SIG OpenXLA Community Meeting

November 15, 2022

OpenXLA

What is OpenXLA?

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

Introductions

Welcome!

- Welcome to any new attendees? What are you looking to focus on?
- SIG member organizations:
  - Alibaba
  - AMD
  - Apple
  - ARM
  - AWS
  - Google
  - Intel
  - Meta
  - NVIDIA

SIG Collaboration

Reference material for our new collaborators

Our Meetings

- Monthly on Zoom, 3rd Tuesday @ 8AM PT
- Rotating meeting host & scribe
- Proposed agenda shared by host week prior in GitHub Discussions
- Meeting minutes & slides shared publicly in the meetings archive the day after
- Meetings should include:
  - Development updates
  - Design proposals
  - Community topics

Our Collaboration Channels

Channel	Content	Access	Archive
GitHub organization	Code, Design proposals, PRs, Issues, Roadmaps	Public	N/A
Community repository	Governance, Meetings, Code of conduct	Public	Public
Community discussions	Meta discussions on openxla/community repo	Public	Public
Technical discussions	Technical discussions on individual repos: xla, stablehlo	Public	Public
Discord	Sync discussions	Open invites	Archived chats
Community meetings	Monthly live meetings	Public	Public agenda, slides, meeting minutes

Development Updates

FP8 in XLA and StableHLO

Reed Wanderman-Milne, Google

reedwm@google.com

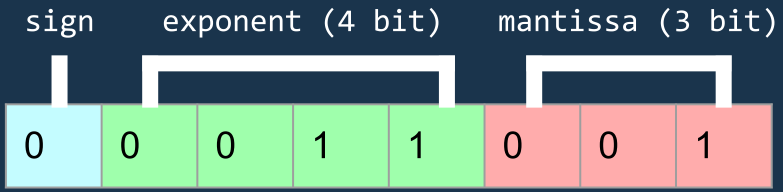
OpenXLA

Introduction

- NVIDIA Hopper GPUs supports two FP8 dtypes
  - E4M3: 4 exponent bits, 3 mantissa bits. Used on forward pass
  - E5M2: 5 exponent bits, 2 mantissa bits. Used on backward pass
- FP8 RFC: https://github.com/openxla/xla/discussions/22
- The RFC proposes supporting these dtypes in XLA and StableHLO
  - The RFC design may still change. Please comment if you have suggestions!
  - RFC proposes adding the FP8 dtypes supported by NVIDIA/Intel/ARM
- Here I present the initial FP8 design that the RFC proposes. The RFC also has details on how the FP8 design may evolve in the future to use StableHLO's quantized types and ops.

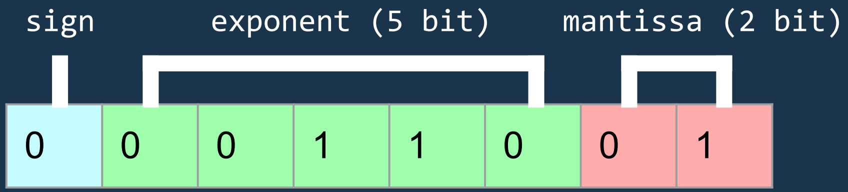
FP8 dtypes

FP8 E4M3



- Range: $[2^{-9}, 448]$
- Used on forward pass

FP8 E5M2



- Range: $[2^{-16}, 57344]$
- Used on backward pass

Scaling

- FP8 easily underflows/overflows!
- To prevent this, each tensor needs a scale
- Recall: E4M3 range is $[2^{-9}, 448]$.

FP32 tensor

2^{-14}	1
3	7

Divide by scale of 2^{-6} →

← Multiply by scale of 2^{-6}

Scaled FP8 E4M3 tensor

2^{-8}	64
192	448

- Optimal scale: max(fp32_tensor) / max_fp8_val

Scaling is computed dynamically during training

- Cannot perforantly compute an optimal scale and use it the same step
- Instead, each step computes the scale for the next step, uses the scale computed in the previous step

Step 0	Step 1	Step 2
• Uses an arbitrary initial scale: say 2^{-6} • Computes scale for Step 1	• Uses scale computed in Step 0 • Computes scale for Step 2	• Uses scale computed in Step 1 • Computes scale for Step 3

FP8 changes to StableHLO/MHLO/HLO

- Add two dtypes: `f8E5M2`, `f8E4M3`
- Support passing tensors of these dtypes to ops that support other floating-point dtypes (which is almost all ops)
- That's it!
- But how do we represent scaling?

