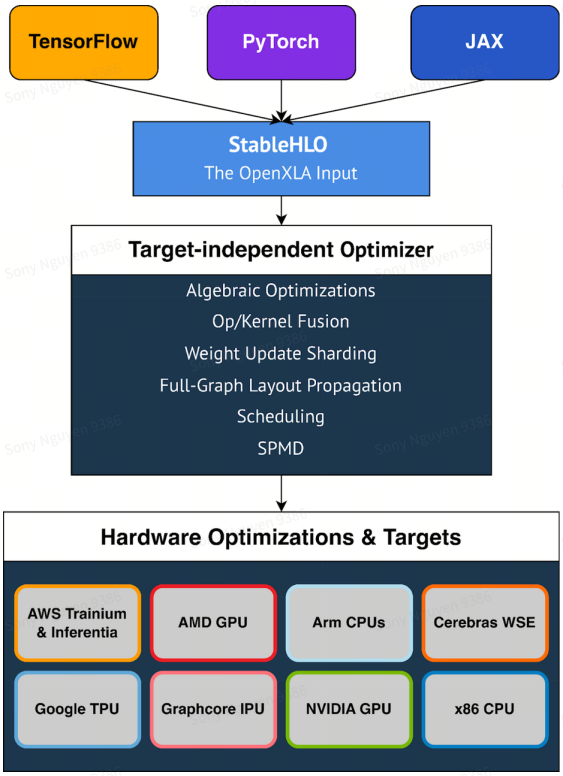
OpenXLA and IREE Notes

OpenXLA

https://opensource.googleblog.com/2023/03/openxla-is-ready-to-accelerate-and-simplify-ml-development.html



Our Solution and Goals

The OpenXLA Project provides a state-of-the-art ML compiler that can scale amidst the complexity of ML infrastructure. Its core pillars are performance, scalability, portability, flexibility, and extensibility for users. With OpenXLA, we aspire to realize the real-world potential of AI by accelerating its development and delivery.

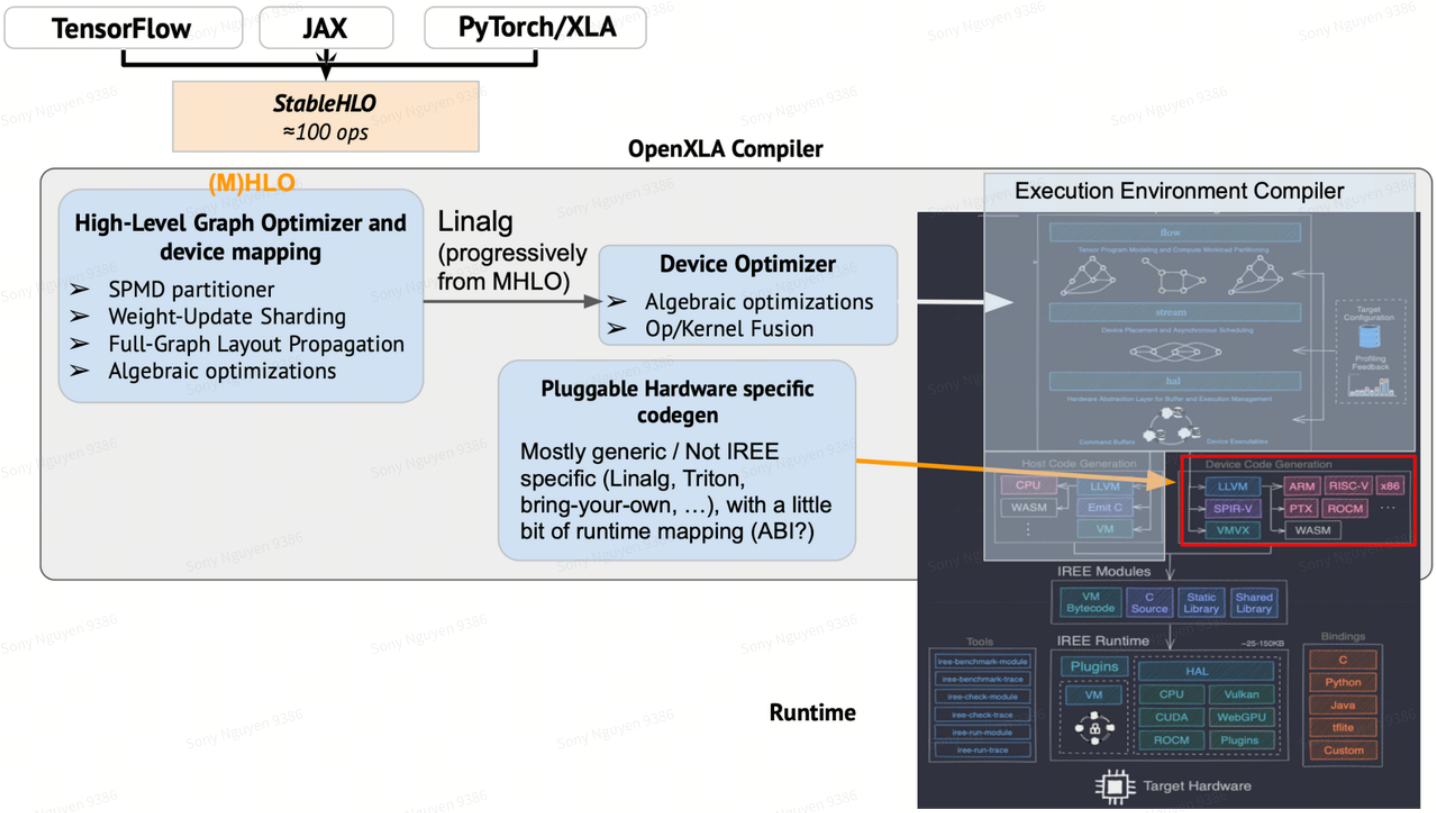
Our goals are to:

