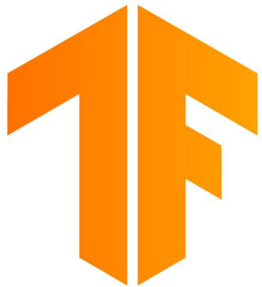
Google AI

TFRT Deep Dive

MLIR Open Design Meeting
March 19, 2020

Presenter: Eric Johnson, Jeremy Lau, Jing Dong, Mingsheng Hong
On behalf of TensorFlow runtime team and other TFRT contributors

Work in progress and subject to change



Talk outline

- TFRT overview (review of TensorFlow Dev Summit presentation)
- High level design: key concepts and subsystems
  - Common infra and graph execution
- Next steps and selected challenges

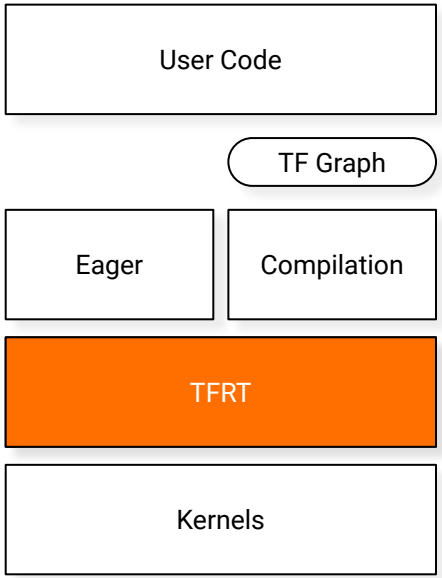
Talk Notes / Caveats:

- **Multi-host (distribution)** support are part of TFRT project scope (e.g. distributed training support), but not covered in this talk
- Lots of compiler-runtime co-design, but only covering **compiler design** in brief
- Does not cover the runtime clients
- WIP and subject to change

TFRT Overview

WHAT: Level-setting

Runtime is part of TF's infrastructure



- Interfaces with eager and graph compiler
- Drives model execution through kernel invocation
- TFRT will replace the existing TF runtime

WHY: Motivations for a new runtime

Observations of the ML/TF ecosystem...

- Faster and bigger models
- Research innovations: ops, kernels, and modeling
- ML everywhere

More performant eager and graph execution

More modular and extensible infrastructure

More flexibility in deployments, across server and mobile

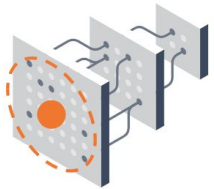
WHAT: TFRT vision

A runtime for the future of Machine Learning

A unified, extensible runtime providing best-in-class performance across a wide variety of domain specific hardware.