How we represent scaling

- Use multiply and divide ops to scale
- Two ways to represent op with scaling: We first show the “generic” approach which works for all ops
- Key idea: Can never have unscaled FP8 tensor
  - So convert to FP16 first, then run op with FP16 input(s)

```
def quantized_dot_generic(x_f8, x_scale, z_scale):
  x_f16_unscaled = cast(x_f8, f16) * x_scale          # Step 1, 2
  z_f16_unscaled = dot(x_f16_unscaled, x_f16_unscaled)  # Step 3
  z_max = max(abs(z_f16_unscaled))                     # Step 4
  z_f8 = cast(z_f16_unscaled / z_scale, f8E4M3)        # Step 5, 6
  z_new_scale = zmax / 448
  return z_f8, new_scale
```

1. Cast inputs to FP16
2. Unscale inputs
3. Run the op
4. Compute max of output
5. Scale output
6. Cast output to FP8

How we represent scaling

- Use multiply and divide ops to scale
- Two ways to represent op with scaling: We first show the “generic” approach which works for all ops
- Key idea: Can never have unscaled FP8 tensor
  - So convert to FP16 first, then run op with FP16 input(s)

```
def quantized_dot_generic(x_f8, x_scale, z_scale):
  x_f16_unscaled = cast(x_f8, f16) * x_scale            # Step 1, 2
  z_f16_unscaled = dot(x_f16_unscaled, x_f16_unscaled)  # Step 3
  z_max = max(abs(z_f16_unscaled))                      # Step 4
  z_f8 = cast(z_f16_unscaled / z_scale, f8E4M3)         # Step 5, 6
  z_new_scale = zmax / 448
  return z_f8, new_scale
```

Handled by cuBLAS

Alternative way to represent scaling

- Mathematically equivalent to generic approach
- Only works on a few ops, notably Dot and Convolve
- Dot/Convolve has fp8 inputs and fp16 outputs – matches what hardware does

```
def quantized_dot_generic(x_f8, x_scale, z_scale):
  x_f16_unscaled = cast(x_f8, f16) * x_scale
  z_f16_unscaled = dot(x_f16_unscaled, x_f16_unscaled)
  z_max = max(abs(z_f16_unscaled))
  z_f8 = cast(z_f16_unscaled / z_scale, f8E4M3)
  z_new_scale = zmax / 448
  return z_f8, new_scale
```

```
def quantized_dot_alternative(x_f8, x_scale, z_scale):
  z_f16_input_scaled = dot(x_f8, x_f8, out_type=f16)
  z_f16_unscaled = z_f16_input_scaled * x_scale**2
  z_max = max(abs(z_f16_unscaled))
  z_f8 = cast(z_f16_unscaled / z_scale, f8E4M3)
  z_new_scale = zmax / 448
  return z_f8, new_scale
```

