SIG OpenXLA Community Meeting

September 20, 2022

What is OpenXLA?

Open, state-of-art ML compiler ecosystem, built collaboratively with Hardware & Software partners, using the best of XLA & MLIR.

Agenda

- Welcome! (10 min)
  - New and existing collaborators
  - SIG collaboration channels
- StableHLO update (10 min) by Eugene Burmako
- XLA Runtime overview (25 min) by Eugene Zhulenev
- Q&A (10 min)
- Next steps (5 min)

Introductions & housekeeping

Welcome!

- Welcome to any new attendees? What are you looking to focus on?
- SIG Member Orgs:
  - AMD
  - Apple
  - ARM
  - AWS
  - Google (XLA, TensorFlow, JAX, PyTorch/XLA)
  - Intel
  - Meta
  - NVIDIA

OpenXLA Meetings

- Monthly on Zoom
- Rotating meeting host & scribe
- Proposed agenda shared by host week prior in community wiki
- Meeting minutes & slides shared publicly on community wiki day after
- Meetings should include:
  - Development updates: this week **XLA Runtime** deep dive!
  - Design proposals
  - Community topics

SIG Collaboration

A quick overview for our new collaborators

Collaboration channels

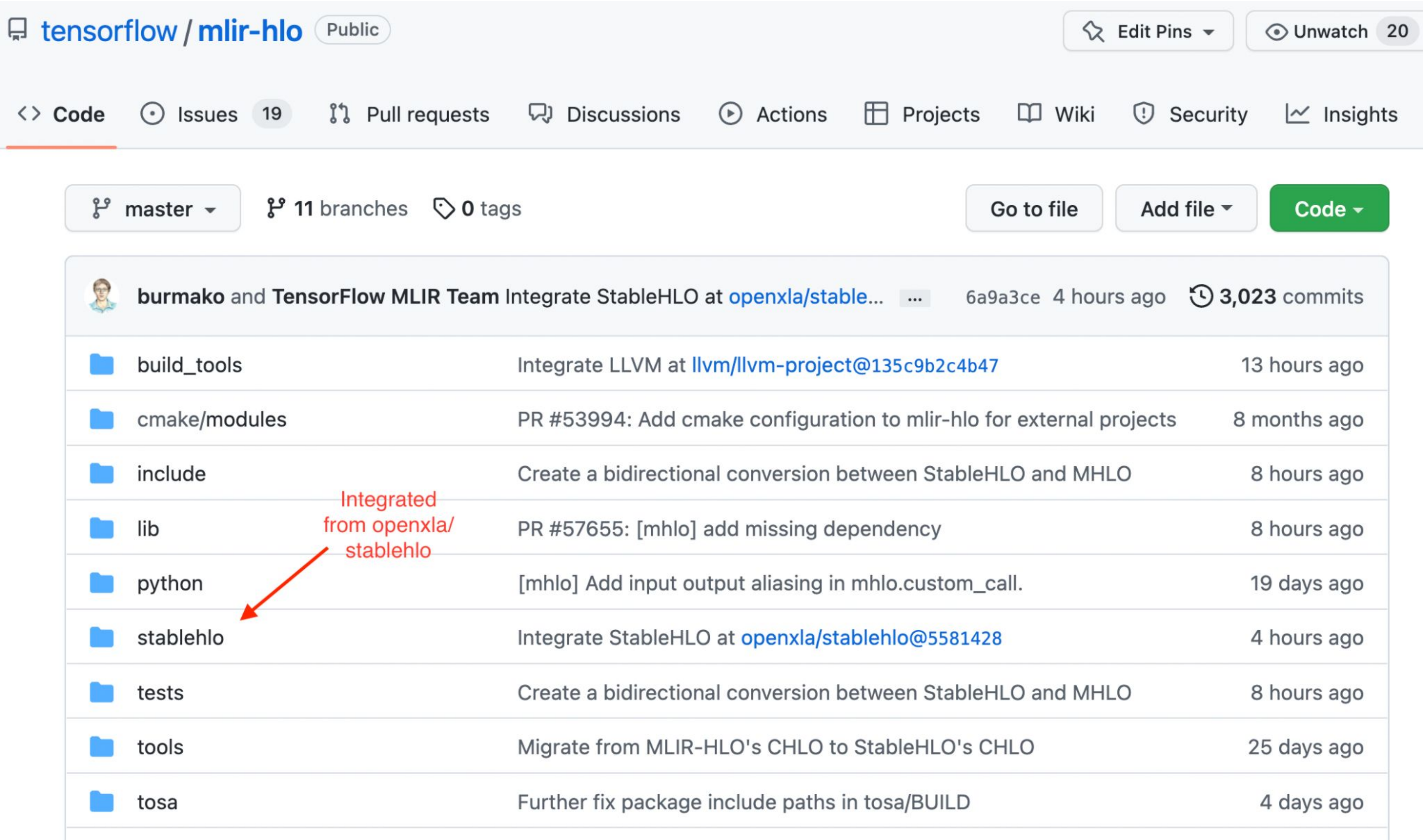
Channel	Content	Access	Archive
OpenXLA GitHub Org	Code, Design proposals, PRs, Issues, Roadmaps	Public	N/A
Community Repo	Governance, Meetings, Code of conduct	Public	Public
GitHub Discussions (Community)	Meta discussions on openxla/community repo	Public	Public
GitHub Discussions (Technical)	Technical discussions on individual repos: xla, stablehlo	Public	Public
Discord Server	Sync discussions	Open invites	Archived chats
SIG Meetings	Monthly live meetings	Public	Public agenda, slides, meeting minutes
SIG Google Drive	*Shared docs, decks*	*Read-only to non members*	*Indefinite*

