IREE: standard-/compilation-based ML stack via Vulkan/SPIR-V

Lei Zhang, on behalf of the IREE team

Khronos ML Webinar, 2022-05-05

Background:
Challenges and Beliefs

Existing ML stack challenges

- ML stacks face huge problem space with combinatorial complexity due to
  - Evolving ML model architectures; various frameworks with shifting user interests
  - Growing heterogeneous hardware (CPUs/GPUs w/ vector/matrix, AI accelerators)
  - Different deployment scenarios (server, desktop/laptop, mobile/edge, web, etc.)
- ML stacks' in-house hardware "interfaces" at ML graph/op level leads to
  - Hardware needs to build full API/runtime/kernel/compiler story to integrate
  - Stack needs to have full story for all hardware and deployment scenarios
- So we see fragmented solution space with
  - Solutions specialize towards a subset and often lack adaptability and generality
  - Extensive duplicated manual engineering efforts within/across various stacks

Towards generalizable and performant ML stack

- We have seen similar challenges in graphics
  - Varying rendering techniques, game engines, GPU vendors, machine form factors
- ML inference stack can draw experiences from decades of learnings in graphics
  - Standards to support various hardware for both commonality and optionality
  - Compilers to handle different architectures for reusability and performance
- Vulkan and SPIR-V presents a modern clean base solution
  - Explicitly exposing hardware functionalities; not opinionated with high-level constructs
  - Low-level; suitable for auto generation (both host scheduling and device executable)
  - Readily available on many platforms; meeting various deployment needs

IREE Architecture

IREE

> A MLIR-based end-to-end compiler and runtime that lowers ML models to a unified IR that scales up to datacenter and down to mobile and edge deployments

Intermediate Representation Execution Environment

IREE key characteristics

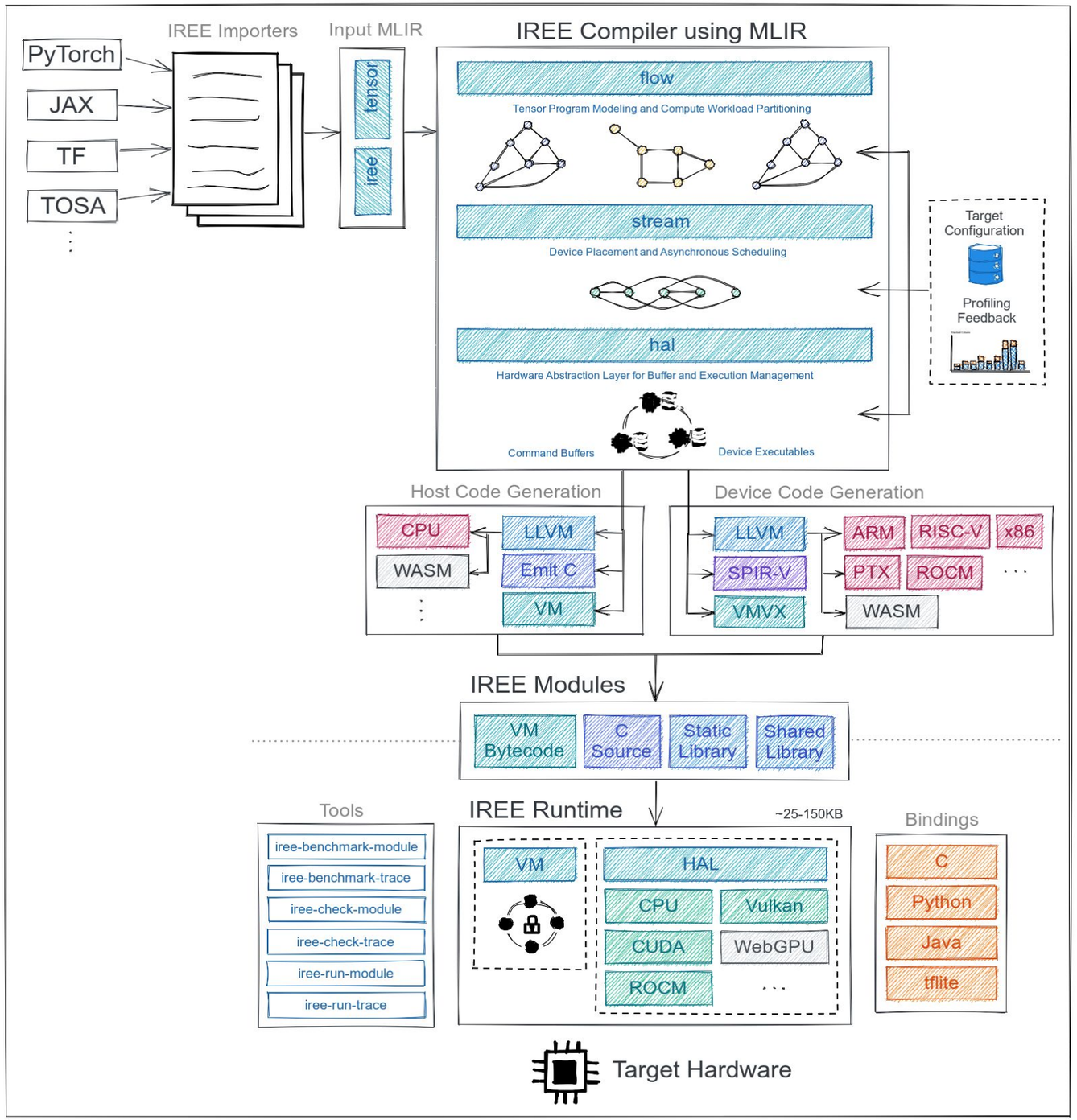
- Standard- and community-based
  - Adopting Vulkan, SPIR-V, WebGPU, etc. and working with OSS community
- Compilation-based
  - Using compilers to bridge the level semantics gap and generate optimal task/job schedule (i.e., automated task system middleware for ML)
- Holistic
  - One unified IR to represent both dispatchable executables and scheduling logic to enable whole-program optimization
- Scalable
  - Cooperating with other accelerator users, aware of resource constraints, friendly to diversified usage and deployment scenarios