Only first two lines differ

OpenXLA

Q&A

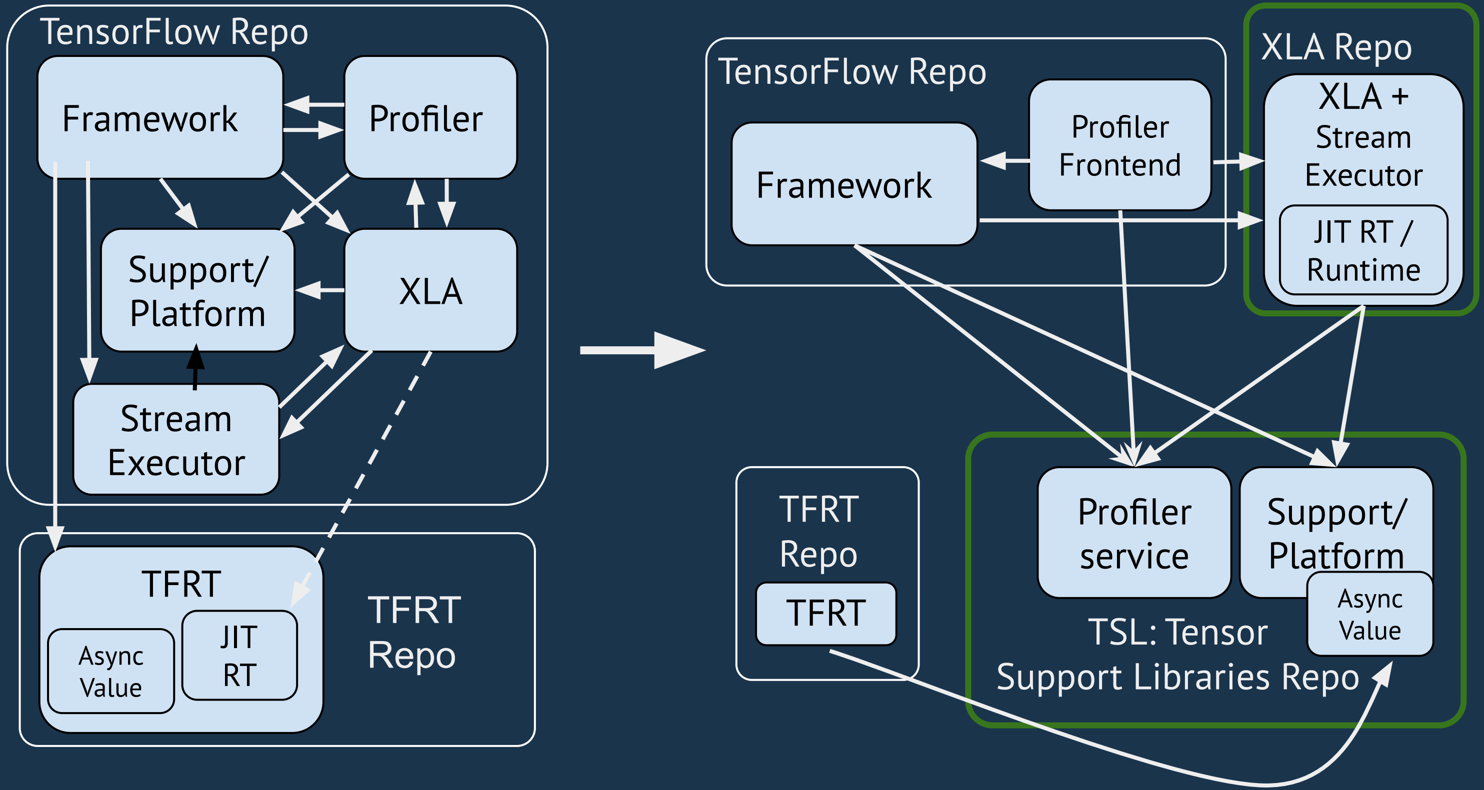
TensorFlow-XLA Refactor Update

Mehdi Amini, Google
aminim@google.com

OpenXLA

Recap: Extracting XLA from TensorFlow

Three repo solution

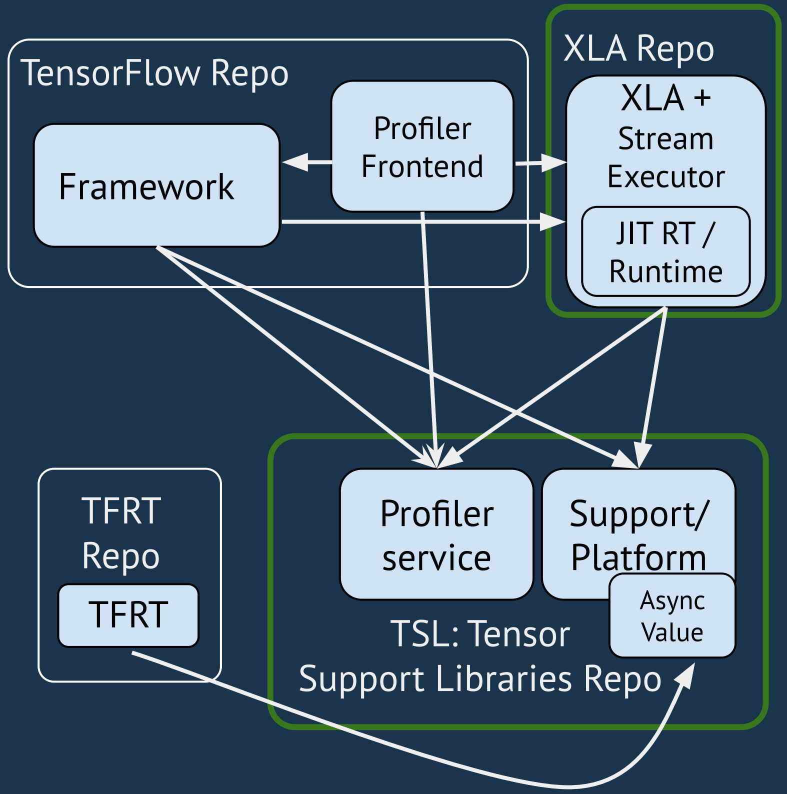


OpenXLA

Recap: Extracting XLA from TensorFlow

A three repo solution

- TSL: new repo that will contain TensorFlow platform-independent portability code (tensorflow/core/lib and /core/platforms) & profiler service.
- XLA repo integrates StreamExecutor and JitRT. XLA will depend only on TSL for platform-independent utilities & profiler APIs.