WHAT: Example deployments

TFRT will impact the most important use cases



Training

e.g. Improved error reporting

Serving

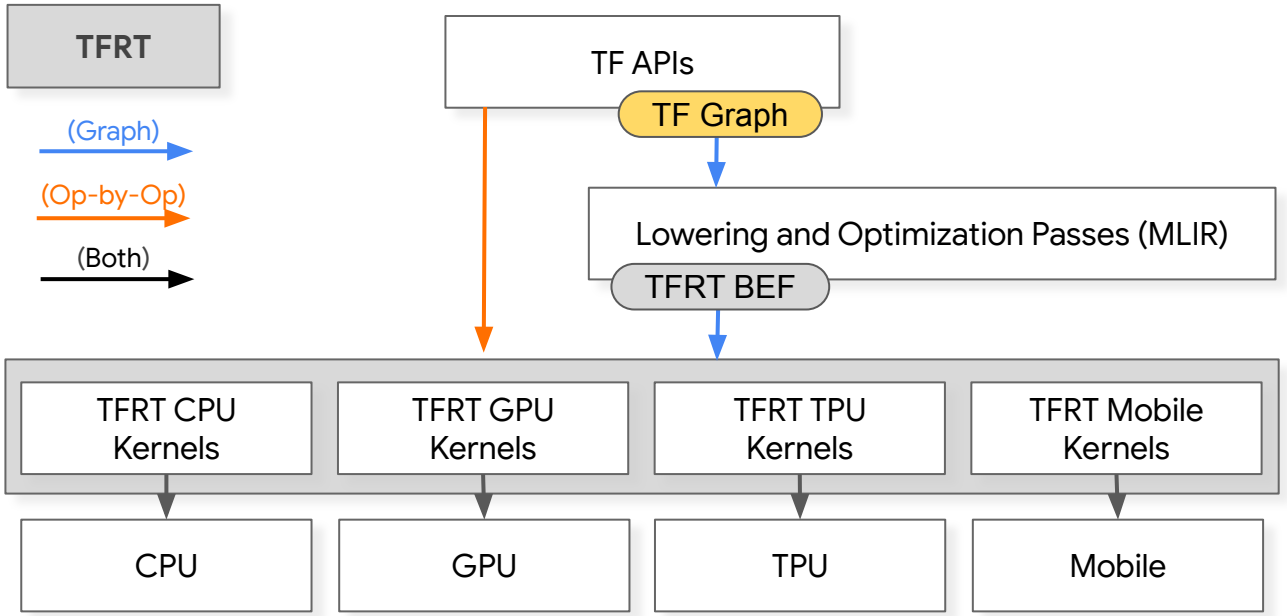
e.g. Best-in-class performance

Mobile

e.g. Unified training/inference infra

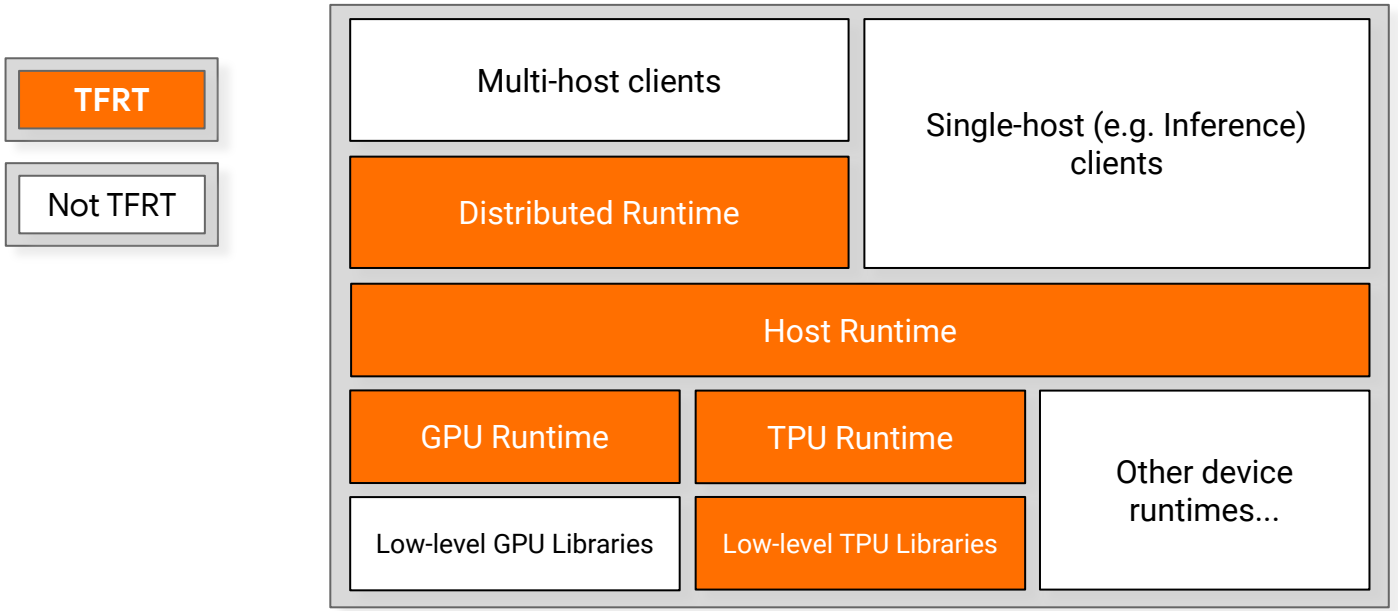
WHAT: Lifetime of an ML model

TFRT's critical role in the model training workflow



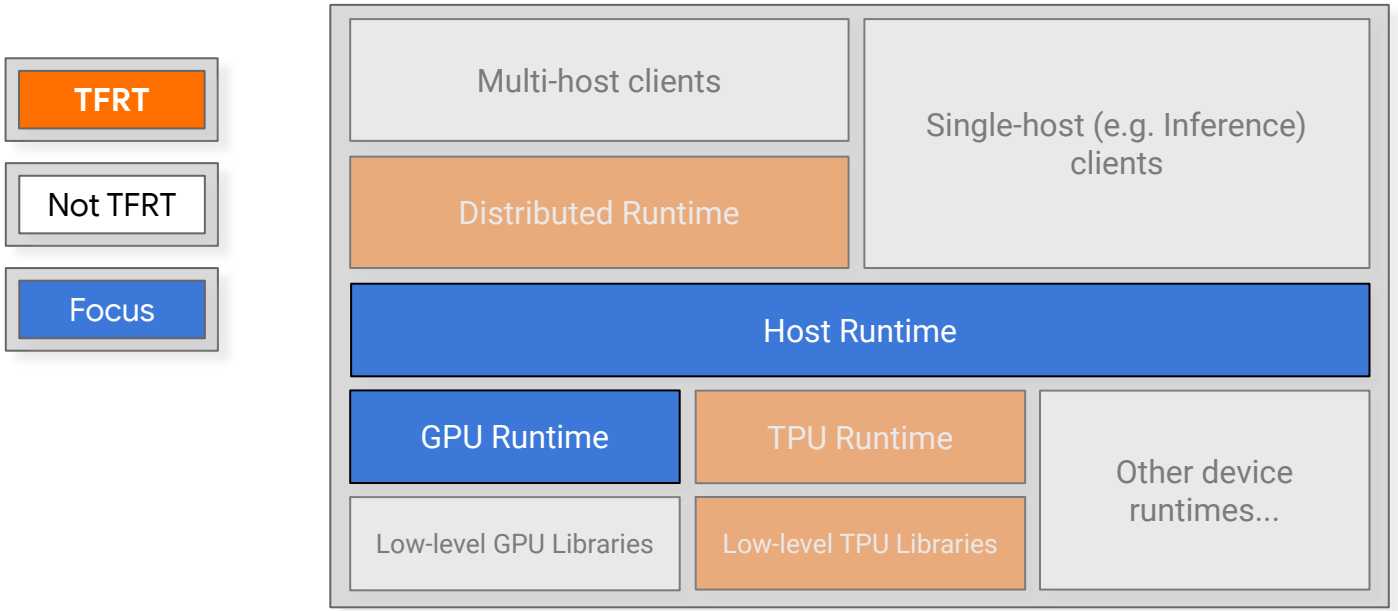
WHAT: TFRT Architecture

A high-level look at the key TFRT components



WHAT: TFRT Architecture

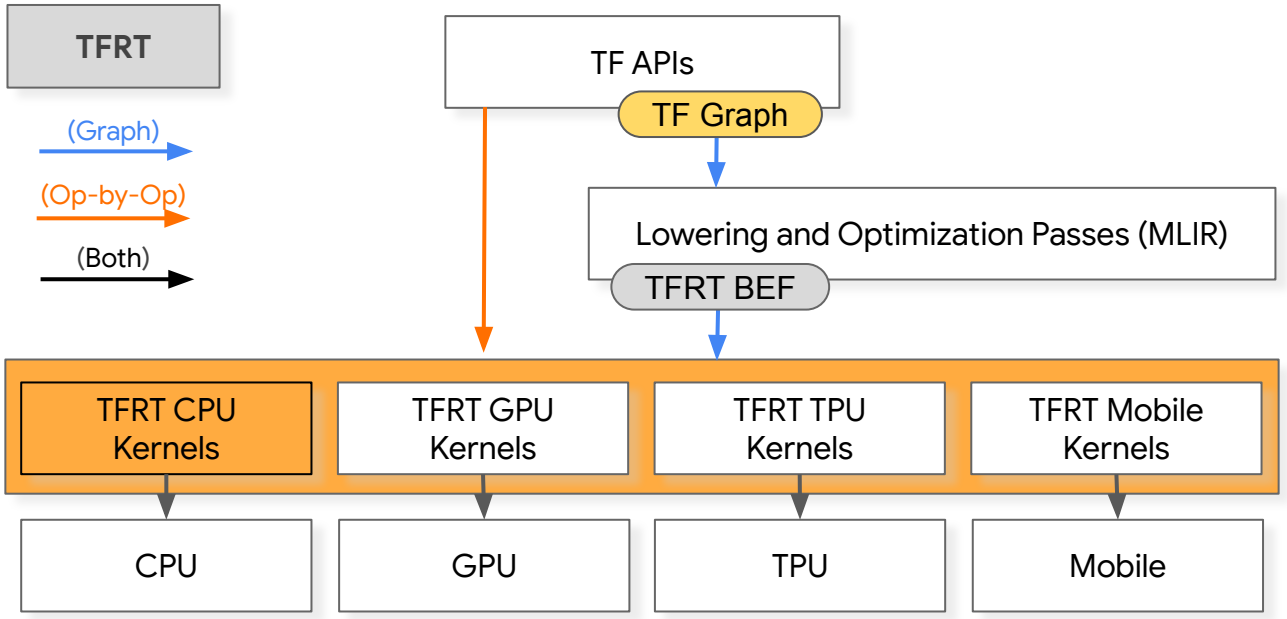
We will focus on Host Runtime and GPU Device Runtime



Google AI

TFRT Host Runtime

Where Host Runtime Fits



Host Runtime Design

- Foundation of TFRT: schedules work on the host and devices
- Clean separation between host and device runtimes:
  - Host runtime does not know anything about devices, just their runtimes (sets of kernels)
- Key design points:
  - Fully **asynchronous** - kernel executions can not block
  - Excellent **error propagation** in the presence of asynchrony
  - **Performance** as a first-class concern, for graph and eager
- Outline:
  - Common runtime infrastructure
  - Graph execution
  - Op-by-op execution (“eager”)

Key Abstraction: AsyncValue

- **Container** for data or resources
  - Not Tensor specific
- A "**future**" type, fulfilled with exactly one value, or an error
- Lock-free, low memory overhead, type erased, reference counted
- Helper class AsyncValueRef<T> provides type safety when contained type is known
- AsyncValues enable efficient asynchronous compute
  - Asynchronous functions return unavailable AsyncValues
  - Caller can schedule dependent computations with AsyncValue::AndThen()
  - Caller need not block until AsyncValue becomes available

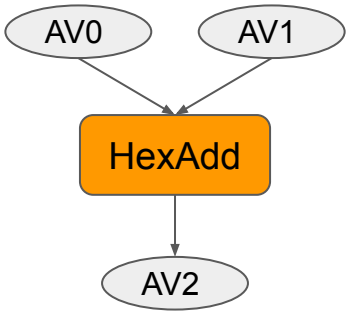
```
class AsyncValue {
 public:
  bool IsConcrete() const;