IREE overall architecture

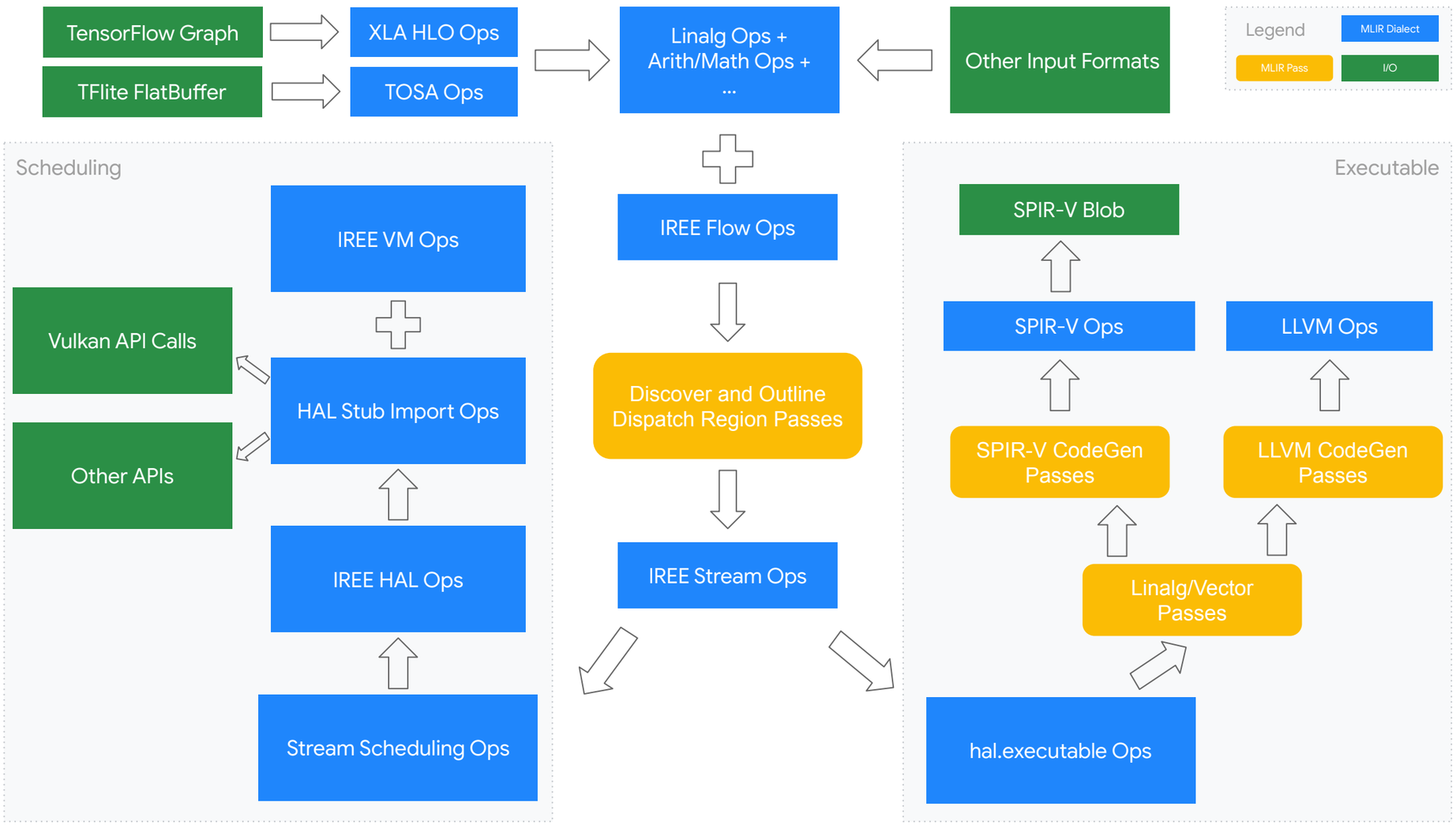
This is the end-to-end flow envisioned by IREE.

