MLIR data visualization using PassInstrumentation

MLIR Open Design Meeting July 22, 2021
Scott Todd

Context:
Overview

- Intro to "MLIR Pipeline Visualizer Prototype"
- Background: similar tools and motivation for making other visualizations
- Deeper dive into pipeline visualizer prototype
- Open questions around implementation, discussion
MLIR Pipeline Visualizer Prototype

MLIR Dialect Ops for simple_abs_llvmmaot

https://scotttodd.github.io/iree-llvm-sandbox/web-tools/pipeline-visualizer/ (interactive and includes more samples)
Background - IR Printing

Compiler work involves looking at lots of IR:

- Working on a specific component
- Inspecting the behavior of a larger pipeline
- Teaching new developers about system architecture

Syntax highlighting helps with viewing and printing options help slice the IR in different ways, but you can still end up with 100MB+ text files which are difficult to spot patterns in.

```
/mlir-disable-threading
/mlir-elide-elementsattrs-if-larger=N
/-print-ir-after-`
/-print-ir-after-change
```

(see [https://mlir.llvm.org/docs/PassManagement/#ir-printing](https://mlir.llvm.org/docs/PassManagement/#ir-printing))

On the left:

[Links to additional resources]

- [https://gist.github.com/ScottTodd/d0fe0f735f7533bc09692227f56e944b](https://gist.github.com/ScottTodd/d0fe0f735f7533bc09692227f56e944b)
For analyzing performance or viewing execution characteristics, run under a profiler like [https://github.com/wolfpld/tracy](https://github.com/wolfpld/tracy)

- Frames (on the left) require recording pass start/stop times (e.g. by using PassInstrumentation like in [iree/compiler/Utils/TracingUtils.cpp](https://github.com/wolfpld/tracy))
- Sampling (on the right) can work without source modifications
IREE simple_abs (LLVM CPU)

MLIR Dialect Ops for simple_abs_llvmmaot

![Chart showing MLIR Dialect Ops for simple_abs_llvmmaot](image-url)
IREE simple_abs (LLVM CPU)

MLIR Dialect Ops for simple_abs_llvmaot

- Input legalization, shapes, enter into linalg + flow
- HAL, prep for LLVM codegen
- Flow -> HAL, LLVM codegen
- Serialize executables, cleanup
- HAL -> VM, finalize

Diagram showing the operational count for different passes and dialects such as flow, hal, iree, linalg, llvm, memref, std, and vm.
**IREE bert_encoder (LLVM CPU)**

`bert_encoder_unrolled_fake_weights.mlir` (transformer-based machine learning model for natural language processing)

**MLIR Dialect Ops for bert_encoder_llvmaot**

- **DispatchLinalgOnTensors, DeduplicateExecutables**
- **ConvertAffineToStandard**
- **ConvertToHAL**
- **Canonicalize**

**LLVM codegen** (parallel across functions)

15 dialects!
Implementation - MLIR C++ to generate JSON

```cpp
void runAfterPass(Pass *pass, Operation *op) override {
  jsonS.begin();
  jsonS.attribute("passName", pass->getName());
  llvm::StringMap<int> opDialectCounts;
  auto topLevelOp = op;
  while (auto *parentOp = topLevelOp->getParentOp())
    topLevelOp = parentOp;
  topLevelOp->walk([](Operation *opWithinModule) {
    auto opDialectNamespace = opWithinModule->getDialect()->getNamespace();
    // Skip built-in ops (ModuleOp, FuncOp, etc.)
    if (opDialectNamespace.empty())
      return;
    opDialectCounts[opDialectNamespace]++;
  });
  jsonS.attributeBegin("opDialectCounts");
  jsonS.arrayBegin();
  for (const auto &opDialectCount : opDialectCounts) {
    jsonS.begin();
    jsonS.attribute("dialectName", opDialectCount.first);
    jsonS.attribute("opCount", opDialectCount.second);
    jsonS.end();
  }
  jsonS.arrayEnd();
  jsonS.attributeEnd();
  jsonS.end();
}
```

---

**source**

**sample**
Implementation - Webpage with interactive chart

~200 lines of code ([source](#)) split between HTML and JS

- Load JSON
- Process data into chart series
- Create chart using [canvasJS](#) and set styling
Limitations / Open Questions

- PassInstrumentation instances operate on PassManager instances. A single compilation may use multiple (nested or not) PassManagers. IREE even splits between several binaries (iree-import-tf, iree-translate).
- Linking from the chart to IR would help dig deeper, ideally with before → after for a highlighted pass
  - IRPrinterInstrumentation almost works for this, but nested passes are tricky

Ideas:

- Maybe add a monotonically increasing identifier and/or a timestamp identifying each pass for runBeforePass/runAfterPass? Then could write multiple JSON files and join them together.
- Other metadata would be nice to access somehow and write into the JSON: compiler tool version number / commit hash, input flags, source location where pass is added (disambiguate Canonicalize)
Contribute upstream?

C++ instrumentation that outputs JSON seems straightforward enough to contribute

- Could expand with other metrics and use to drive other visualizations or data analyses

What about the HTML/JS visualization code / possible hosted webpage?

Could adapt in some way to fit within editor extensions (like the VSCode one)