  bool IsError() const;
  // IsError || IsConcrete
  bool IsAvailable() const;

  template <typename T>
  const T& get() const;

  template <typename T, typename... Args>
  void emplace(Args&&... args);

  template <typename WaiterT>
  void AndThen(WaiterT&& waiter);

  // ...
}
```

Kernels

- Kernel: unit of computation scheduled by the runtime
  - Similar to kernel concept in current TensorFlow
- Kernels accept AsyncValue inputs and produce AsyncValue outputs
  - Runtime coordinates dataflow of AsyncValues between kernels
  - Outputs may not be immediately available, unlike current TensorFlow
- Runtime generally does not understand kernel semantics

```
// Low-level Kernel API.
void HexAdd(KernelFrame* frame) {
  // Fetch AsyncValues from KernelFrame.
  AsyncValue* arg0 = frame->GetArgAt(0);
  AsyncValue* arg1 = frame->GetArgAt(1);

  // Fetch integers from AsyncValues.
  int v0 = arg0->get<int>();
  int v1 = arg1->get<int>();

  // Construct AsyncValue for result and
  // point KernelFrame to it.
  frame->EmplaceResult<int>(v0 + v1);
}
```

```
// High-level Kernel API.
int HexAdd(int arg0, int arg1) {
  return arg0 + arg1;
}
```



Host Program

Host programs encode a dataflow graph:

- Similar to GraphDef in current TensorFlow
- Expressed in MLIR. Typically compiler generated
- Designed for low-level dispatch efficiency
- Designed for compiler transformations and analysis
  ⇒ Example: Use dataflow analysis for buffer reuse

```
func @sample_function() -> () {
  %one = hex.constant.i32 1          // Make AsyncValue with value 1
  %two = hex.constant.i32 2          // Make AsyncValue with value 2
  %three = hex.add.i32 %one, %two    // Add 1 and 2. Store 3 in AsyncValue %three

  hex.print.i32 %three               // Print AsyncValue %three
  hex.return %three : i32            // Return AsyncValue %three
}
```