- Make it easy for developers to compile and optimize any model in their preferred framework, for a wide range of hardware through **(1)** a unified compiler API that any framework can target **(2)** pluggable device-specific back-ends and optimizations.
- Deliver industry-leading performance for current and emerging models that **(1)** scales across multiple hosts and accelerators **(2)** satisfies the constraints of edge deployments **(3)** generalizes to novel model architectures of the future.

- Build a layered and extensible ML compiler platform that provides developers with **(1)** MLIR-based components that are reconfigurable for their unique use cases **(2)** plug-in points for hardware-specific customization of the compilation flow.

OpenXLA and IREE integration

https://groups.google.com/a/openxla.org/g/openxla-discuss/c/DnPUmpyk4y0



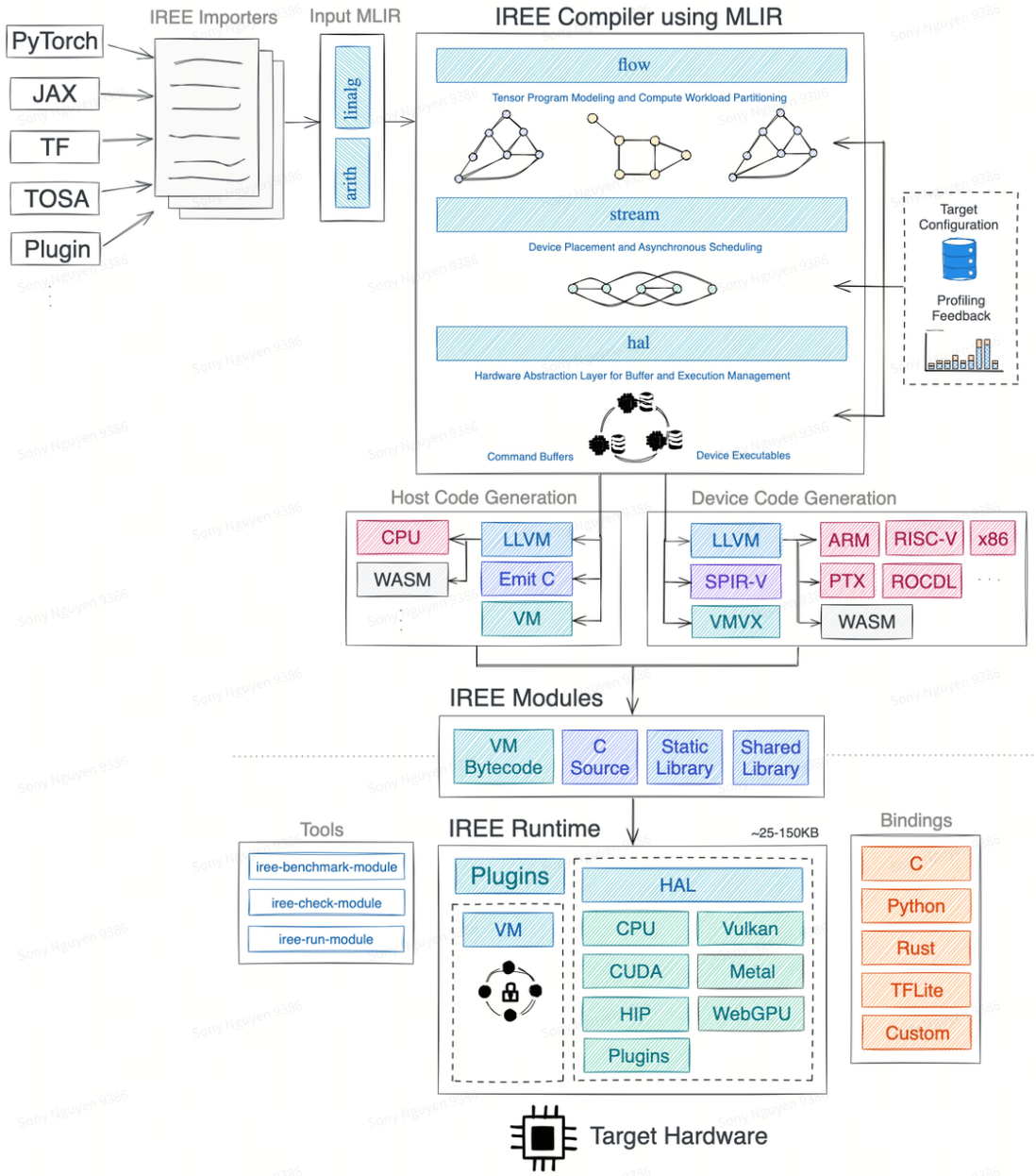
SPMD: https://en.wikipedia.org/wiki/Single_program,_multiple_data