- TensorFlow will depend on XLA & TSL. We will vendor XLA & TSL code into tensorflow/third_party to avoid cross-repo synchronization. Code depending on TensorFlow (e.g. TF/XLA bridge) will stay in TensorFlow repo.
- Expected to deliver: ~~10/2022~~ → 12/2022



OpenXLA

Current Status

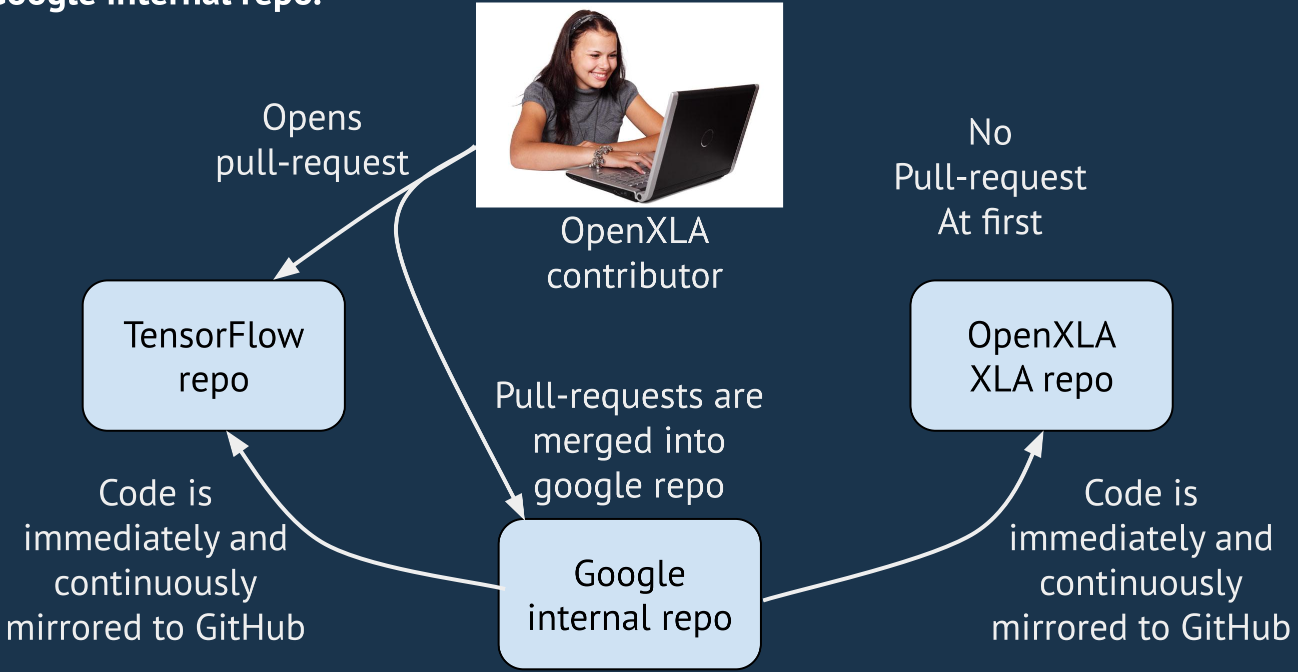
- https://github.com/google/tsl is 99% complete. Can already clone the repo and build / test most of it, See the CI script: tsl/.kokoro/linux/build.sh
- Towards being able to build/test tensorflow/compiler/xla without building anything else from TensorFlow.
- Refactoring the XLA directory structure: see tensorflow/compiler/xla/README.md