Generic Type System

Kernel inputs and outputs are arbitrary C++ data types wrapped in AsyncValues
- Not always Tensors like current TensorFlow

In host programs a MLIR type maps to one C++ data type
- i32 → int32_t
- !ts.shape → tfrt::TensorShape
- !dht.tensor.f32.2 → tfrt::DenseHostTensor<float, 2>

Important for low-level efficiency, flexibility, debuggability:
- Type check host programs
- Device-specific kernels can directly use their native types
  - For example, CUDA kernels could accept and return CUstreams

```
// MLIR Program
%shape_b1 = ts.build_shape [100, 512]
%broadcast_b1 = dht.broadcast.f32.1 %b1, %shape_b1
%h1 = dht.add.f32.2 %matmul1, %broadcast_b1
%relu1 = dht.relu.f32.2 %h1
```

```
// MLIR Types
// () -> !ts.shape
// (!dht.tensor.f32.1, !ts.shape) -> !dht.tensor.f32.2
// (!dht.tensor.f32.2, !dht.tensor.f32.2) -> !dht.tensor.f32.2
// !dht.tensor.f32.2 ->  !dht.tensor.f32.2
```

Kernel Registration

- Kernels are registered at runtime initialization
  - Call `KernelRegistry::AddKernel` to map MLIR ops to C++ functions
  - Similar to MLIR PassRegistration
- Easy to experiment with new kernels:
  - Implement and register C++ function
  - Invoke kernel from MLIR, runtime runs the corresponding C++ code

- Moving towards a modular registration system
  - Dynamically load kernels
    - Deploy kernels as shared libraries
    - Less coupling between TFRT's code base and third party kernels
    - Requires stable kernel ABI
  - Can still statically link kernels if desired

```
// MLIR op hex.add.i32 is implemented by C++ function HexAdd.
static void RegisterKernels(KernelRegistry* registry) {
  registry->AddKernel("hex.add.i32", TFRT_KERNEL(HexAdd));
}
TFRT_STATIC_KERNEL_REGISTRATION(RegisterKernels);
```

TFRT Host Runtime Design Highlights

Asynchronous Execution

- Kernels may not block, nonblocking executor (see next section)
- Lock-free AsyncValue

Thread Pool

- One compute thread pool for the process: Centrally managed to avoid thread contention
- Maintain ~1 compute thread per core for efficiency
- Customizable for different target environments
  - Pool of std::threads for general server deployments
  - Single threaded implementation for resource constrained deployments

Memory Allocation

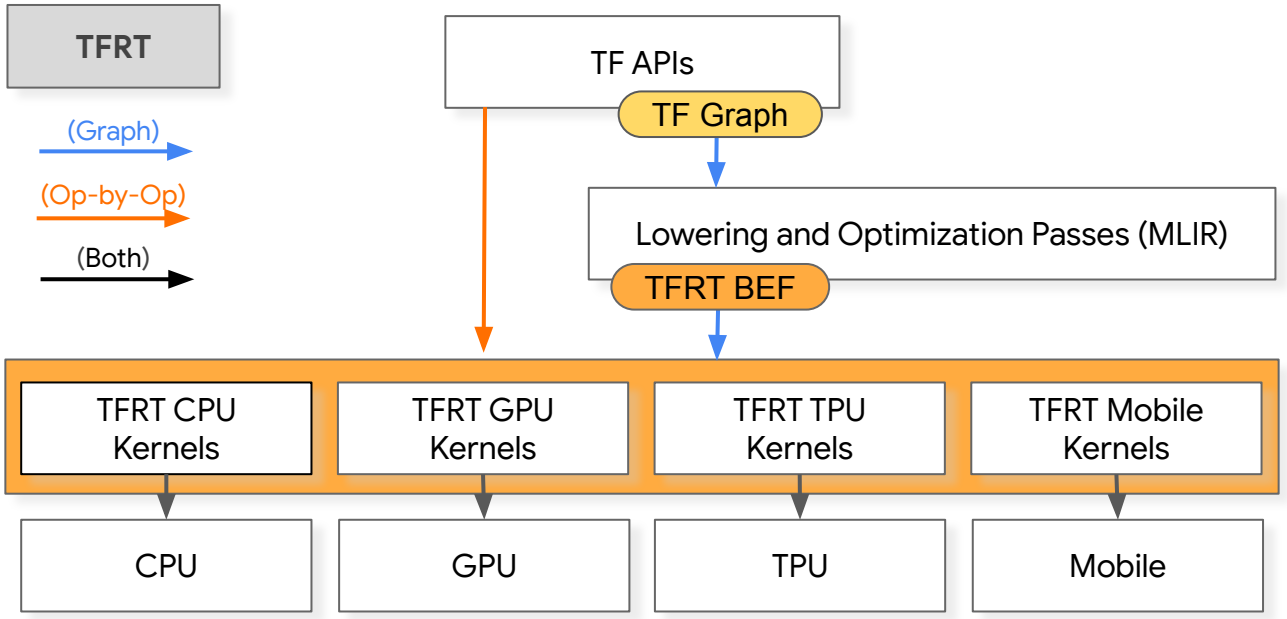
- Central interface (HostAllocator) for most memory allocations
- Customizable for different target environments
  - TCmalloc for general server deployments
  - Simple allocator for embedded environment

Kernels may use libraries with private threadpools or memory allocators but this may reduce efficiency

TFRT BEF Executor

"Binary Executor Format" (BEF) Files

Where BEF Executor Fits



Binary Execution Format (BEF)

- BEF encodes a hardware-specific *lowered* graph function
- Primary interface between compiler and runtime

- Designed for efficient execution
- **Low overhead**: execute program by reading mmap'd byte array
- **Persistent and stable**: Compile once offline, run many times online. Great for inference use-cases

