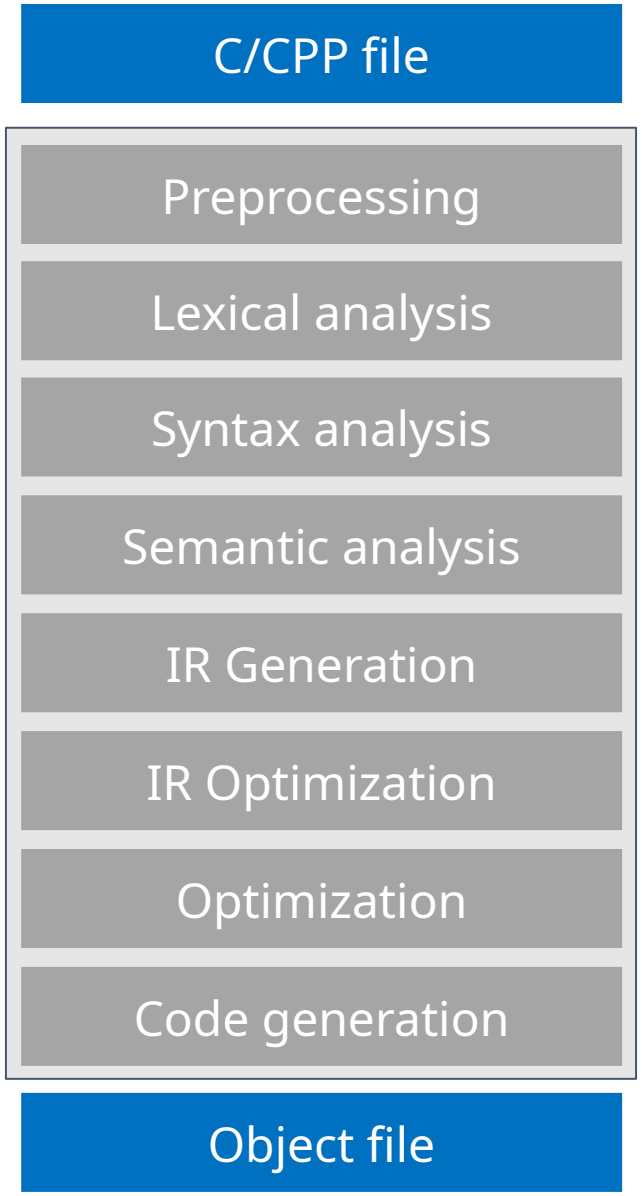
Compilers 101

MLIR - Part 1

Compilation flow

- Is that it?
- Can we have different inputs?
- Is there a chance for an alternative flow?