IREE does not provide them all: it relies on many components from the ecosystem.

The system is strongly layered and many components are optional: various offline compilers and almost zero-cost configurable runtime.



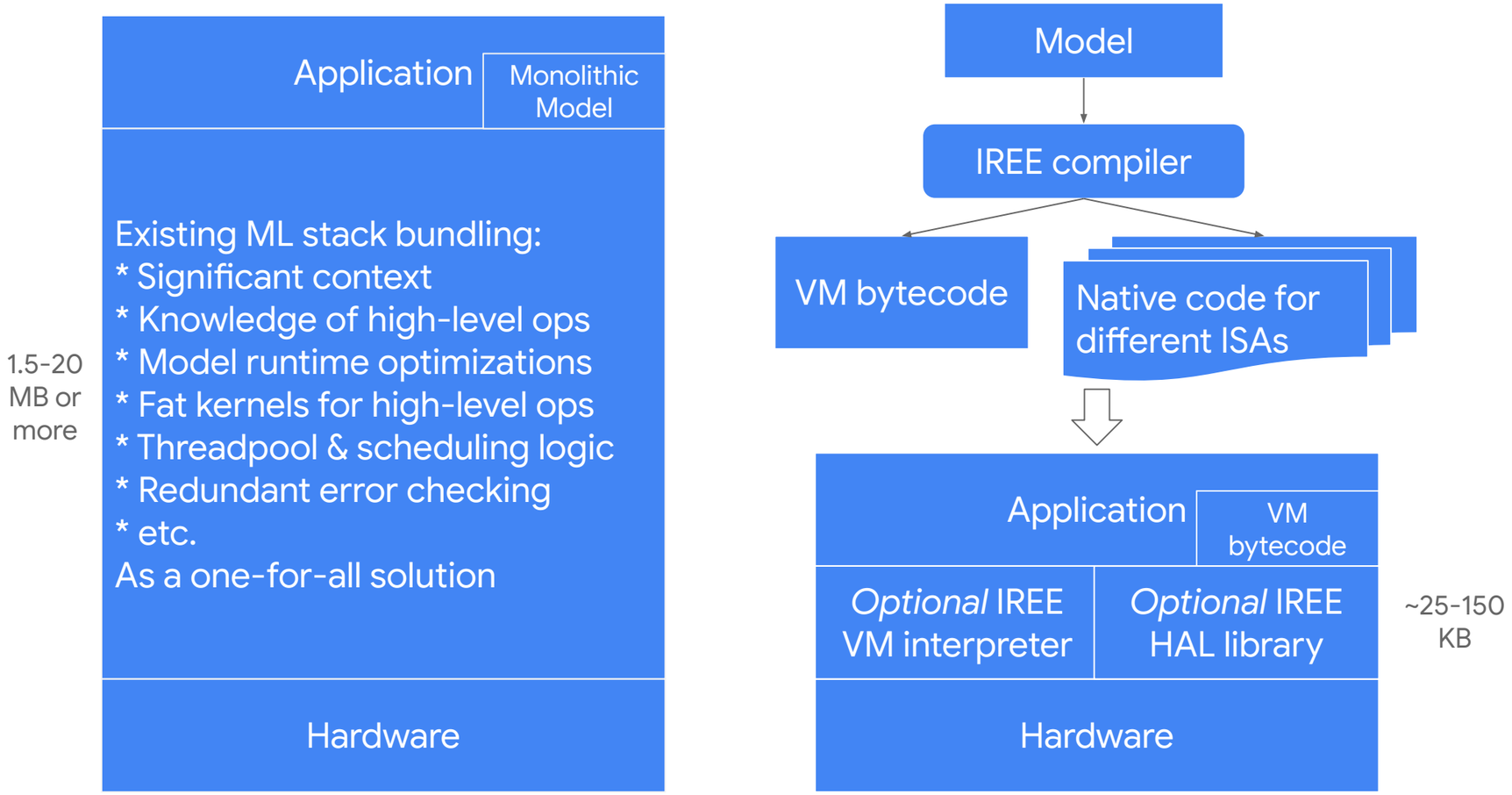
IREE core compilation flow



IREE runtime

IREE does not have a traditional “fat” runtime that bundles everything.

IREE provides an almost zero-cost virtual machine for interpreting host scheduling ops compiled from ML models. It just performs lightweight math for workload size calculation and performs task scheduling.



HAL: Vulkan-inspired hardware abstraction layer

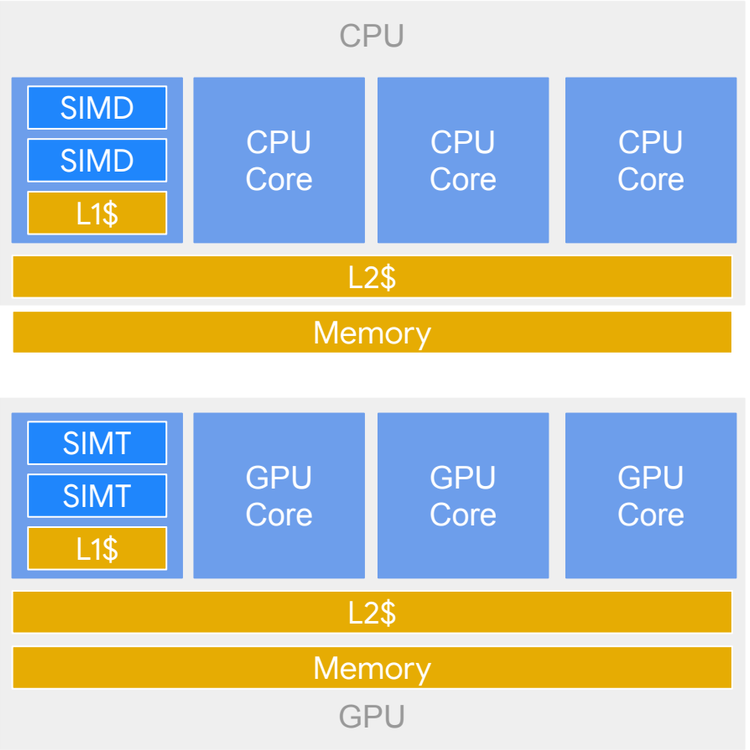
- Common abstraction for CPU, GPU, and beyond
  - All with multi-level memory/compute hierarchies
  - All meant for compute in tiled fashion
- Building pipelines to exploit the scheduling hierarchy
  - Submissions (workload + coarse-grained sync)
    - Command buffers (workload + fine-grained sync)

On host

      - Dispatches (⟶ GPU; ⟶ CPU)
        - Workgroups (⟶ GPU cores; ⟶ CPU threads)
          - Subgroups (⟶ GPU SIMT; ⟶ CPU: SIMD)

In executable
(On hardware; target CodeGen)

            - Instr. (⟶ GPU thread; ⟶ CPU: lane)



CPU & GPU compute and memory
(abstract illustration)

HAL IR example

HAL has scheduling ops that map to new generation explicit GPU APIs like Vulkan.