- Composed of sections, similar to ELF. Each section has its own format
- Extensible: BEF is versioned, reader ignores unknown sections, new versions may define new sections

```
BEF_FILE     ::= `0x0B` `0xEF` SECTION*

SECTION_DATA ::= FORMAT_VERSION_SECTION
SECTION_DATA ::= LOCATION_FILENAMES_SECTION
SECTION_DATA ::= LOCATION_POSITIONS_SECTION
SECTION_DATA ::= STRINGS_SECTION
SECTION_DATA ::= CONSTANTS_SECTION
SECTION_DATA ::= KERNELS_SECTION
SECTION_DATA ::= TYPES_SECTION

SECTION_DATA ::= FUNCTIONS_SECTION
SECTION_DATA ::= FUNCTION_INDEX_SECTION

SECTION_DATA ::= CONSTANT_TYPES_SECTION
SECTION_DATA ::= CONSTANT_NAMES_SECTION
SECTION_DATA ::= REGISTER_TYPES_SECTION

// Unknown section.
SECTION_DATA ::= BYTE*
```

BEFExecutor

BEF Executor evaluates a BEF dataflow graph “executor” style:

- Not a bytecode-like interpreter: no concept of program counter
- “Strict” execution by default: run a kernel only when **all** its inputs are available

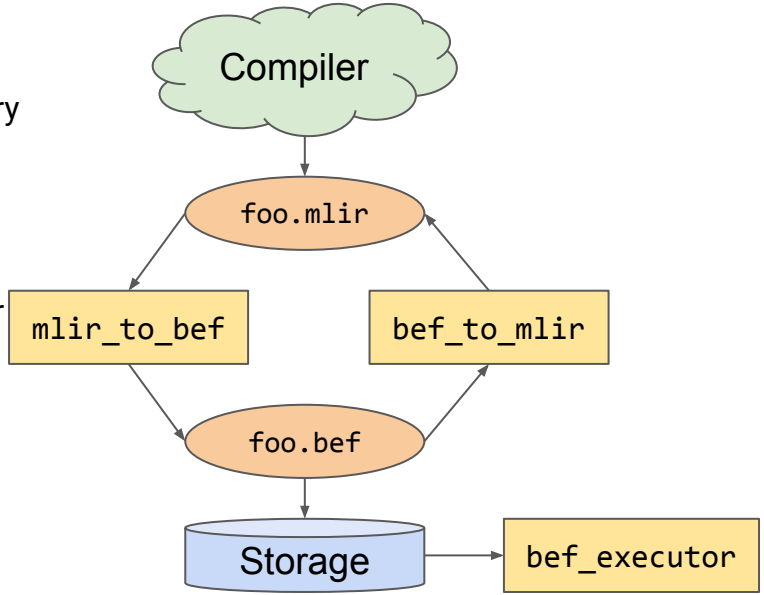
- Executor features:
  - Lock-free: atomics instead of mutexes
  - Non-blocking: defer dependent work with AsyncValue::AndThen
  - Supports “non-strict” execution: may run a kernel when **some** of its inputs are available
    - Good for efficiently forwarding unavailable inputs to outputs

- Key concepts:
  - BEF: dataflow graph
  - Kernel: dataflow node
  - AsyncValue: dataflow edge

Lowering to BEF

- `mlir_to_bef` walks a MLIR host program and emits a BEF binary
- Round trip: `mlir_to_bef` / `bef_to_mlir`
  - Very mechanical, similar to assembler / disassembler

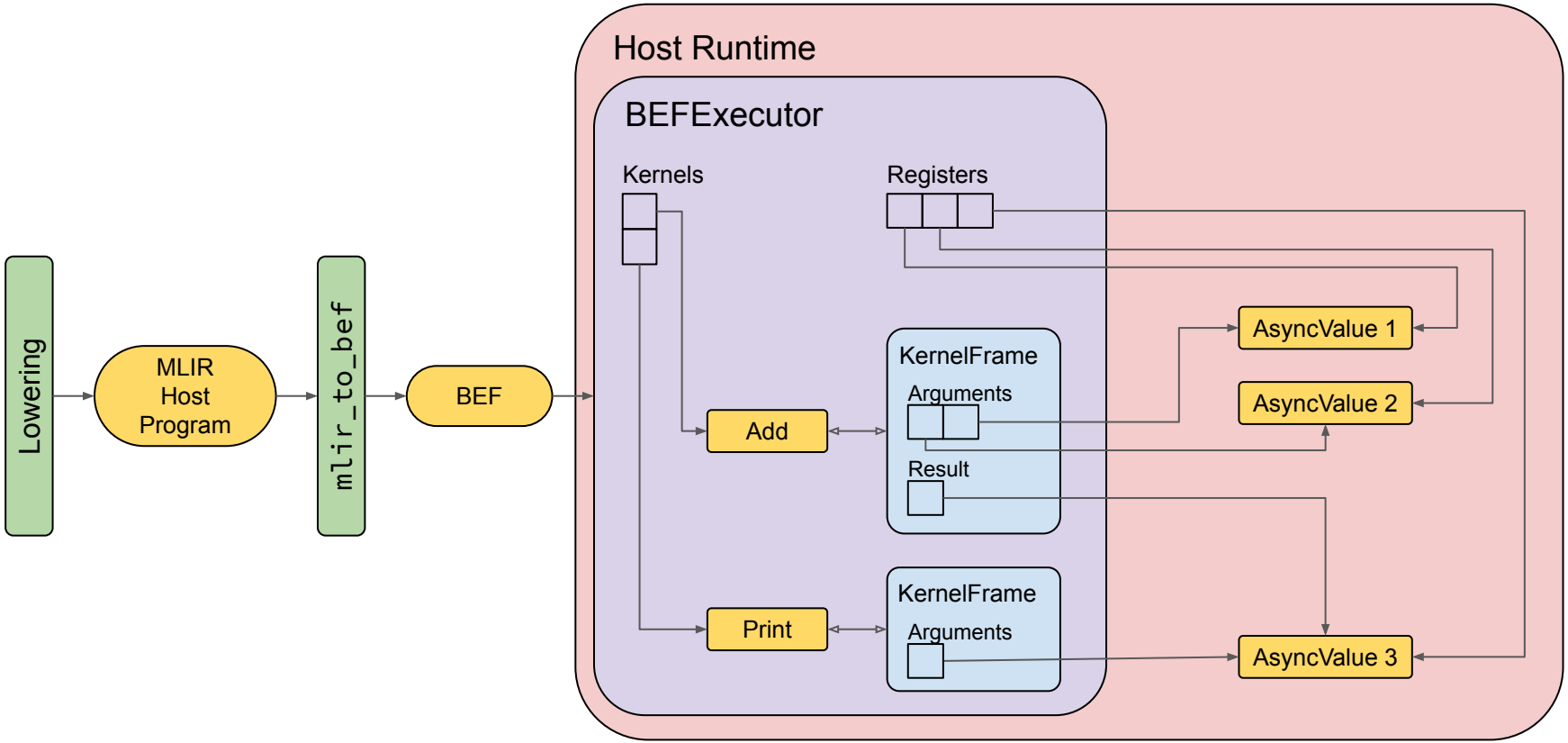
- Great for testing: Test kernels by running them with the executor
- Tests are input data, not encoded in binaries
- FileCheck verifies CHECKs in tests