Technical Updates

StableHLO

openxla/stablehlo

- Tons of momentum! 🔥
- 74 issues & 83 pull requests since creation about a month ago.
- Main areas of development
  - Specification & interpreter.
  - Completeness of verification and shape inference.
  - Compatibility guarantees.
  - Integration into MLIR-HLO.


tensorflow / mlir-hlo
Public
Edit Pins
Unwatch 20
Code
Issues 19
Pull requests
Discussions
Actions
Projects
Wiki
Security
Insights
master
11 branches
0 tags
Go to file
Add file
Code
burmako and TensorFlow MLIR Team Integrate StableHLO at openxla/stable...
6a9a3ce 4 hours ago
3,023 commits
build_tools
Integrate LLVM at llvm/llvm-project@135c9b2c4b47
13 hours ago
cmake/modules
PR #53994: Add cmake configuration to mlir-hlo for external projects
8 months ago
include
Create a bidirectional conversion between StableHLO and MHLO
8 hours ago
lib
PR #57655: [mhlo] add missing dependency
8 hours ago
python
[mhlo] Add input output aliasing in mhlo.custom_call.
19 days ago
stablehlo
Integrate StableHLO at openxla/stablehlo@5581428
4 hours ago
tests
Create a bidirectional conversion between StableHLO and MHLO
8 hours ago
tools
Migrate from MLIR-HLO's CHLO to StableHLO's CHLO
25 days ago
tosa
Further fix package include paths in tosa/BUILD
4 days ago
Integrated from openxla/ stablehlo

StableHLO

supersedes MHLO as compiler interface

- Same repo.
- Same ops*
- Comes with compatibility guarantees.
- Thanks to conversions, MHLO is just one hop away.

* Except for 10 MHLO ops which aren't used as an interface between ML frameworks and ML compilers

Compatibility proposal

1. libStablehlo provides 6 months of backward compatibility.
2. libStablehlo provides 3 weeks of forward compatibility.
3. Source compatibility for C, C++ and Python APIs within libStablehlo is an aspirational goal.

Learn more from the RFC on GitHub - a super detailed document that goes through supported ops and evolution scenarios!