Today

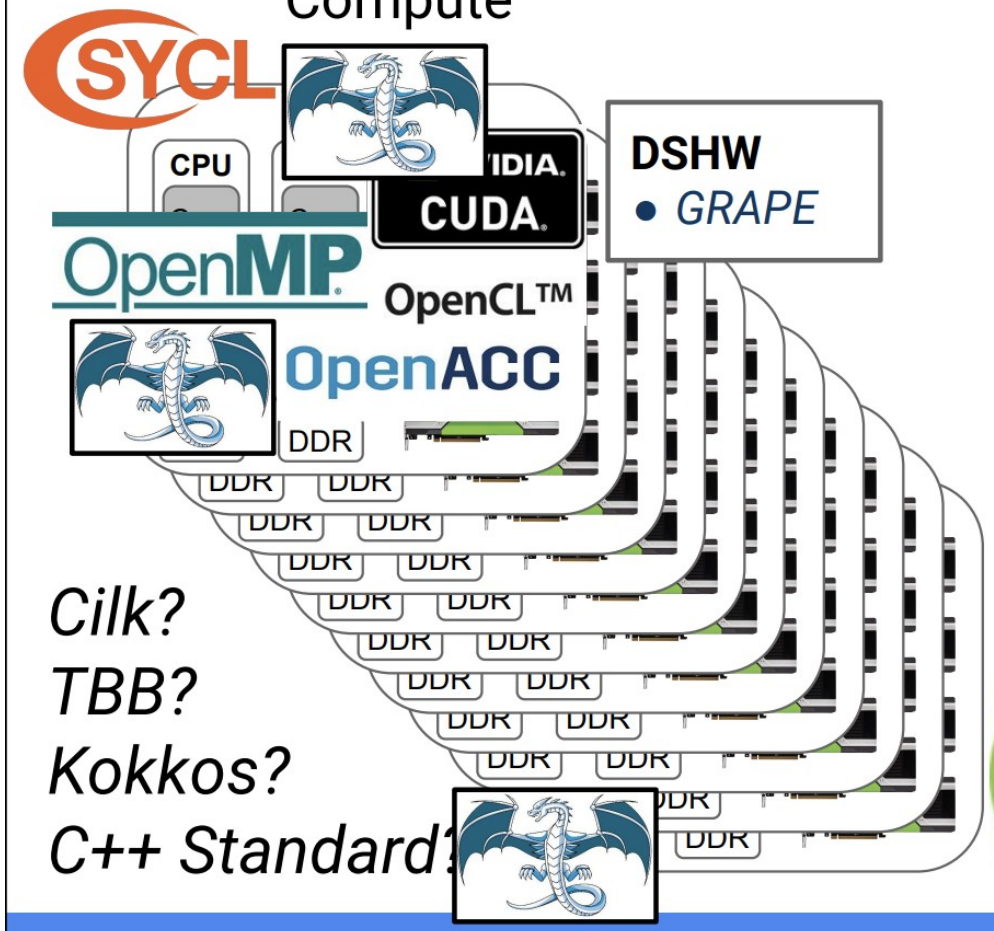
- Why MLIR?
- MLIR history
- MLIR description
- Usage scenarios

Your Typical HPC Setup

Compute

Network

Storage



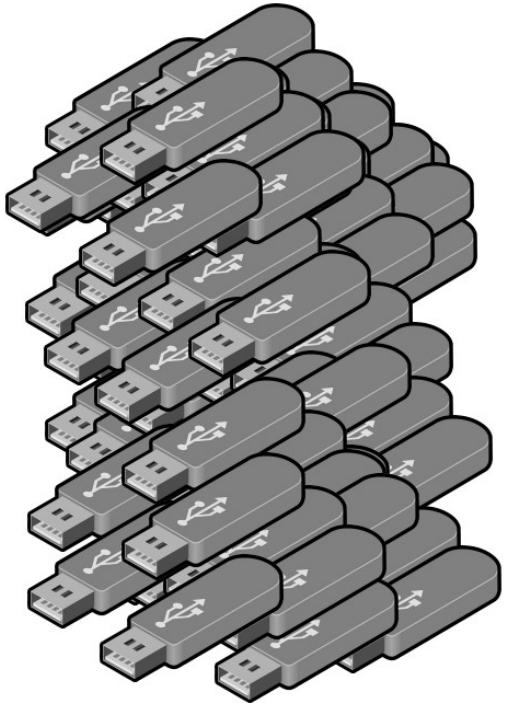
DSLs
- Ebb
- CLAW
- Liszt
- Spiral?

MPI

GASNet, Charm++

Legion?