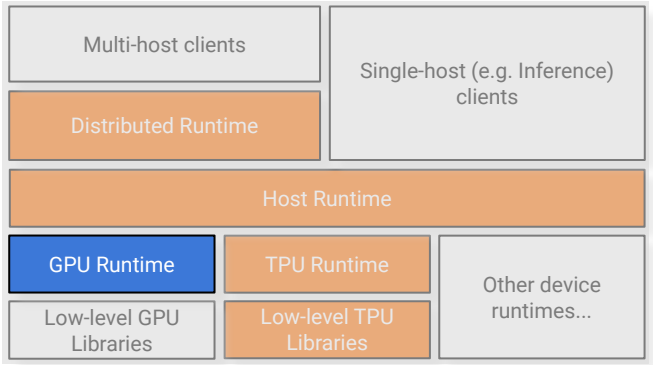

```
static int32_t HexAddI32(
    int32_t arg0,
    int32_t arg1) {
  return arg0 + arg1;
}

registry->AddKernel(
    "hex.add.i32",
    TFRT_KERNEL(HexAddI32));
```

```
func @test.add.i32() {
  %x = hex.constant.i32 42
  %one = hex.constant.i32 1
  %y = hex.add.i32 %x, %one

  // CHECK: int32 = 43
  hex.print.i32 %y

  hex.return
}
```

Host Runtime Summary



Device Runtime for Graph Execution



Device Runtime Design Principles

- A **thin** wrapper of low-level (driver) APIs, exposing device capabilities to graph compiler:
  - Memory Allocation
  - Async host <-> device transfer, and kernel execution
  - Dependency management
- Focus on mechanism instead of policy
  - E.g. No built-in special-purpose streams for GPU support:
    - For pure eager execution, can default to one stream for everything
    - For `tf.function` execution, compiler can pick streams

CUDA in TFRT: Kernel Execution

`crt.launch <stream> <launchable> <arguments>`

- `crt.launch` launches a `launchable` with `arguments` on `stream.`
- `launchable` can contain a few different types of programs:
  - In the simplest case, it can contain a cuda kernel symbol. Then, crt.launch will simply call `cudaLaunchKernel`
  - At the other extreme, it can build and execute a CUDA Graph
- It would be nice to make `launchable` serializable.

CUDA in TFRT: Kernel Execution

```
// Allocate a buffer and make a GPU tensor
%b1 = crt.mem.alloc %stream %size %align
%t2 = crt.make_tensor %b1 %shape

// Launch some kernels. %t1 has some data on GPU. %ch0 is a chain
%ch1 = crt.launch %stream <sigmoid> %t1 %t2 %ch0    // t2 = sigmoid(t1)
%ch2 = crt.launch %stream <sqrt> %t2 %t2 %c

// Allocate pinned host memory, copy, and print
%hb = crt.mem.host_alloc %size %align
%ch3 = crt.mem.copy_dtoh %stream %t2 %hb %ch2
%ch4 = crt.mem.free %t2 %ch3                  // can free immediately after launching
%ev = crt.event.create %flags
%ch5 = crt.event.record %stream %ev %c
%ch6 = crt.event.poll %ev %ch5                // %ch c                    ched
hex.print %hb2 %ch6
```