The workflow looks like:

TensorFlow/PyTorch model --> StableHLO --> XLA Compiler --> IREE Compiler --> model.so --> IREE Runtime

IREE

https://iree.dev/#project-architecture

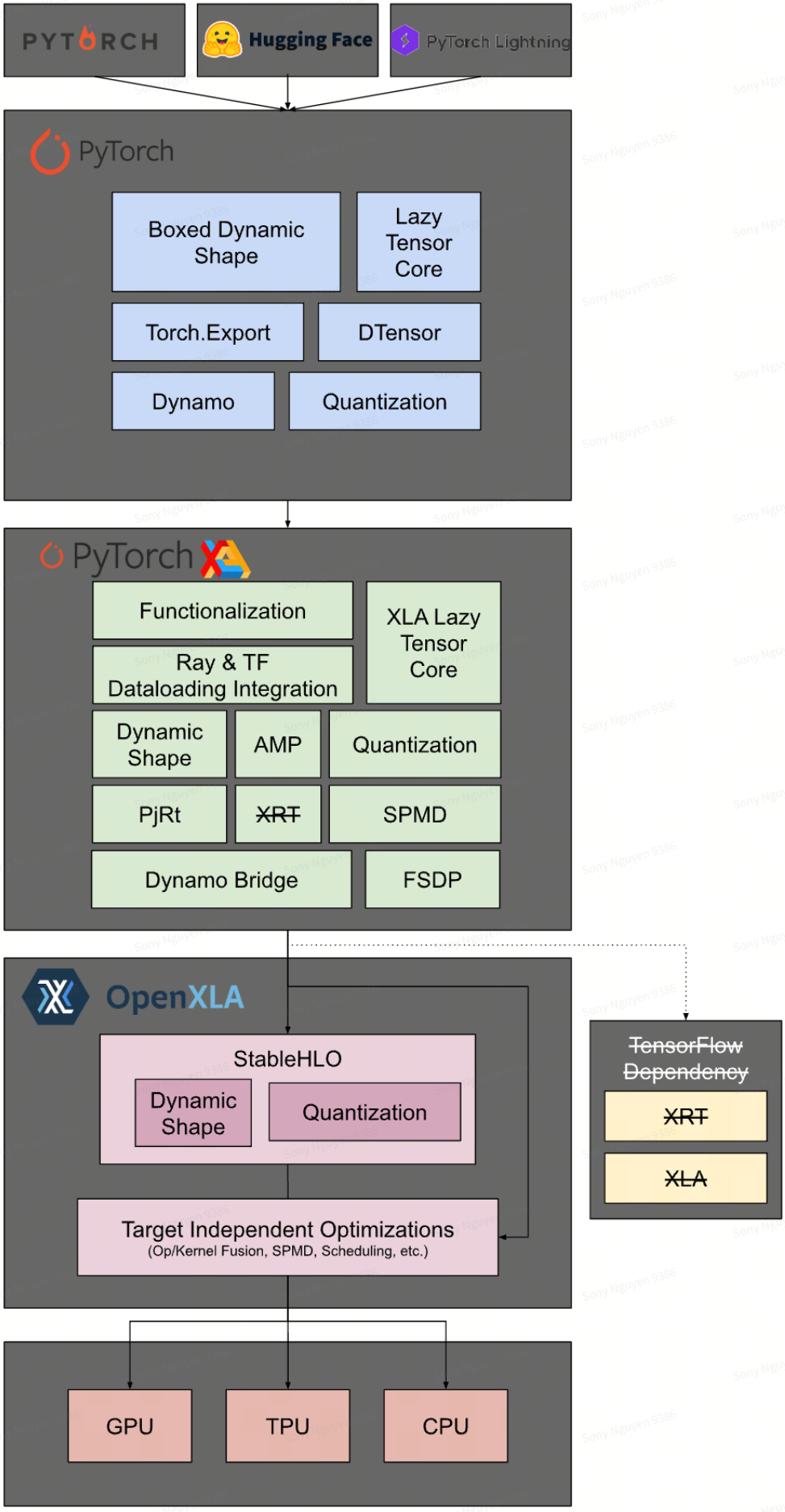


- **OpenXLA:** the whole project, I would describe it as "an ecosystem of ML Infrastructure modular components that can be assembled to form an e2e stacks targeting CPU/GPU with extension points enabling xPU targets"
- **StableHLO:** "a portability layer between ML frameworks and ML compilers, is an operation set for high-level operations (HLO) that supports dynamism, quantization, and sparsity. Furthermore, it can be serialized into MLIR bytecode to provide compatibility guarantees." It is a stable format that is intentionally decoupled for MHLO, which in turn is positioned as a compiler IR (no stability and different ergonomic goals).
- **OpenXLA compiler:** it is the component that takes StableHLO as input and generates an output for an execution environment. The out-of-the-box OpenXLA compiler execution environment is IREE and the IREE compiler uses extension points from the OpenXLA compiler to plug in as the execution environment. Other execution environments should be possible to plug-in for platforms which haven't adopted IREE.

- **High-level optimization and device mapping:** it is the component in the OpenXLA that operates at the full-graph level and performs transformations accordingly. It also considers the topology of the target (for multi-devices environments) and performs all sharding/partitioning necessary, and optimizes the cross-device communication scheduling to overlap computations. It is parameterized and customized for a given execution environment (platform/runtime/…).