CRAY CHAPEL



https://llvm-hpc-2020-workshop.github.io/presentations/llvmhpc2020-amini.pdf

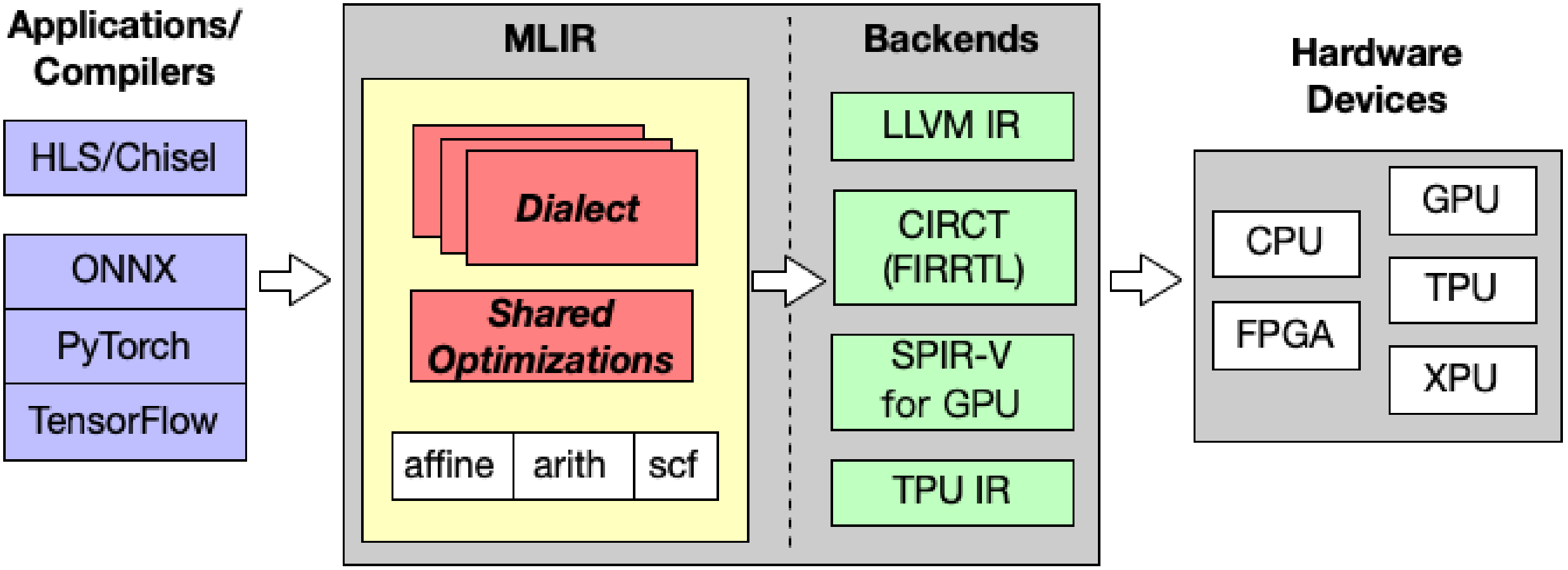
MLIR intro

MLIR stands for Multi-Level Intermediate Representation

- MLIR is a framework for building custom intermediate representations
- Collection of basic algorithms for general-purpose IR transformations
- Basic data structures
- Custom parsers and printers

https://blog.google/technology/ai/mlir-accelerating-ai-open-source-infrastructure/

MLIR



http://lastweek.io/notes/MLIR/

MLIR

- Supports diverse domains
  - Machine Learning
  - Hardware development
  - Custom Domain Specific Languages (DSL)
  - LLVM IR
- Provides a view on the source code from different levels

MLIR history: pre-historic era

- Before MLIR was conceived, the LLVM project already had a powerful intermediate representation (IR) that was highly effective for optimizing and generating machine code for CPUs.
- However, as the landscape of computing hardware grew to include GPUs, TPUs, and other accelerators, it became apparent that LLVM IR was not ideally suited for representing the high-level, domain-specific abstractions needed by these new architectures.

MLIR history: beginning

- MLIR was officially announced and open-sourced by Google in April 2019, but its development started much earlier, around 2018, as a collaboration between different teams within Google and eventually with external contributors
- The project was born out of the need for an infrastructure that could bridge the gap between high-level, domain-specific representations of programs and the low-level, hardware-specific code required to execute them efficiently