Author ▾ | Label ▾ | Projects ▾ | Milestones ▾ | Assignee ▾ | Sort ▾

Move CrossReplicaSum from StableHLO to CHLO. ✓ RFC — 1

#118 opened 6 days ago by subhankarshah • Changes requested

StableHLO Compatibility Spec Proposal ✓ RFC — 21

#115 opened 7 days ago by GleasonK • Review required

[Feature Request] Support Scalar in StableHLO RFC

#43 opened 26 days ago by yaochengji

[Feature Request] Custom Call with no_side_effect Trait RFC — 2

#42 opened 26 days ago by yaochengji

Unify different versions of round op (viz., stablehlo::round_nearest_afz & stablehlo::round_nearest_even) into one. RFC — 2

#35 opened 27 days ago by sdasgup3

Next steps

- Bootstrapping of the RFC process.
- Adoption by *HLO producers: JAX, PyTorch/XLA, TensorFlow.
- Conversations with ONNX-MLIR and Torch-MLIR.

Next steps

- Bootstrapping of the RFC process.
- Adoption by *HLO producers: JAX, PyTorch/XLA, TensorFlow.
- Conversations with ONNX-MLIR and Torch-MLIR.
- Collaboration & integration with TCP

XLA Runtime

An overview of the XLA's new runtime

As of today, XLA:GPU is ready to flip the switch and XLA:CPU is on its way.

Runtime for XLA Today

- **XLA:CPU** compiles to a **native X86 function pointer** with arguments passes as void** pointer

- **XLA:GPU** compiles to a **ThunkSequence** which is interpreted sequentially. This can be viewed as a very simple VM or interpreter with opcodes that do not have arguments or results.
    - Thunk kind: Cholesky, Fft, Gemm, Convolution, … (~20)
    - *Cons*: **(1)** All or nothing approach for running HLO programs **(2)** All operations must be executed on a device **(3)** All arguments and results must be in device memory. For example it is impossible to have a while loop with predicate computed on the host. **(4)** Data between thunks can be passed only through device buffers (thunks have not args or results).

- New **XLA Runtime** aims to unify the "executable artifact" produced by different XLA backends
    - With new runtime every HLO program will be lowered to "XLA runtime executable"
    - We can easily compile HLO programs to heterogeneous executables running on host and device(s). Use XLA:CPU compiler for host code and XLA:GPU for device code.

XLA Runtime for XLA Programs

- XLA programs (HLO modules) compiled to native functions with a help of LLVM JIT compiler (XLA:GPU ThunkSchedule is compiled to X86 function, XLA:CPU will compile to a slightly different X86 function)

- XLA Runtime provides high level C++ API to the user and takes care of matching ABI of the compiled XLA program so that users do not need to know any of the internal details

- XLA Runtime provides a small runtime that supports executing compiled XLA programs, similar to how libc (standard C library) provides a small “runtime” that supports executing C programs
  - API for returning values and errors to the caller
  - API for launching concurrent tasks

- XLA Runtime defines it’s own FFI mechanism with custom encoding/decoding for arguments and result (to decouple from platform dependent ABI):
  - Users of XLA Runtime can implement their own “custom call libraries” on top of this FFI mechanism to implement custom “Hardware Abstraction Layer” (e.g. Stream Executor HAL)

XLA Runtime Workflow

Compilation Pipeline → Executable → Run time

Compilation Pipeline

- User declares what MLIR dialects are legal in the input ML Model (for XLA:GPU it's LMHLO)
- User provides a compilation/specialization pipeline from input dialects to LLVM
- User provides a custom call convention (how to map high level types into types with an ABI)
- Compilation pipeline is responsible for outlining regions of code for accelerated execution (e.g. forming GPU kernels)

Executable

- ELF Library (can be saved on disk and shipped over the network)
- Entrypoint function signature
- Host-side executable program is just a regular function
- Device kernels stored in the constants section (PTX, CUBIN, TPU binary, etc is just a byte array defined as an LLVM global)

