MLIR data visualization using PassInstrumentation

MLIR Open Design Meeting July 22, 2021

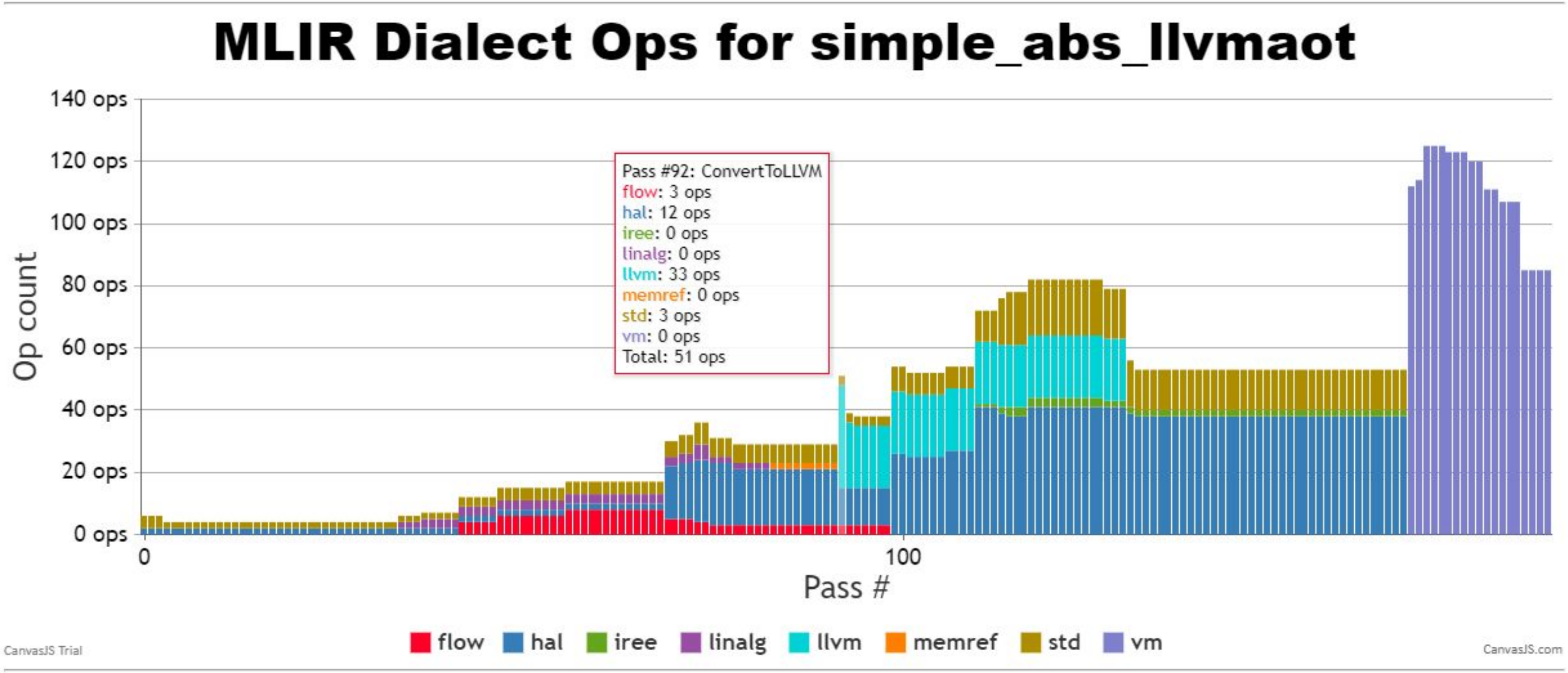
Scott Todd

Context:
https://llvm.discourse.group/t/data-visualizations-and-correlating-data-between-passinstrumentations/3870

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



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

Background - IR Printing

IR Dump After mlir::iree_compiler::Shape::`anonymous-namespace'::ExpandFunctionDynamicDimsPass

```
func @abs(%arg0: !hal.buffer_view) -> !hal.buffer_view attributes {iree.abi.stub} {
  %0 = hal.tensor.cast %arg0 : !hal.buffer_view -> tensor<f32>
  %1 = absf %0 : tensor<f32>
  %2 = hal.tensor.cast %1 : tensor<f32> -> !hal.buffer_view
  return %2 : !hal.buffer_view
}
```

