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
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.
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

TensorFlow Graph
TFLite FlatBuffer

XLA HLO Ops
TOSA Ops

Linalg Ops +
Arith/Math Ops +
...

Discover and Outline
Dispatch Region Passes
IREE Flow Ops

Other Input Formats

Legend

MLIR Dialect
MLIR Pass
I/O

MLIR CodeGen
Passes
LLVM CodeGen
Passes

EXEC
SPIR-V CodeGen
Passes

SPIR-V Ops
LLVM Ops

SPIR-V Blob

hal.executable Ops

IREE HAL Ops
IREE VM Ops
HAL Stub Import Ops

IREE Stream Ops

Stream Scheduling Ops
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.

---

<table>
<thead>
<tr>
<th>Application</th>
<th>Monolithic Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existing ML stack bundling:</td>
<td></td>
</tr>
<tr>
<td>* Significant context</td>
<td></td>
</tr>
<tr>
<td>* Knowledge of high-level ops</td>
<td></td>
</tr>
<tr>
<td>* Model runtime optimizations</td>
<td></td>
</tr>
<tr>
<td>* Fat kernels for high-level ops</td>
<td></td>
</tr>
<tr>
<td>* Threadpool &amp; scheduling logic</td>
<td></td>
</tr>
<tr>
<td>* Redundant error checking</td>
<td></td>
</tr>
<tr>
<td>* etc.</td>
<td></td>
</tr>
</tbody>
</table>

As a one-for-all solution

1.5-20 MB or more

---

<table>
<thead>
<tr>
<th>Application</th>
<th>VM bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional IREE VM interpreter</td>
<td></td>
</tr>
</tbody>
</table>

| Application | Optional IREE HAL library |

Hardware

~25-150 KB
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)
      - Dispatches (→ GPU; → CPU)
      - Workgroups (→ GPU cores; → CPU threads)
        - Subgroups (→ GPU SIMT; → CPU: SIMD)
        - Instr. (→ GPU thread; → CPU: lane)
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.
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

<table>
<thead>
<tr>
<th>FP32 Model</th>
<th>GPU / FLOps</th>
<th>IREE Latency</th>
<th>Comparison Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mobile</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MobileBERT</td>
<td>ARM Mali G78 (Pixel 6) / 2T</td>
<td>120ms</td>
<td>TFLite OpenCL 123ms</td>
</tr>
<tr>
<td><strong>Laptop</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miniLM</td>
<td>Apple M1 Max / 10.4T</td>
<td>11.6ms</td>
<td>TF-Metal 16.99ms</td>
</tr>
<tr>
<td><strong>Desktop</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miniLM</td>
<td>AMD RX 5700XT / 9.7T</td>
<td>8ms</td>
<td></td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miniLM</td>
<td>NVIDIA Tesla V100 / 15.7T</td>
<td>6.3ms</td>
<td></td>
</tr>
</tbody>
</table>
## Models on mobile GPU

<table>
<thead>
<tr>
<th>Pixel 6</th>
<th>FP32 Model</th>
<th>IREE Latency</th>
<th>TFLite Latency (Buffer)</th>
<th>TFLite Latency (Texture)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileBERT</td>
<td></td>
<td>120ms</td>
<td>172ms</td>
<td>123ms</td>
</tr>
<tr>
<td>MobileNetV2</td>
<td></td>
<td>9ms</td>
<td>8ms</td>
<td>6ms</td>
</tr>
<tr>
<td>DeepLabV3</td>
<td></td>
<td>12ms</td>
<td>12.1ms</td>
<td>9.2ms</td>
</tr>
<tr>
<td>PoseNet</td>
<td></td>
<td>15ms</td>
<td>14.4ms</td>
<td>8ms</td>
</tr>
</tbody>
</table>
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