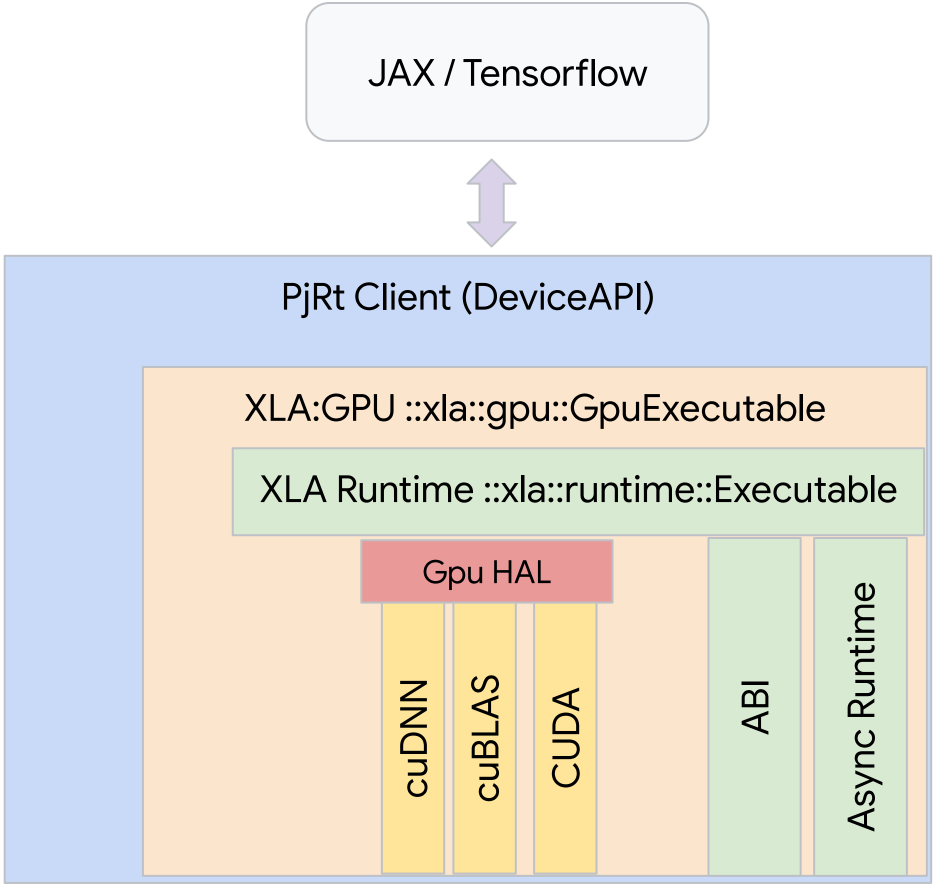
Run time

- XLA runtime relies on LLVM ORC dynamic linker to load compiled executables
- XLA runtime by default links functions for returning results and errors and launching async tasks
- User is responsible for providing libraries enabling compiled executable to talk with the accelerator (bring your own HAL)
- XLA:GPU for example defines its own "GPU hardware abstraction layer"

PjRt, XLA and XLA Runtime

- XLA Runtime is an implementation detail of the XLA:GPU executable, not visible to the XLA users

- XLA dynamically links jit-compiled (or loaded AOT compiled) executables with a “runtime support” libraries (XLA:GPU hardware abstraction layer)



