MLIR Lighthouse Project

Contributors: Renato Golin, Rolf Morel, Tuomas Karna, Adam Siemieniuk, Frank Schlimbach, Petr Kurapov, Dmitry Chigarev, Julian Oppermann, Charitha Saumya, Arun Thangamani, MD Asghar Ahmad Shahid, etc.

Special thanks to: Andrzej Warzynski, Oleksandr Zinenko, Nicolas Vasilache, Jacques Pienaar, Matthias Springer, Javed Absar, Jian Hui Li, Alexander Heinecke, and everyone else that interacted on the forum threads and PRs.

Agenda

- How did we get here?
- What are we doing?
- Where do we want to go?
- Summary

https://github.com/llvm/lighthouse

How did we get here?

... between end of 2024 and end of 2025 ...

Recurring problems in MLIR

- Upstream dialect definitions were lacking substance
  - Operations and types have weak semantics (various interpretations)
  - Verification is often minimum, canonicalization opinionated, *best-effort*
  - Some dialects are *left-over* from a previous time (`quant`, `ml_program`)
  - Dialect documentation is *severely* lacking (but improving)

- Upstream rewrites are disconnected from each other
  - There's no upstream definition of IR normal forms, invariants
  - Canonicalization's *best effort* had some disagreements on what **it is**
  - Most (all?) users are downstream with their own expectations

Path to a solution

- Those problems led to fragmentation…
  - We discussed restructuring MLIR, which led to a survey and a charter
  - Separation between dialect groups, maintainers, focus groups
- And the fact that MLIR is *not* a compiler…
  - Common misconception, it's a framework with which to *build* compilers
  - But these compilers don't always agree on *how…*
- We wanted a project that *uses* MLIR in an *upstream-agreed* way
  - Define invariants, fixed-points, forms and common transforms
  - Create **end-to-end pipelines** to validate those assumptions
  - Provide evidence to **improve MLIR** semantics and usability

What are we doing?

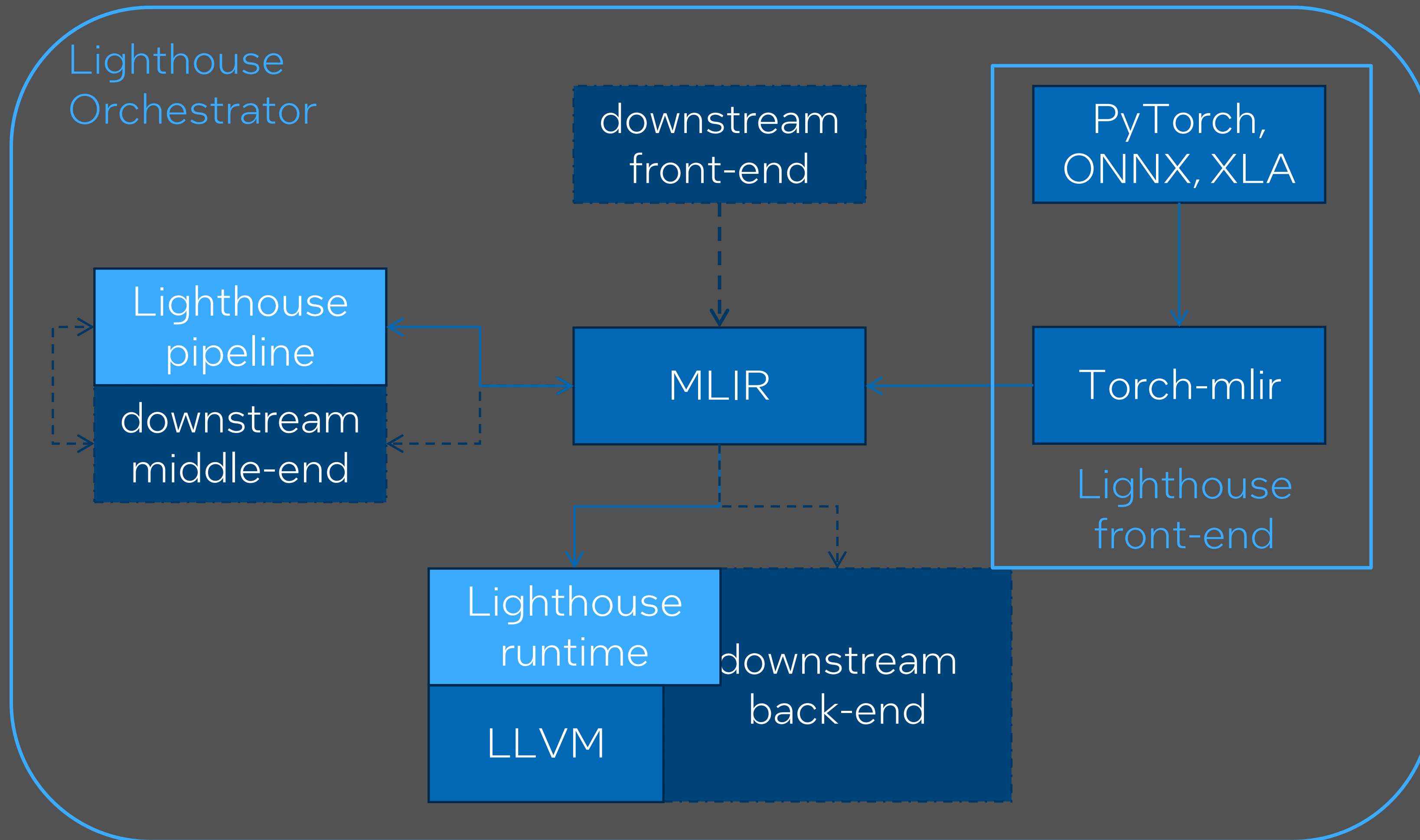
… since end 2025 when we started working on it …