- **Device Optimizer:** it is a point where the partitioning is complete and the code has a "single device" view of the program and optimizes accordingly. This is a level where some *linalg* fusions may happen for example (In cases where *linalg* is being used). This is composed of generic and reusable transformations, but this is likely invoked and customized by a particular "execution environment compiler" (like the IREE compiler) since there is a dance that starts to take place with lowering towards a particular environment (and possibly a particular platform).
- **Pluggable HW specific codegen:** this is a point where a single "fusion" or "dispatch" is handed over to the vendor plugin to generate the executable for a given fusion (e.g. to generate ptx/cubin), we can plug the Triton compiler as a codegen for specific kind of "fusions" here.
- **Execution Environment Compiler:** in OpenXLA this is the "IREE Compiler", it takes as input the output of the "High-level optimization and device mapping" and sets up the "Device Optimizer" according to its need (to lower as needed and transform it accordingly). Other environments are possible (a "no-runtime" embedded CPU environment compiler for example), which could reuse the "device Optimizer" but not map to the same kind of runtime abstractions as IREE.
- **Runtime:** isn't directly part of the compiler, it is the sets of components that are available on the target platform to allow execution of the resulting program. It is extensible in similar ways to the compiler, with different goals and constraints. There is a strong coupling between the execution environment compiler and the runtime.
- **MLIR:** this is an infrastructure providing tools for building compilers, with reusable dialects and codegen components. OpenXLA is built using MLIR, reusing the codegen components provided as much as possible, and contributing back any improvements or new components (including tools) developed for the need of OpenXLA.