IR Dump After ConvertElementwiseToLinalg

```
func @abs(%arg0: !hal.buffer_view) -> !hal.buffer_view attributes {iree.abi.stub} {
  %0 = hal.tensor.cast %arg0 : !hal.buffer_view -> tensor<f32>
  %1 = linalg.generic {indexing_maps = [affine_map<() -> ()>, affine_map<() -> ()>], iterator_types = []} ins(%0 : ten
  ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
    %3 = absf %arg1 : f32
    linalg.yield %3 : f32
  } -> tensor<f32>
  %2 = hal.tensor.cast %1 : tensor<f32> -> !hal.buffer_view
  return %2 : !hal.buffer_view
}
```

IR Dump After FusionOfTensorOps

```
func @abs(%arg0: !hal.buffer_view) -> !hal.buffer_view attributes {iree.abi.stub} {
  %0 = hal.tensor.cast %arg0 : !hal.buffer_view -> tensor<f32>
  %1 = linalg.init_tensor [] : tensor<f32>
  %2 = linalg.generic {indexing_maps = [affine_map<() -> ()>, affine_map<() -> ()>], iterator_types = []} ins(%0 : ten
  ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
    %4 = absf %arg1 : f32
    linalg.yield %4 : f32
  } -> tensor<f32>
  %3 = hal.tensor.cast %2 : tensor<f32> -> !hal.buffer_view
  return %3 : !hal.buffer_view
}
```

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.

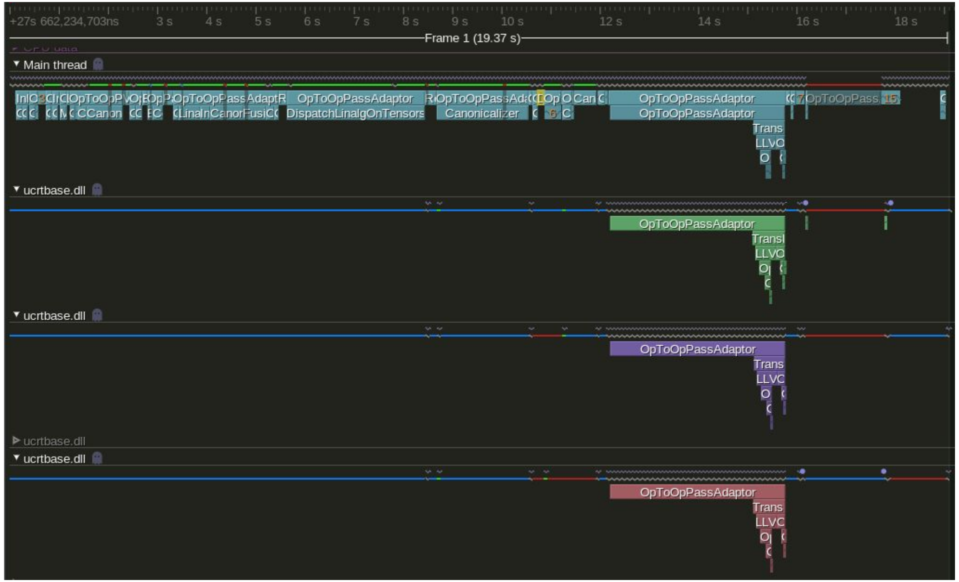
```
-mlir-disable-threading
-mlir-elide-elementsattrs-if-larger=N
-print-ir-after-*
-print-ir-after-change
```

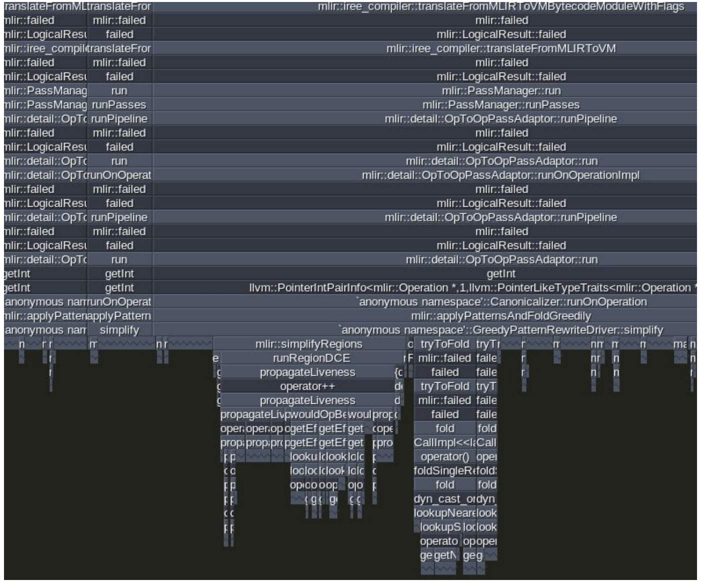
(see https://mlir.llvm.org/docs/PassManagement/#ir-printing)