MLIR history: nowadays

MLIR infrastructure is being developed rapidly

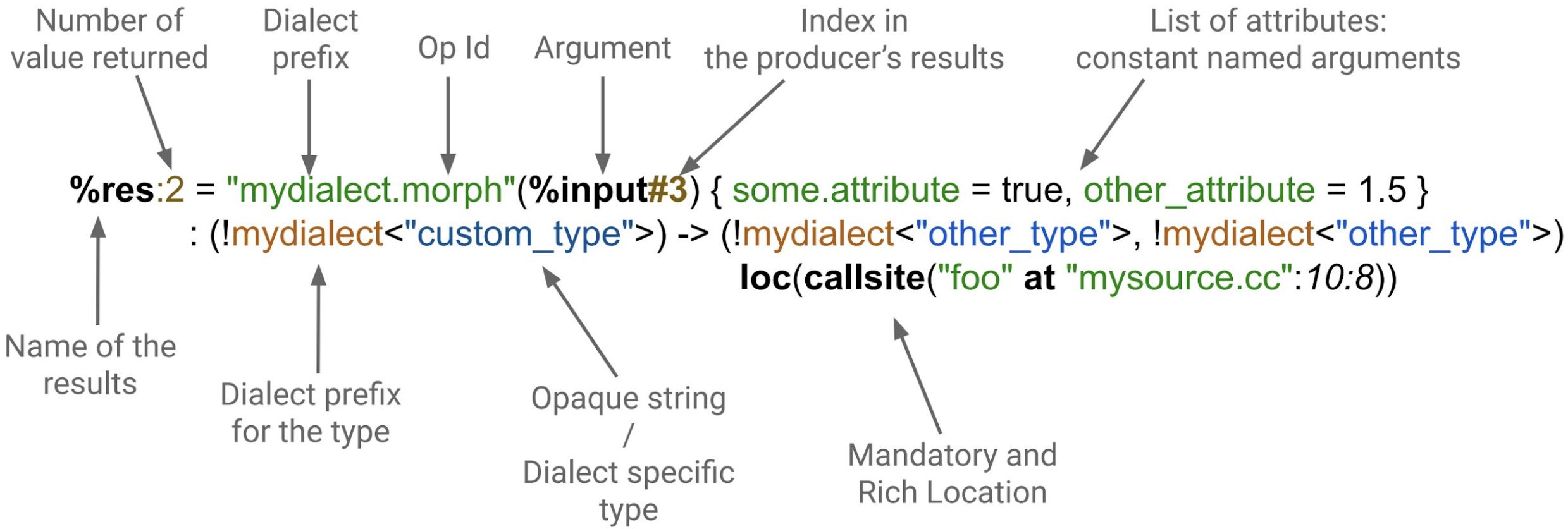
Top industry companies use that:

- Google
- Meta
- Microsoft
- Intel
- AMD

other…

Operations, Not Instructions

- No predefined set of instructions
- Operations are like “opaque functions” to MLIR



https://mlir.llvm.org/docs/LangRef/#operations
https://github.com/llvm/llvm-project/blob/master/mlir/include/mlir/IR/Operation.h#L27

Dialects

- Extensible collections of operations and types that represent various levels of abstraction or domains.
- Dialects represent modular structure of combining operations
- Dialects are basically custom IRs
- Custom data types, custom operations, custom syntax
- Example dialects: arithmetic, affine, omp, spv, linalg

Dialects

Documentation:

https://mlir.llvm.org/docs/Dialects/

- 'acc' Dialect
- 'affine' Dialect
- 'amdgpu' Dialect
- 'amx' Dialect
- 'arith' Dialect
- 'arm_neon' Dialect
- 'arm_sve' Dialect
- 'ArmSME' Dialect
- 'async' Dialect
- 'bufferization' Dialect
- 'cf' Dialect
- 'complex' Dialect
- 'dlti' Dialect
- 'emitc' Dialect
- 'func' Dialect
- 'gpu' Dialect
- 'index' Dialect
- 'irdl' Dialect
- 'linalg' Dialect
- 'llvm' Dialect
- 'math' Dialect
- 'memref' Dialect
- 'mesh' Dialect
- 'ml_program' Dialect
- 'mpi' Dialect
- 'nvgpu' Dialect
- 'nvvm' Dialect
- 'omp' Dialect
- 'pdl_interp' Dialect
- 'pdl' Dialect
- 'quant' Dialect
- 'rocdl' Dialect
- 'scf' Dialect
- 'shape' Dialect
- 'sparse_tensor' Dialect
- 'tensor' Dialect
- 'ub' Dialect
- 'vcix' Dialect
- 'vector' Dialect
- 'x86vector' Dialect
- Builtin Dialect
- SPIR-V Dialect
- Tensor Operator Set Architecture (TOSA) Dialect
- Transform Dialect