XLA:GPU <-> XLA Runtime

XLA at run time links compiled program with a support library

```
func private @xla.gpu.gemm(%a: memref<?x?xf32>,
                           %b: memref<?x?xf32>,
                           %c: memref<?x?xf32>)
  attributes { rt.custom_call = "xla.gpu.gemm" }

func private @xla.gpu.launch(%arg: memref<?x?xf32>)
  attributes { rt.custom_call = "xla.gpu.launch"}

func @main(%arg0: memref<?x?xf32>,
           %arg1: memref<?x?xf32>,
           %arg2: memref<?x?xf32>) {
  call @xla.gpu.gemm(%arg0, %arg1, %arg2)
  call @xlu.gpu.launch(%arg2) { binary = "<cubin>" }
  return
}
```

```
// Calls into the cuBLAS Gemm implementation
absl::Status Gemm(
  StreamExecutor* stream_executor,
  MemRef a, MemRef b, Memref c) {
 ...
}
```

```
// Launch device kernel on the Gpu
absl::Status Launch(
   StreamExecutor* stream_executor,
   RepeatedArgs args, Cubin binary) {
 ...
}
```

XLA Runtime FFI (Type-Checked Custom Calls)

```
Status GemmImpl(StreamExecutor* exec,
  MemRefView a, MemRefView b,
  MemRefView c, float alpha);
```

Implement your runtime support library in C++

```
bool Gemm(runtime::ExecutionContext* ctx,
          void** args, void attrs*) {
  static auto* hdl = CustomCall::Bind("gemm")
                       .UserData<StrExec*>()
                       .Arg<MemRefView>()
                       .Arg<MemRefView>()
                       .Arg<MemRefView>()
                       .Attr<float>("alpha")
                       .to(GemmImpl);
  return hdl->call(args, attrs, UserData(ctx));
}
```

All custom library calls use unified ABI when linked with the executable

Define a binding that decodes opaque arguments and attributes and calls into the C++ implementation

```
DirectCustomCallLibrary XlaGpuRuntimeLib() {
 DirectCustomCallLibrary lib;
 lib.Insert("xla.gpu.gemm", &Gemm);
 return lib;
}
```

Package custom calls into a library, e.g. “XLA Gpu Runtime”

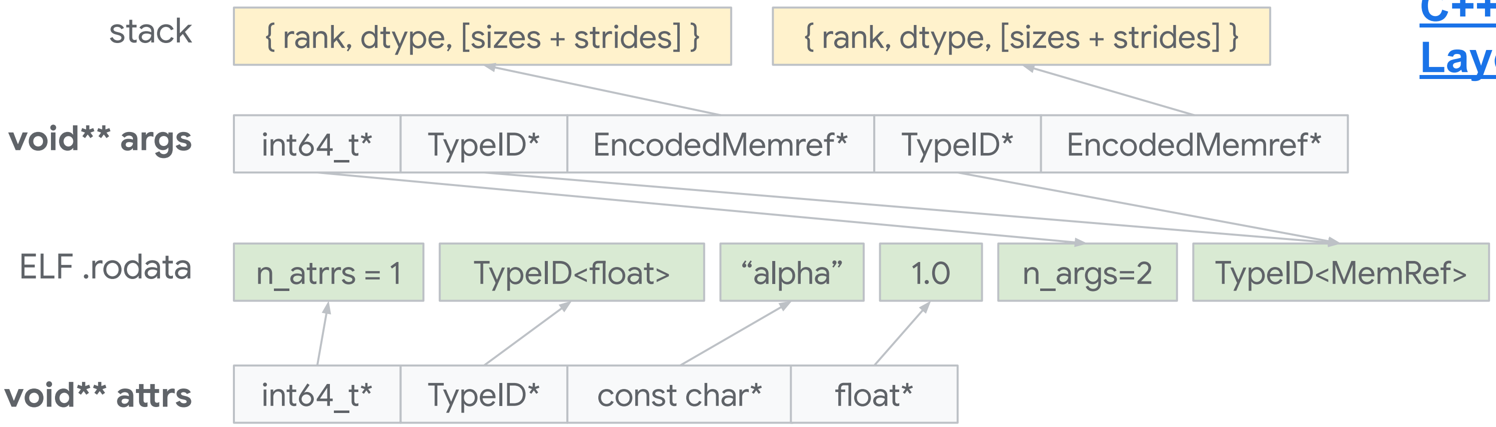
XLA Runtime FFI ABI (Type-Checked Custom Call ABI)

```
func @main(%arg0: memref<?x?xf32>,
           %arg1: memref<?x?xf32>,
           %arg2: memref<?x?xf32>) {
  call @xla.gpu.gemm(%arg0, %arg1, %arg2)
    { alpha = 1.0 : f32}
  return
}
```

```
llvm.struct<(i8, i8, ptr<i8>, array<i64>>
```

```
struct EncodedMemref {
 uint8_t dtype;
 uint8_t rank;
 void* data;
 int64_t dims[]; // C99 flexible array member
};
```