These ops effectively expose Vulkan C APIs as compiler IRs for automatic transformation, to enable codify best practices via compilers.

This is where we materialize concrete (binding-based) ABIs between executables and scheduling.

```
module {
 hal.executable @executable_module {
  hal.interface @abi {
   hal.interface.binding @ret, set=0, binding=0, type="StorageBuffer", access="Read"
   ...
  }
  hal.executable.binary {data = dense<[...]> : vector<1620xi8>, format = "SPIR-V"} ...
```

Executable: SPIR-V

```
func @main(%arg0: !iree.ref<!hal.buffer>, %arg1: !iree.ref<!hal.buffer>) -> !iree.ref<!hal.buffer> {
 %dev = hal.ex.shared_device : !iree.ref<!hal.device>
 %allocator = hal.device.allocator %dev : !iree.ref<!hal.allocator>
 %buffer = hal.allocator.allocate %allocator, ..., shape=[...], element_size=4 : !iree.ref<!hal.buffer>
 %cmd = hal.command_buffer.create %dev, "OneShot", "Transfer|Dispatch" : !iree.ref<!hal.command_buffer>
 hal.command_buffer.begin %cmd
 hal.ex.push_descriptor_set %cmd, ...
 hal.device.switch(%dev: !hal.device)
 #hal.device.match.id<"vulkan*"> : !iree.ref<!hal.buffer> {
    ...
    %exe = hal.executable.look_up, ...
    hal.command_buffer.dispatch %cmd, %exe, entry_point=0, workgroup_xyz=[%c1, %c5, %c1]
 }
 %memory_barrier = hal.make_memory_barrier "DispatchWrite", "DispatchRead" : tuple<i32, i32>
 hal.command_buffer.execution_barrier %cmd, "CommandRetire", "CommandIssue",
                                      memory_barriers=[%memory_barrier]
 ...
 hal.command_buffer.end %cmd
 hal.ex.submit_and_wait %dev, %cmd
 return %buffer : !iree.ref<!hal.buffer>
```

Scheduling

Vulkan Current Status and Roadmap

General approaches

- Investing in basics to establish solid foundation for generalization and performance
  - Prioritizing tasks benefiting a broad range of models and/or architectures
  - Aiming to provide reasonably good default solution towards all cases
  - Leaving the door open for power users to hyper tune specific cases
- Built out SPIR-V CodeGen in MLIR and Vulkan runtime in IREE
  - Can compile and execute many vision and language models on various hardware
  - Targeting Vulkan compute shaders and core Vulkan compute API subset
- Focusing widely applicable compilation optimizations thus far
  - No manual/automated tuning; using one set of heuristics and default parameters

Transformer models across various GPUs

	FP32 Model	GPU / FLOps	IREE Latency	Comparison Latency
Mobile	MobileBERT	ARM Mali G78 (Pixel 6) / 2T	120ms	TFLite OpenCL 123ms
Laptop	miniLM	Apple M1 Max / 10.4T	11.6ms	TF-Metal 16.99ms
Desktop	miniLM	AMD RX 5700XT / 9.7T	8ms	
Server	miniLM	NVIDIA Tesla V100 / 15.7T	6.3ms	

Models on mobile GPU

	FP32 Model	IREE Latency	TFLite Latency (Buffer)	TFLite Latency (Texture)
Pixel 6	MobileBERT	120ms	172ms	123ms
Pixel 6	MobileNetV2	9ms	8ms	6ms
Pixel 6	DeepLabV3	12ms	12.1ms	9.2ms
Pixel 6	PoseNet	15ms	14.4ms	8ms

Roadmap and tasks

- General functionality features, e.g.,
  - Smaller bitwidths (fp16, int8, etc.)
  - Reducing initialization overhead
- General optimizations, e.g.,
  - Better fusion, better buffer layout, supporting texture
  - Search, autotuning
- More platforms, e.g.,
  - SPIR-V CodeGen + WebGPU HAL ⟶ Web platform
  - SPIR-V CodeGen + Metal HAL ⟶ Apple platform

References

- Codebase and documentation
  - https://github.com/google/iree
  - https://google.github.io/iree/

- Mailing list
  - iree-discuss@googlegroups.com
  - https://groups.google.com/forum/#!forum/iree-discuss

- Chat room
  - https://discord.gg/26P4xW4