Multi level examples

```
func.func @vector_add(%A: vector<100xf32>, %B: vector<100xf32>) -> vector<100xf32> {
  %C = arith.addf %A, %B : vector<100xf32>
  return %C : vector<100xf32>
}
```

Vector: https://godbolt.org/z/oj1TErxv9

```
func.func @vector_add(%A: memref<100xf32>, %B: memref<100xf32>, %C: memref<100xf32>) {
  affine.for %i = 0 to 100 {
    %a = affine.load %A[%i] : memref<100xf32>
    %b = affine.load %B[%i] : memref<100xf32>
    %sum = arith.addf %a, %b : f32
    affine.store %sum, %C[%i] : memref<100xf32>
  }
  return
}
```

Affine: https://godbolt.org/z/P7KTrW665

```
func.func @vector_add(%A: memref<100xf32>, %B: memref<100xf32>, %C: memref<100xf32>) {
  %c0 = arith.constant 0 : index
  %c100 = arith.constant 100 : index
  %c1 = arith.constant 1 : index
  scf.for %i = %c0 to %c100 step %c1 {
    %a = memref.load %A[%i] : memref<100xf32>
    %b = memref.load %B[%i] : memref<100xf32>
    %sum = arith.addf %a, %b : f32
    memref.store %sum, %C[%i] : memref<100xf32>
  }
  return
}
```

scf + memref: https://godbolt.org/z/o883bEjr7

MLIR transformations

Affine -> scf:  https://godbolt.org/z/zcYnsTEvz

scf -> cf:  https://godbolt.org/z/95o1nvrsM

SCF - structured control flow

CF - control flow

Why do we need dialects?

- MLIR is suited to define IRs on different levels of abstractions
- Built-in dialects
  - represent more or less generic operations (arith, scf, memref, llvm)
  - operations used in LLVM stack and needed in the community (x86vector, arm_sve)
- It provides an infrastructure to add different dialects
  - ML frameworks use that (PyTorch aten dialect, onnx dialect, tf dialect) to represent supported network layers for further lowering

Recursive nesting: Operations -> Regions -> Blocks

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.
^block(%argument: !d.type):
    %value = "nested.operation"() ({
      // Ops can contain nested regions.
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
^other_block:
    "d.terminator"() [^block(%argument : !d.type)] : ()
-> ()
}) : () -> (!d.type, !d.other_type)
```

Region

Block

Region

- Regions are list of basic blocks nested inside of an operation.
  - Basic blocks are a list of operations: **the IR structure is recursively nested**!
- Conceptually similar to function call, but can reference SSA values defined outside.
- SSA values defined inside don't escape.

https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/
https://mlir.llvm.org/docs/LangRef/#high-level-structure

Block arguments

```
func @example_function() {
  %0 = "some_operation"() : () -> i32
  "branch_conditional"(%0) ( {
    // Block 1: true branch
    ^bb0(%arg0: i32):  // %arg0 is a block argument
      %1 = "some_other_operation"(%arg0) : (i32) -> i32
      "another_operation"(%1) : (i32) -> ()
      "branch"(%1) : (i32) -> ()
  }, {
    // Block 2: false branch
    ^bb1(%arg1: i32):  // %arg1 is a block argument
      "yet_another_operation"(%arg1) : (i32) -> ()
      "branch"(%arg1) : (i32) -> ()
  }) : (i32) -> ()
}
```

Mechanism to modify variables in a structured way that fit the similar purpose as PHI nodes in LLVM IR

Operations

- Describe some action
- Take arguments and returns them

- Have generic op representation and custom printers
  - Example on the next slide
  - a[illegible]

https://godbolt.org/z/cfjY375z7

Example: Affine Dialect

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (8*d0 - 4 >= 0, -8*d0 + 7 >= 0)(%k) {
        // Dead code, because no multiple of 8 lies between 4 and 7.
        "foo"(%k) : (index) -> ()
      }
    }
  }
  return
}
```

With custom parsing/printing: affine.for operations with an attached region feels like a regular for!