C++ Standard Layout



Extensible Encoding for Custom Args and Attrs

```
func @main(...) {
  call @xla.gpu.conv(%arg0, %arg1, %arg2)
     { activation = #mhlo.activation<Relu>,
       window = #mhlo.window<strides = [1, 1],
                              dilation = [2, 2]>
     }
}
```

```
// Encode `WindowAttr` as an aggregate attribute.
encoding.Add<AggregateAttrEncoding<mhlo::WindowAttr, Window>>(
     encoding, AggregateAttrDef<mhlo::WindowAttr>()
       .Add("strides", &mhlo::WindowAttr::getStrides)
       .Add("dilation", &mhlo::WindowAttr::getDilation));
```

```
enum class Activation : uint8_t {
  kSoftmax, kRelu, kRelu6, kTanh
};

struct Window {
  ArrayRef<int64_t> strides;
  ArrayRef<int64_t> dilation.
};
```

```
XLA_REGISTER_ENUM_ATTR_DECODING(Activation);

XLA_REGISTER_AGGREGATE_ATTR_DECODING(Window,
    XLA_AGGREGATE_FIELDS("strides", "dilation"),
    ArrayRef<int64_t>, ArrayRef<int64_t>);
```

```
CustomCall::Bind("conv").
  ....
  .Attr<Activation>("activation")
  .Attr<Window>("window")
  .To([](..., Activation activation, Window window) {...});
```

XLA at run time will type check the attributes and also verify that the struct has correct fields names and type