On the left:

https://gist.github.com/ScottTodd/d0fe0f735f7533bc09692227f56e944b
https://github.com/google/iree/blob/main/scripts/ir_to_markdown.py

Background - Tracing with frame/sampling profilers



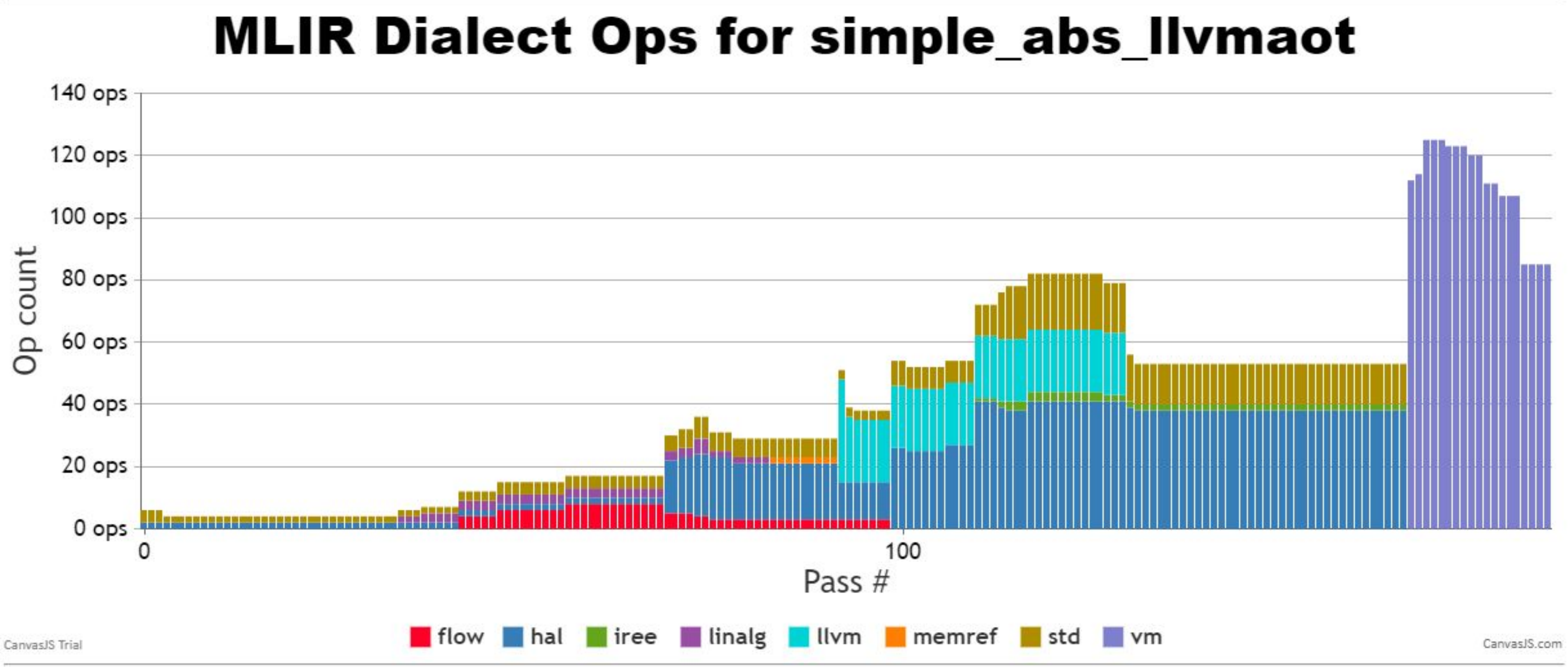


For analyzing performance or viewing execution characteristics, run under a profiler like https://github.com/wolfpld/tracy

- Frames (on the left) require recording pass start/stop times (e.g. by using `PassInstrumentation` like in iree/compiler/Utils/TracingUtils.cpp)
- Sampling (on the right) can work without source modifications