Extra semantics constraints in this dialect: the if condition is an affine relationship on the enclosing loop indices.

```
#set0 = (d0) : (d0 * 8 - 4 >= 0, d0 * -8 + 7 >= 0)
func @test() {
  "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> () {
  ^bb1(%i0: index):
    "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> ()
{
    ^bb2(%i1: index):
      "affine.if"(%i0) {condition: #set0} : (index) -> () {
        "foo"(%i0) : (index) -> ()
        "affine.terminator"() : () -> ()
      } { // else block
      }
      "affine.terminator"() : () -> ()
    }
    ...
```

Same code without custom parsing/printing: **isomorphic to the internal in-memory representation.**

https://mlir.llvm.org/docs/Dialects/Affine/

Pass infrastructure

- MLIR provides a pass manager
- Passes can be operation-agnostic or be run on specific operation
- Pass manager is multithreaded
  - Independent operations can be processed simultaneously
- Pass manager is dynamic
  - I.e., allows running another pass manager from within a pass
- MLIR has built-in tools for reproducer generation

mlir-opt

- opt analog for MLIR
- help:  https://godbolt.org/z/x5zjxbq4T

LLVM as a dialect

```
%13 = llvm.alloca %arg0 x !llvm.double : (!llvm.i32) -> !llvm.ptr<double>
%14 = llvm.getelementptr %13[%arg0, %arg0]
          : (!llvm.ptr<double>, !llvm.i32, !llvm.i32) -> !llvm.ptr<double>
%15 = llvm.load %14 : !llvm.ptr<double>
llvm.store %15, %13 : !llvm.ptr<double>
%16 = llvm.bitcast %13 : !llvm.ptr<double> to !llvm.ptr<i64>
%17 = llvm.call @foo(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
%18 = llvm.extractvalue %17[0] : !llvm.struct<(i32, double, i32)>
%19 = llvm.insertvalue %18, %17[2] : !llvm.struct<(i32, double, i32)>
%20 = llvm.constant(@foo : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>) :
        !llvm.ptr<func<struct<i32, double, i32> (i32)>>
%21 = llvm.call %20(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
```

More intro to MLIR: https://mlir.llvm.org/docs/Tutorials/Toy/

An MLIR Dialect for High-Level Optimization of Fortran

Eric Schweitz (NVIDIA)

LOOPS

An example of loop optimization

```
// subroutine convolution(r, f, g)
func @convolution(%r : !fir.box<!fir.array<?:f32>>,  %f : !fir.box<...>,  %g : !fir.box<...>) {
  %uf:3 = fir.box_dims %f, 0 : (!fir.box<...>,  index) -> (index, index, index)  ... // and %ug:3
  fir.loop %n = 1 to %uf#1 {
    fir.loop %k = 1 to %ug#1 {
      %2 = subi %n, %k : index
      %3 = fir.coordinate_of %f, %2 : (!fir.box<...>,  index) -> !fir.ref<f32>
      %4 = fir.load %3 : !fir.ref<f32>  ... // and likewise %6 = load g[k]
      %7 = mulf %6, %4 : f32              ... // and likewise %9 = load r[n]
      %10 = addf %9, %7 : f32
      fir.store %10 to %8 : !fir.ref<f32>
}}}
```

Advantages of MLIR

- Modularity and extensibility
  - Easy to introduce needed dialects, passes and build the pipeline
  - Enables with better compiler development quality and rapid prototyping
- Support for generic and domain specific optimizations
- Enhanced reusability and interoperability
- Improved analysis and debugging
- Facilitates Heterogeneous Hardware Support

Disadvantages of MLIR

- Increased Complexity
  - MLIR's multi-level approach adds extra layers of abstraction compared to traditional single-level IRs
  - As a result, steep learning curve
- Lack of stable API
  - Every commit to LLVM repo can potentially break the IR
- Dialects zoo
  - Everyone comes up with their own dialect
- Lack of tooling and ecosystem maturity
  - While MLIR is continuously evolving, its ecosystem is still maturing

Where MLIR is used?