OpenXLA

Next Steps

December: [tensorflow/compiler/xla] published to [https://github.com/openxla/xla]

December-January: no changes to contribution workflow, XLA codebase exists in two places, perfectly synchronized through Google internal repo.

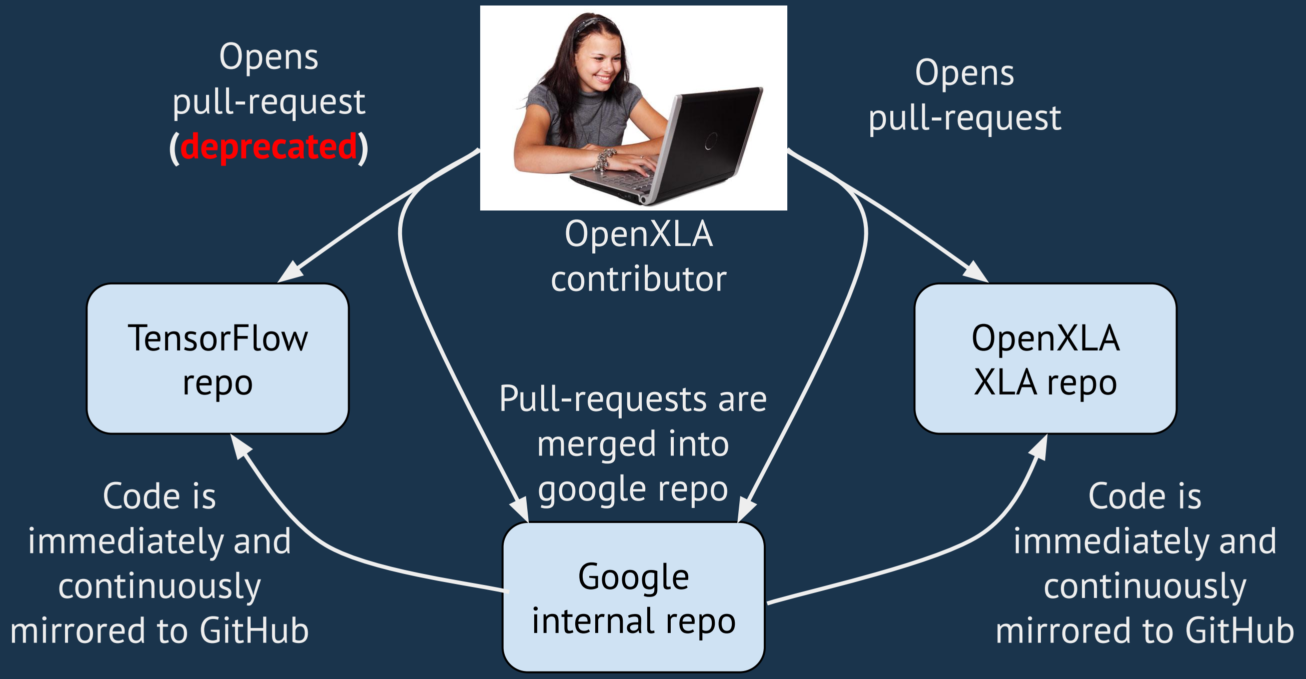


OpenXLA

Next Steps

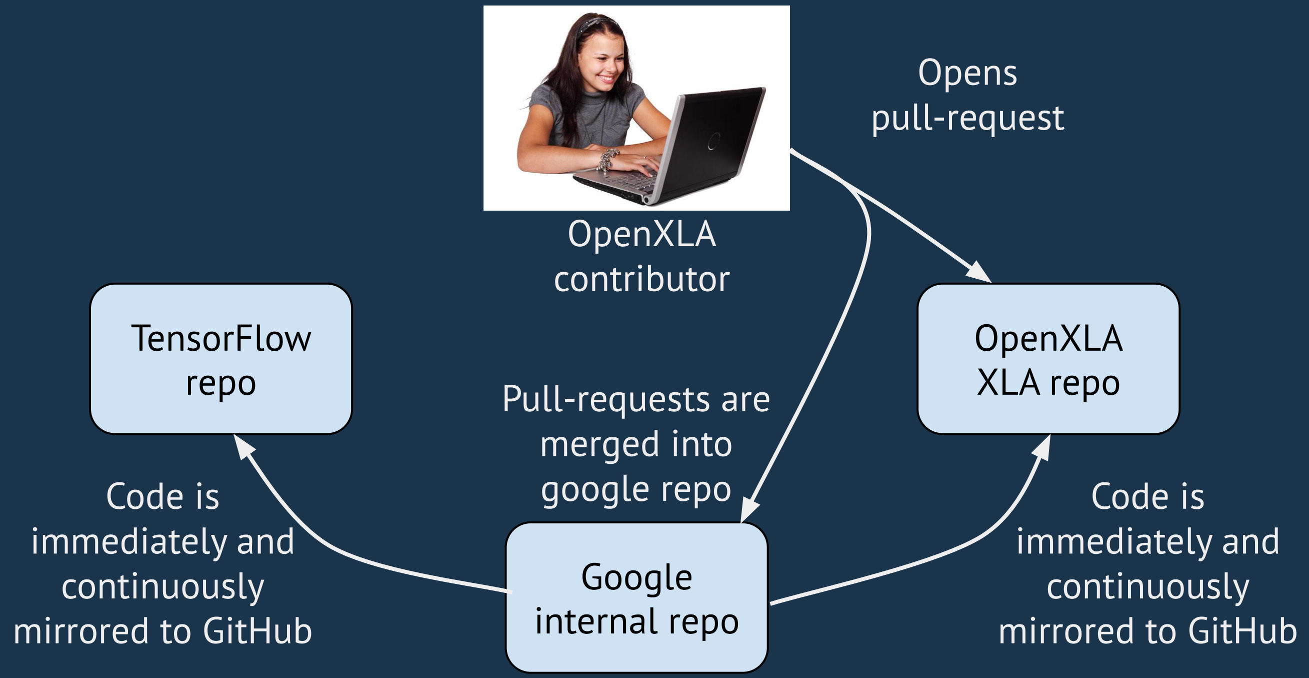
January-February: start accepting pull-requests in the openxla/xla repo.

- **Still possible to send PRs to XLA in the TensorFlow repo – same codebase existing in two places**



Next Steps

February: the openxla/xla repository will be the only way to submit new changes to XLA.



OpenXLA

Community Updates

Active RFCs

- StableHLO Compatibility / Versioning: [link]
- StableHLO Bounded Dynamism: [link]
- StableHLO Evolution: [link].
- FP8: [link]

December community meeting rescheduled to 12/13 at 8am PT!

Let's continue to discuss on GitHub!

github.com/openxla/xla/discussions