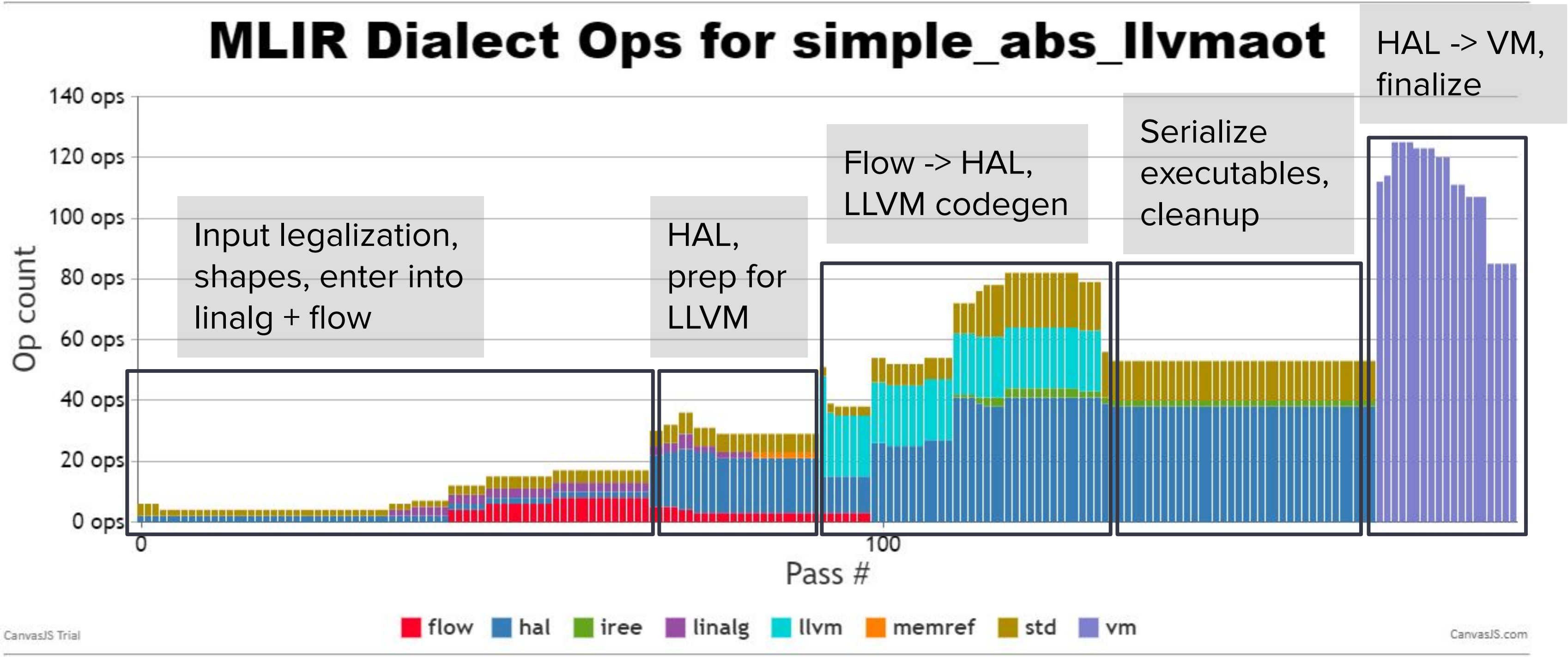
IREE simple_abs (LLVM CPU)

```
func @abs(%input : tensor<f32>) -> (tensor<f32>) {
  %result = absf %input : tensor<f32>
  return %result : tensor<f32>
}
```