Discovering Pathways (finished)

- Multiple examples to find common patterns using the MLIR API
  - And end-to-end XeGPU reaching high % of peak performance
  - A CPU matmul example, using some shared and x86 specific schedules
  - Two ingress examples, from PyTorch and a local MLIR generator
  - A distributed MPI Feed Forward layer running on CPUs

- Gave us enough information to design a common framework
  - Ingress (PyTorch, OpenXLA, ONNX, generators) into MLIR Modules
  - Pipeline stage composition independent of passes or transforms
  - Execution needs memory & device driver & I/O managers
  - Native distribution is *complicated*

Initial Design: Lighthouse Modules



Importing from PyTorch

Wrappers around torch-mlir

```python
from lighthouse.ingress.torch import import_from_model

# Import a Torch model object into MLIR
from MLPModel.model import MLPModel
model = MLPModel()
sample_input = torch.randn(1, 10)
mlir_module_ir: ir.Module = import_from_model(
    model,
    sample_args=(sample_input)
    dialect="linalg-on-tensors",
    ir_context=ir.Context()
)
print(f"{mlir_module_ir}")
```

```python
from lighthouse.ingress.torch import import_from_file

# File here is a Python file with a Torch nn.Module
mlir_module_ir: ir.Module = import_from_file(
    model_path=“MLPModel/model.py",
    model_class_name="MLPModel",
    init_args_fn_name="get_init_inputs",
    sample_args_fn_name="get_sample_inputs",
    dialect="linalg-on-tensors",
    ir_context=ir.Context()
)
print(f"{mlir_module_ir}")
```

TODO: Import from saved model, protobuf, etc

Pass/Transform Pipeline

Wrapped in lh-opt tool

```
driver = CompilerDriver(filename, stages)

optimized_module = driver.run()

print(optimized_module)

PipelineDriver & TransformDriver for hand
crafted compilers...
```

- The `filename` parameter is an MLIR file (text form), but the pipeline also accepts `ir.Module`
- The `stages` parameter is a list of stages (passes, bundles, transforms, etc.)
- The **run** method just iterates through the stages and **applies** them to the evolving payload.

```
           Pass List:  lh-opt --stage=one-shot-bufferize --stage=canonicalize payload.mlir
     Descriptor File:  lh-opt --stage=pipeline-check.yaml payload.mlir
       MLIR Schedule:  lh-opt --stage=my-transform.mlir payload.mlir
     Python Schedule:  lh-opt --stage=pipeline-check.py[gen=create_schedule] --stage=canonicalize payload.mlir
 Schedule Arguments:  lh-opt --stage=pipeline-check.py[]{skip_llvm=False} payload.mlir
```

Pipeline Descriptor Format (YAML)

Collection of composable pipelines

```
main-pipeline.yaml:
 Pipeline:
  # Include other YAML files, intermixed with calling other stage types
  include: linalg-stages.yaml
  bundle: CleanupBundle
  include: device-stages.yaml
  # The lower levels can be common to most pipelines before execution
  bundle: BufferizationBundle
  bundle: MLIRLoweringBundle
  bundle: LLVMLoweringBundle
```

Upstream passes

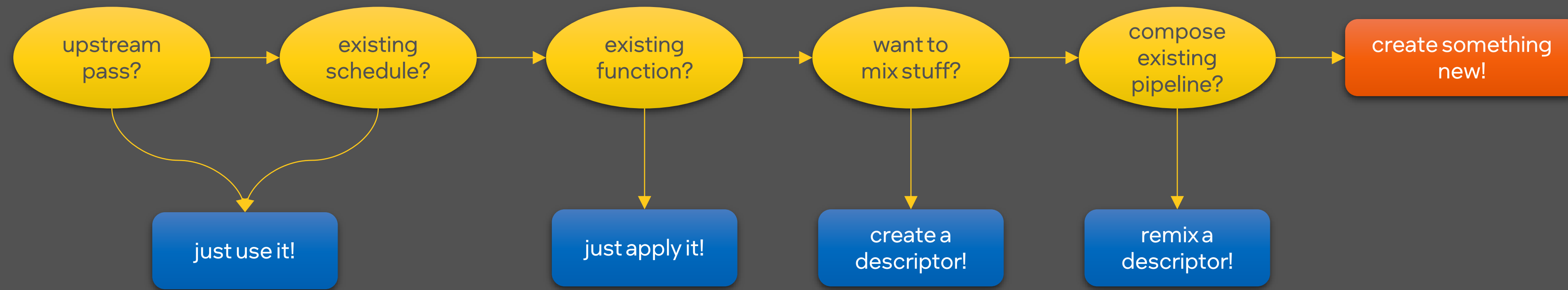
```
$ lh-opt –stage=main-pipeline.yaml payload.mlir
```

```
linalg-stages.yaml:
 Pipeline:
  pass: linalg-generalize-named-ops
  # Stages can have arguments…
  pass: linalg-tile-and-fuse-tensor-ops{tile-sizes=32,32}
  transform: linalg-reorder-loops.mlir{access_pattern=k,n,m}
  pass: eliminate-empty-tensors
```

```
device-stages.yaml:
 Pipeline:
  # One could parametrize further…
  transform: my-cpu-mega-sched.py{target=$target cache=$l1d}
  pass: canonicalize
```