PyTorch & OpenXLA: The Path Forward

https://pytorch.org/blog/pytorch-2.0-xla-path-forward/



https://openxla.org/stablehlo/tutorials/pytorch-export

PJRT

https://opensource.googleblog.com/2023/05/pjrt-simplifying-ml-hardware-and-framework-integration.html

Portability: Seamless Execution

The workflow to enable this vision with PJRT is as follows (shown in Figure 1):

1. The hardware-specific compiler and runtime provider implement the PJRT API, package it as a plugin containing the compiler and runtime hooks, and register it with the frameworks. The implementation can be opaque to the frameworks.
2. The frameworks discover and load one or multiple PJRT plugins as dynamic libraries targeting the hardware on which to execute the workload.
3. That’ s it! Execute the workload from the framework onto the target hardware.

The PJRT API will be backward compatible. The plugin would not need to change often and would be able to do version-checking for features.

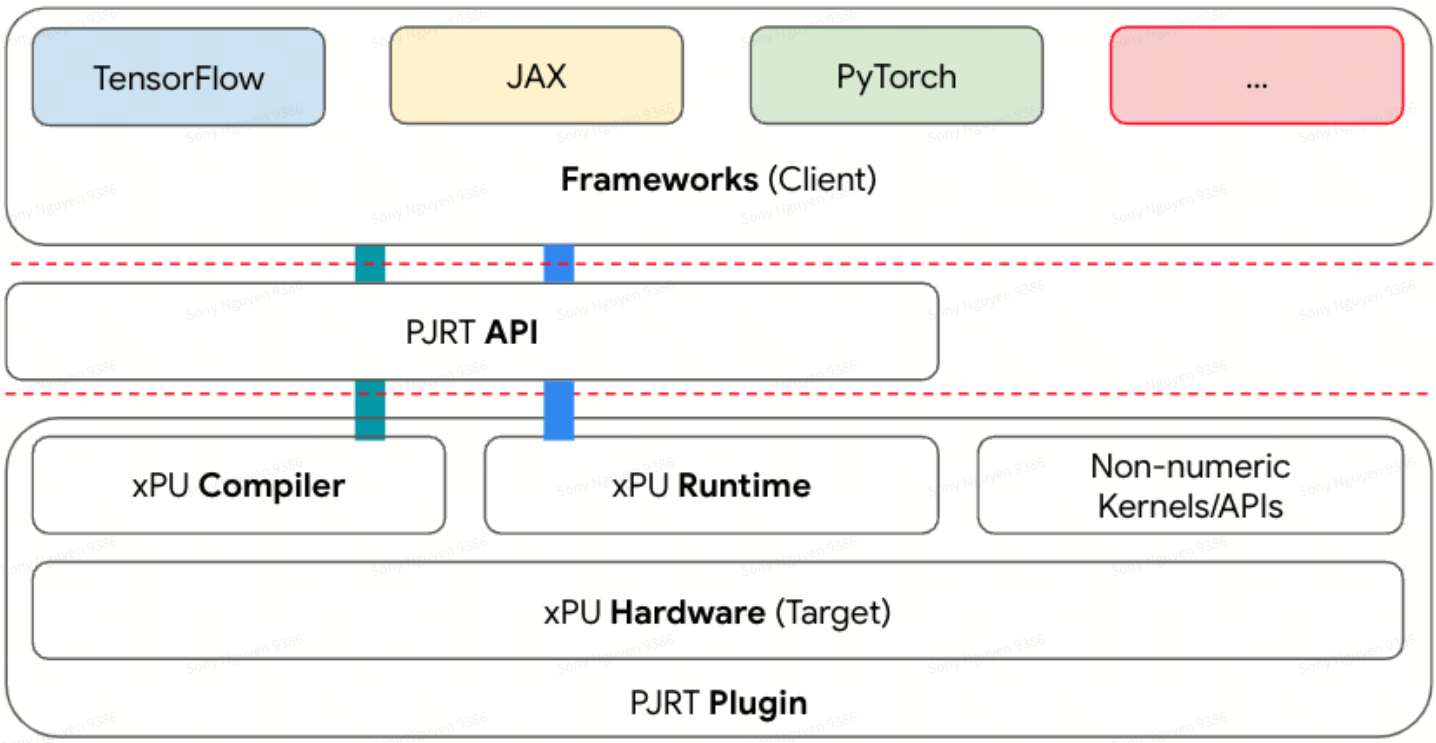


Figure 1: To target specific hardware, provide an implementation of the PJRT API to package a compiler and runtime plugin that can be called by the framework.

