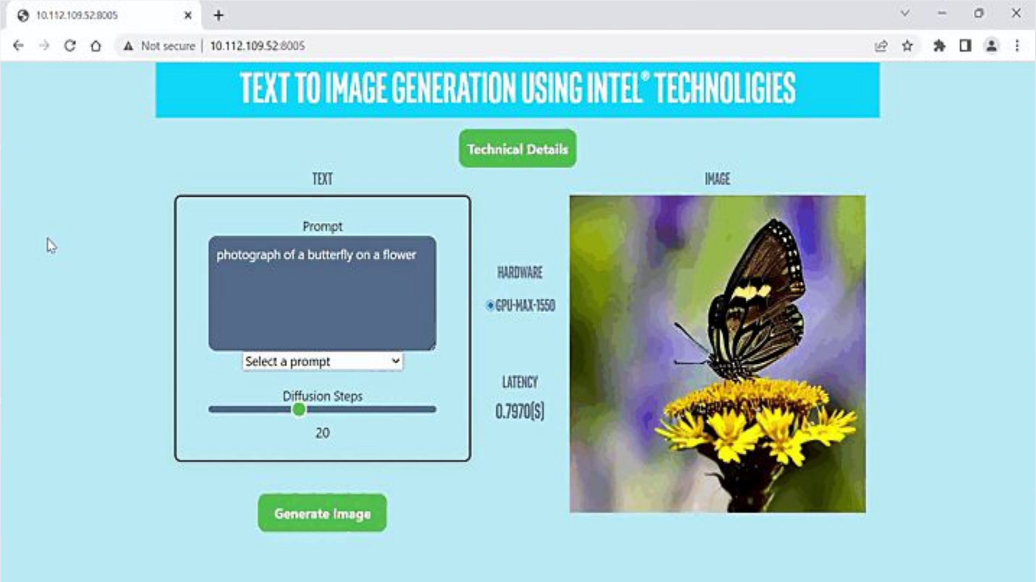
https://huggingface.co/CompVis/stable-diffusion-v1-4#jaxflax
No code change is required for Intel GPU

```
import jax, sys, time
import numpy as np
from flax.jax_utils import replicate
from flax.training.common_utils import shard
from diffusers import FlaxStableDiffusionPipeline

scheduler, scheduler_state =
FlaxDPMSolverMultistepScheduler.from_pretrained("CompVis/stable-diffusion-v1-4",
subfolder="scheduler")
pipeline, params = FlaxStableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4",
scheduler=scheduler, revision="bf16", dtype=jax.numpy.bfloat16)
params["scheduler"] = scheduler_state
prompt = "a photo of an astronaut riding a horse on mars"

prng_seed = jax.random.PRNGKey(0)
prompt = jax.device_count() * [prompt]
prompt_ids = pipeline.prepare_inputs(prompt)

params = replicate(params)
prng_seed = jax.random.split(prng_seed, jax.device_count())
prompt_ids = shard(prompt_ids)

def elapsed_time(nb_pass=10, num_inference_steps=20):
  # warmup
  images = pipeline(prompt_ids, params, prng_seed, num_inference_steps, jit=True).images
  start = time.time()
  for _ in range(nb_pass):
    _ = pipeline(prompt_ids, params, prng_seed, num_inference_steps, jit=True).images
  end = time.time()
  return (end - start) / nb_pass

print("Latency per image is: {:.3f}s".format(elapsed_time(nb_pass=5, num_inference_steps=20)))
```

```
$ export LD_LIBRARY_PATH="python-path/jaxlib/:$LD_LIBRARY_PATH"
$ export PJRT_NAMES_AND_LIBRARY_PATHS="xpu:/path/libitex_xla_extension.so"
$ numactl -N 0 -m 0 python jax_stable.py
```

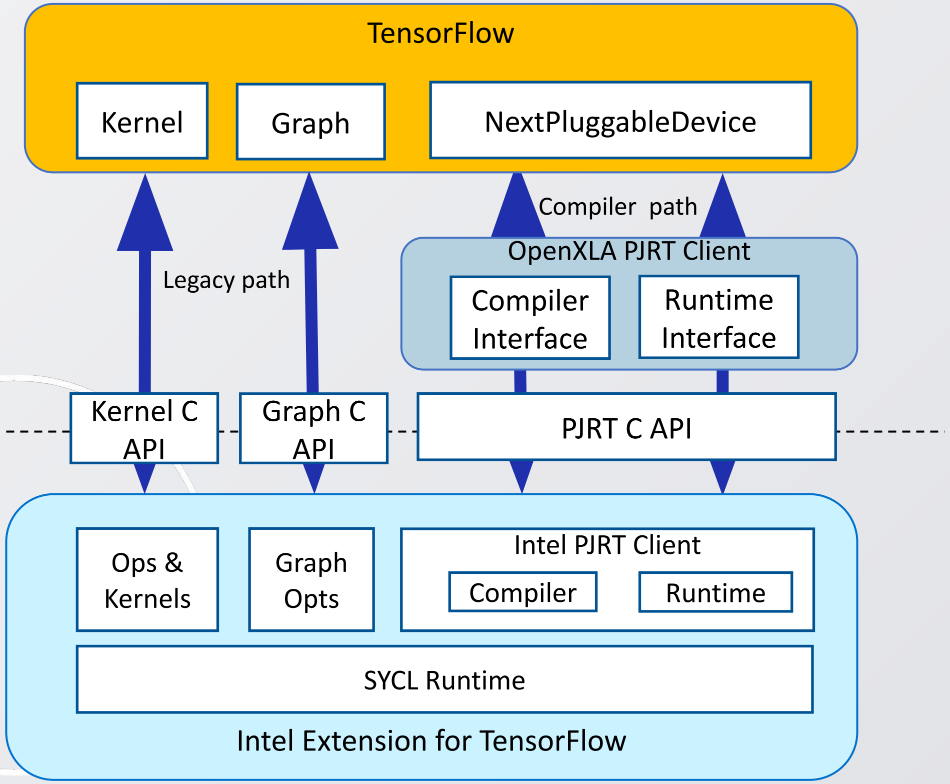
Latency per image is: 0.79s on Intel® Data Center Max GPU 1550

| Model | Precision | Diffusion Steps | Latency (s) |
|---|---|---|---|
| stable-diffusion-v1-4 | BF16 | 20 | 0.79 |
| | BF16 | 50 | 1.84 |



# NextPluggableDevice

- TensorFlow supports mixed execution mode
  - Run single TF model with both traditional and OpenXLA runtime
- Problems for TensorFlow plugins
  - PluggableDevice is StreamExecutor based, while OpenXLA is PJRT based
  - “*NO*” interoperation between them
- NextPluggableDevice solves this issue by extending PluggableDevice to use unified PJRT runtime for both



# Summary

- PJRT simplifies ML Hardware and Framework integration with unified API to support all frameworks (TensorFlow, JAX, PyTorch via PyTorch-XLA)
- Intel GPU plugin integrated with JAX via PJRT API using oneAPI and LLVM/ SPIR-V and demonstrates good performance on Intel® Data Center Max GPU

# Questions