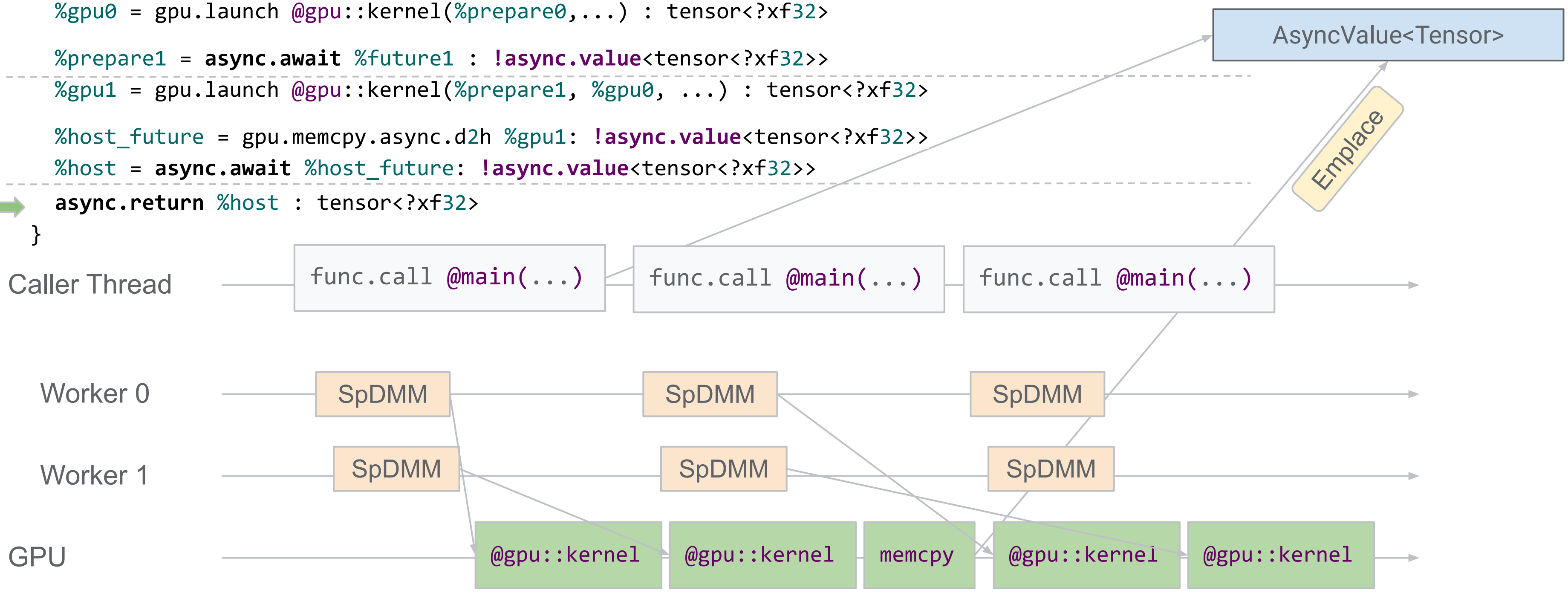
Integration with Async Runtime

```
async.func @main(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>,...) -> !async.value<tensor<?xf32>> {
  %future0 = async.execute { SparseDenseMatMul(%arg0, ...) } : !async.value<tensor<?xf32>>
  %future1 = async.execute { SparseDenseMatMul(%arg1, ...) } : !async.value<tensor<?xf32>>

  %prepare0 = async.await %future0 : !async.value<tensor<?xf32>>
  %gpu0 = gpu.launch @gpu::kernel(%prepare0,...) : tensor<?xf32>

  %prepare1 = async.await %future1 : !async.value<tensor<?xf32>>
  %gpu1 = gpu.launch @gpu::kernel(%prepare1, %gpu0, ...) : tensor<?xf32>

  %host_future = gpu.memcpy.async.d2h %gpu1: !async.value<tensor<?xf32>>
  %host = async.await %host_future: !async.value<tensor<?xf32>>
  async.return %host : tensor<?xf32>
}
```



Current Status And Roadmap

XLA:GPU

- Passes all internal tests
- Will be enabled by default in ~October
- End-to-end compilation flow without leaving MLIR in 2023?

XLA:CPU

- Work in progress, passes ~60% of internal tests
- Plan is to get correctness parity with current XLA:CPU in Q4
- New runtime opens opportunities for inter-op concurrency within XLA programs, this will be a focus in 2023

XLA

- New runtime makes it easy to execute heterogeneous XLA programs, e.g. partition HLO module into CPU and GPU computations and run them together
- Lower level of the stack can be unified between all XLA backends

Code is available on GitHub

Q&A

Next steps

Next steps

- Google to send out update on OpenXLA workstreams:
    - Project and product messaging proposal (Sept 22)
    - Standalone repository progress (week of Sept 26)
    - Governance model proposal (week of Sept 26)
    - New OpenXLA logo proposal (week of Sept 26)

- Follow-up GitHub discussion on XLA Runtime (Sept 20)

- Request for marketing approvals to announce the **SIG launch**

- Community feedback for next SIG meeting and technical deep dive

Thank you!

SIG OpenXLA Marketing

SIG OpenXLA Co-marketing Overview

2022

- Formally launch the formation of **SIG OpenXLA** to the broader public
- Align on the core messaging for OpenXLA (SIG + products)
- Align on partner-specific messaging policies
- Identify neutral and partner-specific events and marketing channels for 2022 (SIG launch) and 2023 (product-focused)

2023

- Announce community roadmap and product vision (“product launch”)
- Jointly promote OpenXLA in neutral and partner-specific forums