Why provide so much choice?

- Principle of *least effort*



- Reuse, compose, reorder: **dynamically** building the pipeline
  - Pipelines can be created dynamically, bundled, split, *conditional**
  - Based on arguments, properties or even payload analysis

(*) coming soon

Running and benchmarking

Wrapped in lh-run tool

```
# Optimize/lower the module
module = driver.run()

# Arguments as NumPy arrays
for shape in input_shapes:
  buffers.append(KernelArgument(shape).argument)

# Create the runner and execute / benchmark
runner = Runner(module, shared_libs)
runner.execute(host_input_buffers=buffers) # warmup

times = runner.benchmark(host_input_buffers=buffers)
return mean(times)
```

- Composes naturally with the `CompilerDriver`
- Automatically initializes input arguments
  - NumPy arrays that are passed to the execution
  - Ones/Zeros, Random, Identity
- Shared libraries for GPU execution
  - MLIR runtime added automatically
- Benchmark wraps entry point in a loop
  - Return execution timers (no Python cost)
  - FIXME: Needs early transform to create the wrapper

Auto-tuners

- Reuse `PipelineDriver` and `Runner` to iterate through variants
  - Multi-version input module, run various pipelines, execute options
  - Using timer as feedback, update pipelines / modules, repeat
- `Stages` can be customized dynamically
  - Schedules and pass pipelines can be built on demand
  - Depending on payload analysis, target information, profile results
- See Tuomas' presentation at the MLIR Workshop (Mon 10:00)
  - Slides will be available if you missed

Testing Infrastructure

- Current CI tests all examples and tools
  - Validate those assumptions against a nightly build of MLIR
  - Running on x86 and Arm (pre/post commit CI green)

- Additional testing at Intel on XeGPU (performance)
  - Test is upstream, CI loop is downstream
  - It’s a *canary* for Intel Xe team, no noise upstream

- Move to upstream validation in due time
  - Add AMD, Nvidia, RISCV, etc.

Prototyping

- Create dialects and transforms in Python
  - Quickly iterate using MLIR's own Python bindings
  - Upstream changes to MLIR once done
  - See Rolf's presentation (Tue 14:15)

- Create, compose, rehash pipelines
  - Find common *"bubbles"* to reuse across payloads/targets
  - Define and agree upon canonical forms upstream

- Running and benchmarking on multiple targets
  - Use target descriptors, cost models, parametrization to differentiate

Where are we going?

... now that we have *some* infrastructure to play with ...

Collect Usage Patterns & Pipelines

- Examine public workloads: **KernelBench**
  - End-to-end KB -> CPU/GPU benchmarking
  - Arguments, return values, shapes, initializers
- Discover common pipelines
  - Using composition/parametrization to find common ground
  - Update upstream MLIR patterns/transforms to facilitate
- Classify target specific behaviours
  - CPU (Arm, x86, RV), GPU (Xe, NV, AMD), etc.
  - See Adam's quick talk (Wed 11:00)

Validate and reinforce dialect assumptions

- Simplify existing dialects
  - Deprecate unused/redundant forms with solid alternatives
- Encode semantics of existing ops
  - Tests on forms and transforms, from higher-level ingress (ex. PyTorch)
  - Complex sub-graphs with different dialects / ops usage
- Work with downstream, propagate expectations
  - Discuss forms, propose changes, apply upstream, trickle down, repeat

Summary

Main Goals

- Short Term (<1y):
  - Discover and codify common usage patterns
  - Converge usage, group patterns, define forms
  - Provide a package to help quickly prototype compilers
  - Provide a testing suite to validate upstream MLIR assumptions
- Medium Term (1-2y):
  - Focus on exploration and research, not production quality
  - Used by community to discover and discuss design points in MLIR
  - Official test-suite buildbot on all MLIR commits (pre-commit too?)
- Long Term (2+y):
  - Widely adopted way to prototype MLIR based compilers

current work

ramping up…

Thank You!

MLIR Round tables:

- Lighthouse, today at 16:15
- Direct Lowering, today at 17:15
- Early Exit, tomorrow at 10:30
- Canonicalization, tomorrow at 16:15

intel