Use Chain to sequence device kernels

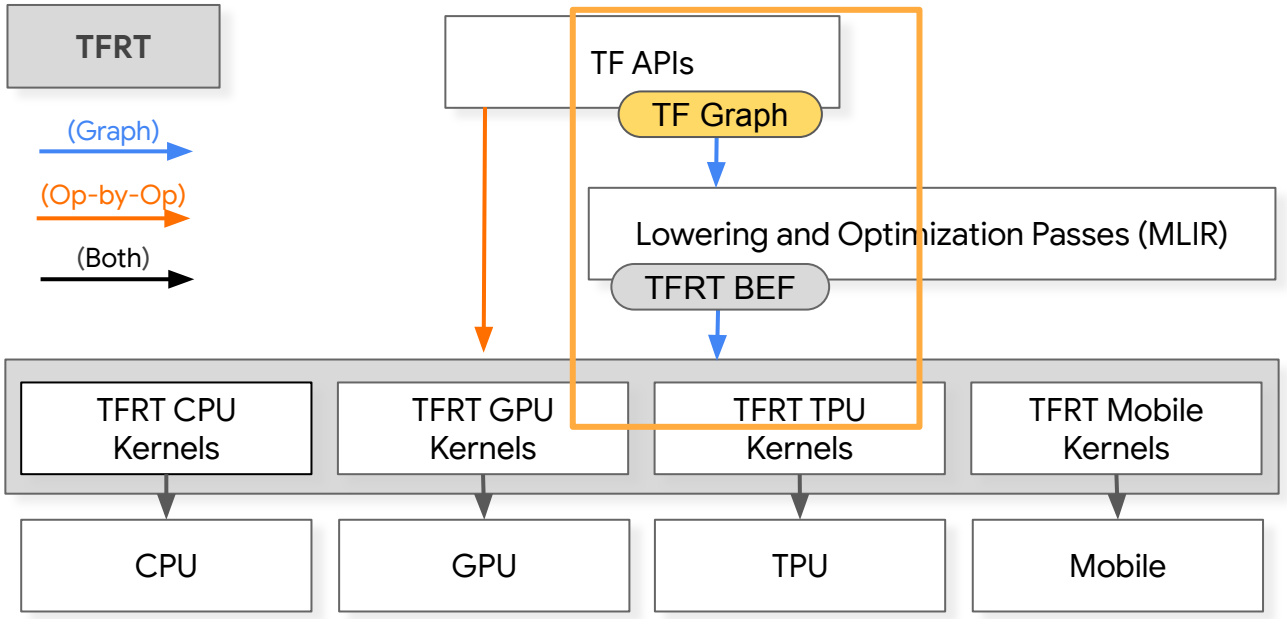
Consecutive synchronous kernels can be JITed/AOTed to reduce runtime overhead.

API closely resembles CUDA APIs

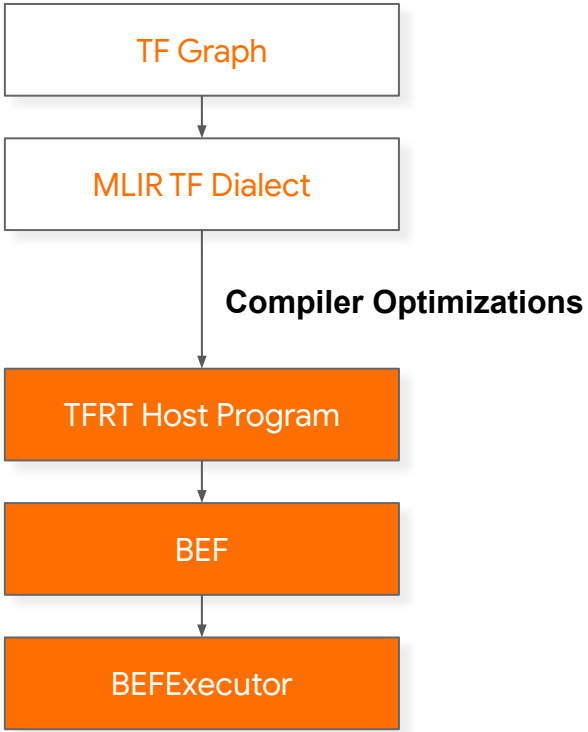
Google AI

TFRT End-to-End Inference Workflow

How the e2e Inference Workflow Fits



TFRT End-to-End Inference Workflow



Compiler Optimizations: Layout Optimization

Domain specific optimizations in MLIR passes

Channels First

```
%0 = "tf.Conv2d"(%input, %filter)
          {strides: [1,1,2,2], padding: "SAME", data_format: "NCHW"}
    : (tensor<1x64x56x56*xf32>, tensor<*xf32>) -> tensor<1x64x28x28xf32>

%1 = "tf.FusedBatchNorm"(%x, %scale, %offset) {data_format: "NCHW"}
    : (tensor<1x64x28x28xf32>, ...) -> tensor<1x64x28x28xf32>

%2 = "tf.Mean”(%1) {reduction_indices: [2,3]}
    : (tensor<1x64x28x28xf32>) -> tensor<1x64xf32>
```

Compiler Optimizations: Layout Optimization

Domain specific optimizations in MLIR passes

Channels Last

```
%0 = "tf.Conv2d"(%input, %filter)
          {strides: [1,1,2,2], padding: "SAME", data_forma: "NHWC"}
    : (tensor<1x56x56x64*xf32>, tensor<*xf32>) -> tensor<1x28x28x64xf32>

%1 = "tf.FusedBatchNorm"(%x, %scale, %offset) {data_format: "NHWC"}
    : (tensor<1x28x28x64xf32>, ...) -> tensor<1x28x28x64xf32>

%2 = "tf.Mean”(%1) {reduction_indices: [1,2]}
    : (tensor<1x28x28x64xf32>) -> tensor<1x64xf32>
```

Power of MLIR!

Dialects opt-in by implementing op interfaces

- Layout optimization works with MLIR Op interfaces and doesn't know anything about concrete operations (https://mlir.llvm.org/docs/Interfaces/)
- Optimizations are reusable across different dialects (e.g. the same layout optimization pass can be shared between TF and XLA dialects)
- Much much easier to write optimization passes with MLIR than with GraphDef

Benchmarking TFRT

ResNet GPU inference on TFRT vs. the existing stack

Setup

Model: Resnet-50 v1.5

Data precision: trained and served with FP16 mixed precision

Batch size: Inference using batch size 1 with image data (NHWC [1, 224, 224, 3]) loaded in memory