Cohesive Ecosystem

As a foundational pillar of the OpenXLA Project, PJRT is well-integrated with projects within the OpenXLA Project including StableHLO and the OpenXLA compilers (XLA, IREE). It is the primary

interface for TensorFlow and JAX and fully supported for PyTorch through PyTorch/XLA. It provides the hardware interface layer in solving the combinatorial framework x hardware ML infrastructure fragmentation (see Figure 2).

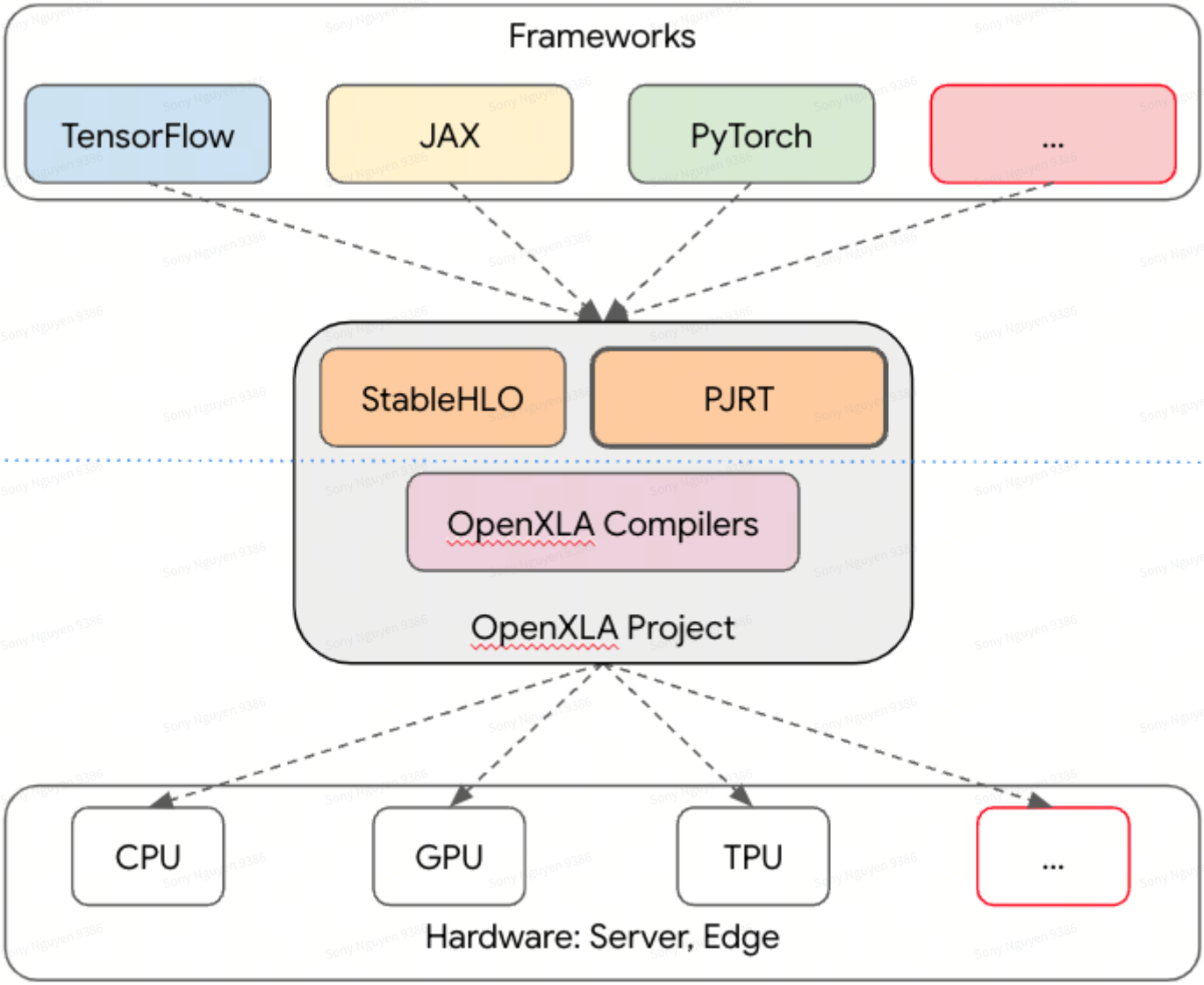


Figure 2: PJRT provides the hardware interface layer in solving the combinatorial framework x hardware ML infrastructure fragmentation, well-integrated with OpenXLA.