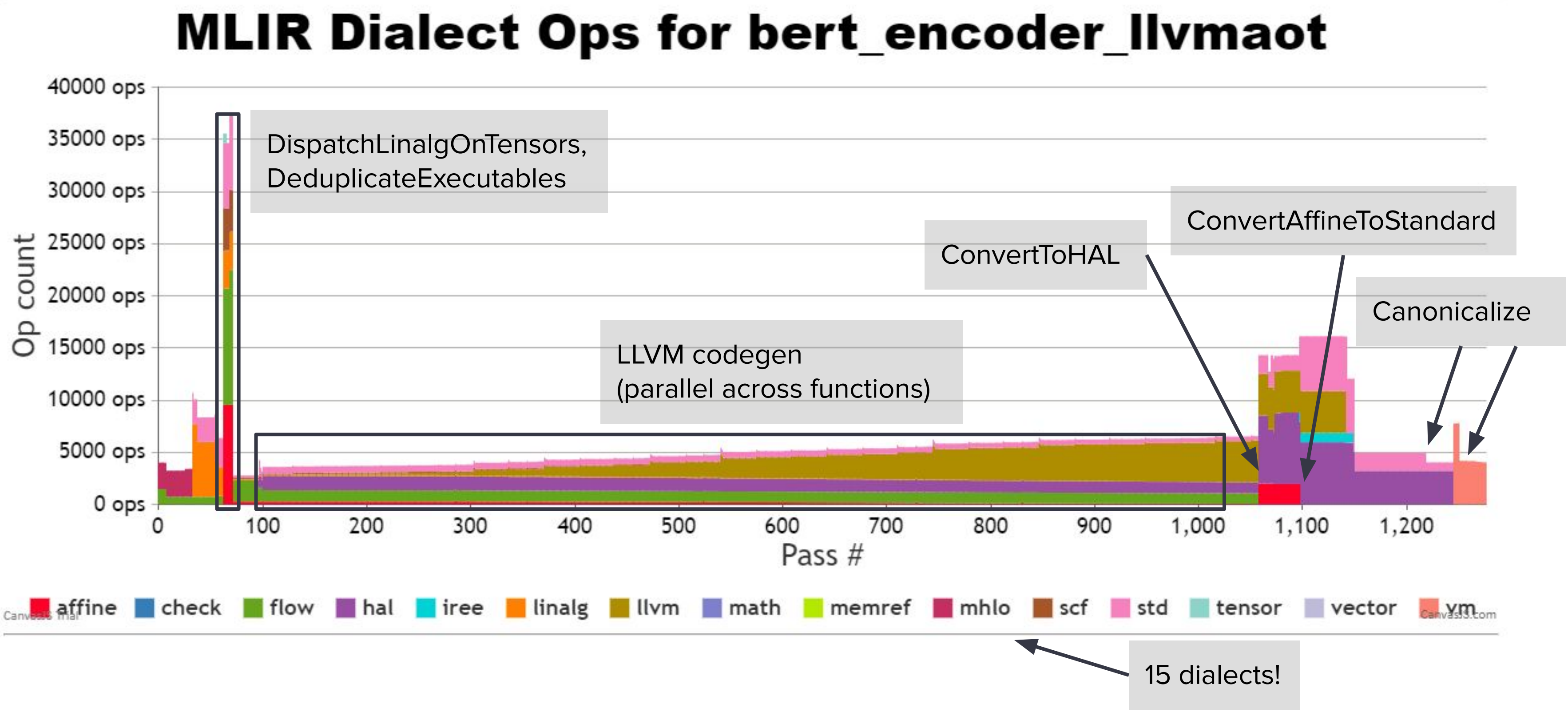
IREE simple_abs (LLVM CPU)

```
func @abs(%input : tensor<f32>) -> (tensor<f32>) {
  %result = absf %input : tensor<f32>
  return %result : tensor<f32>
}
```



IREE bert_encoder (LLVM CPU)

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



Implementation - MLIR C++ to generate JSON

```
+  void runAfterPass(Pass *pass, Operation *op) override {
+    jsonOS.objectBegin();
+    jsonOS.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]++;
+    });
+
+    jsonOS.attributeBegin("dialectOpCounts");
+    jsonOS.arrayBegin();
+    for (const auto &opDialectCount : opDialectCounts) {
+      jsonOS.objectBegin();
+      jsonOS.attribute("dialectName", opDialectCount.getKey());
+      jsonOS.attribute("opCount", opDialectCount.getValue());
+      jsonOS.objectEnd();
+    }
+    jsonOS.arrayEnd();
+    jsonOS.attributeEnd();
+
+    jsonOS.objectEnd();
+  }
```

source

3184 lines (3184 sloc) | 52.9 KB

```
[
  {
    "passName": "mlir::iree_compiler::IREE::ABI::WrapEntryPointsPass",
    "dialectOpCounts": [
      {
        "dialectName": "hal",
        "opCount": 2
      },
      {
        "dialectName": "std",
        "opCount": 4
      }
    ]
  },
  {
    "passName": "Canonicalizer",
    "dialectOpCounts": [
      {
        "dialectName": "hal",
        "opCount": 2
      },
      {
        "dialectName": "std",
        "opCount": 4
      }
    ]
  },
```

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)