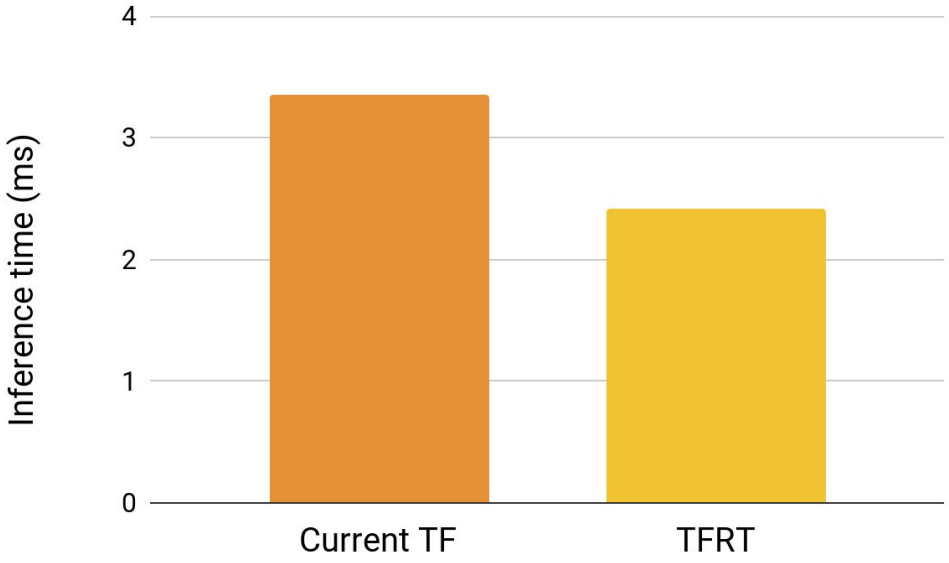
Scenario: MLPerf single-stream mode

Hardware: Mainstream CPU (Xeon Gold 6154) and GPU (NVIDIA TITAN V)

Toolchain: CUDA 10.1, CuDNN 7.6.4, GPU driver 430.34

Benchmarking results

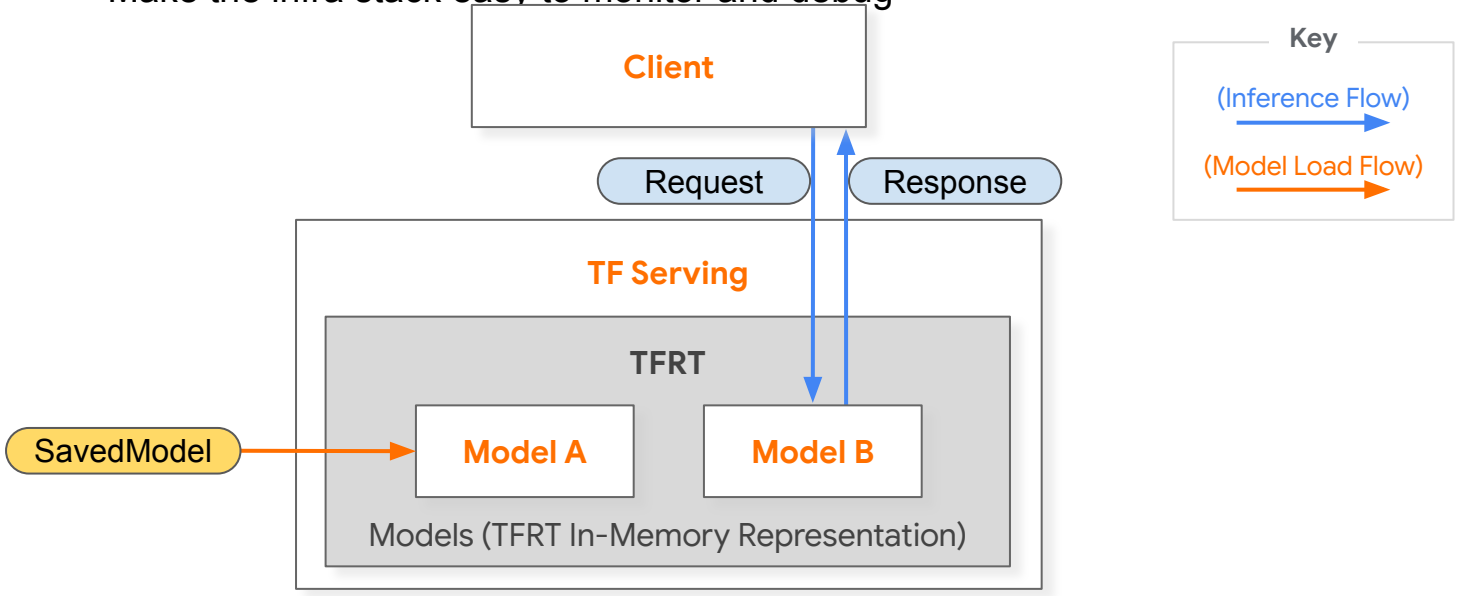
ResNet GPU inference is **28**% faster with TFRT



Next Steps and Selected Challenges

TFRT serving support

- Productionize TFRT integration with TensorFlow Serving and other Google production serving stacks
  - Build out graph compiler and runtime for serving
  - Provide a general purpose “runtime fallback” mechanism, to call into existing kernels via current runtime
  - Fine tune threadpool and tail latency
  - Make the infra stack easy to monitor and debug



TFRT training support

- TensorFlow integration
  - Graph execution
    - Graph compiler support (aligned with serving needs)
  - Eager execution
    - Parallel efforts in reducing python stack overhead
- Exploring the integration with other ML frontends

TFRT mobile support

- An opportunity to provide a unified mobile & server experience
- Binary size and library dependency management
- WIP: Bridging performance and feature gaps with TFLite
- Selected opportunities and challenges
  - On-device compiler with small binary size
  - AOC and interpreter modes
  - Running op scheduling that balances performance and power concerns
- Stay tuned!

Google AI

Thank you! Questions?