- flang - Fortran compiler
- clang-based MLIR codegen for C/C++:
  - clangir
  - Polygeist (C/C++/CUDA C++)
- CIRCT Project
- Mojo

List of users updated by LLVM community:

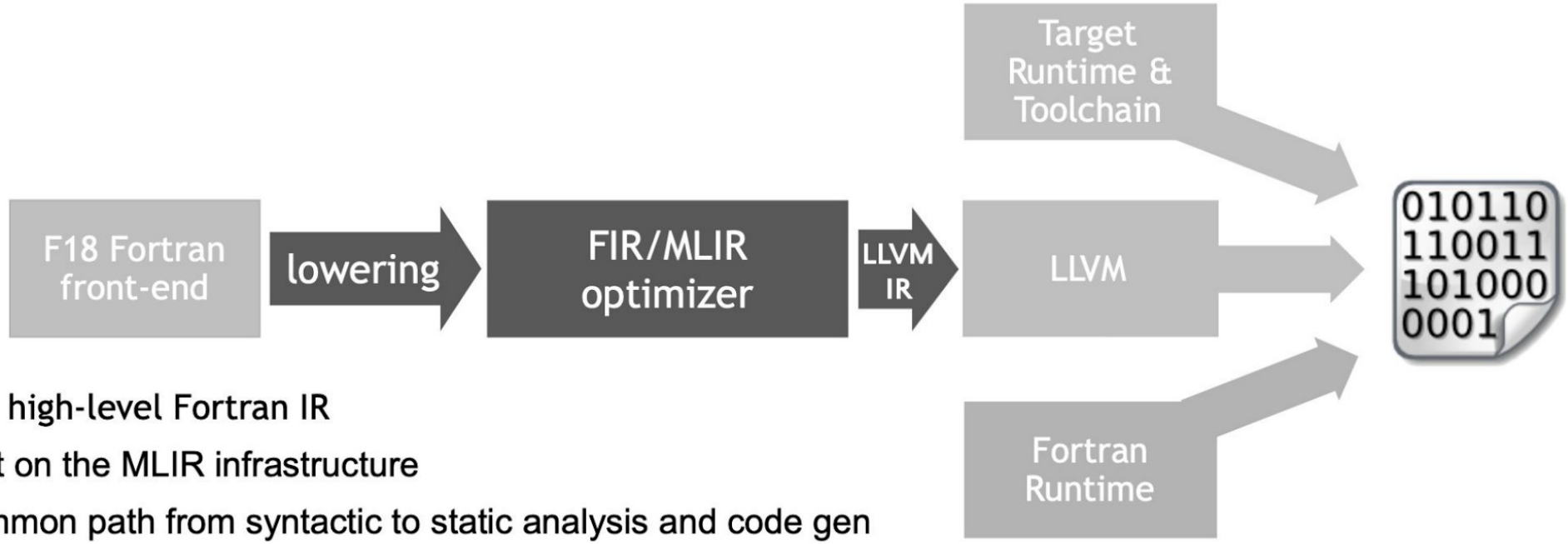
- https://mlir.llvm.org/users/

An MLIR Dialect for High-Level Optimization of Fortran

Eric Schweitz (NVIDIA)

FLANG

The LLVM Fortran compiler



FIR: high-level Fortran IR

Built on the MLIR infrastructure

Common path from syntactic to static analysis and code gen

Shrink abstraction gap: core Fortran operational properties

Focus on writing Fortran aware optimizations

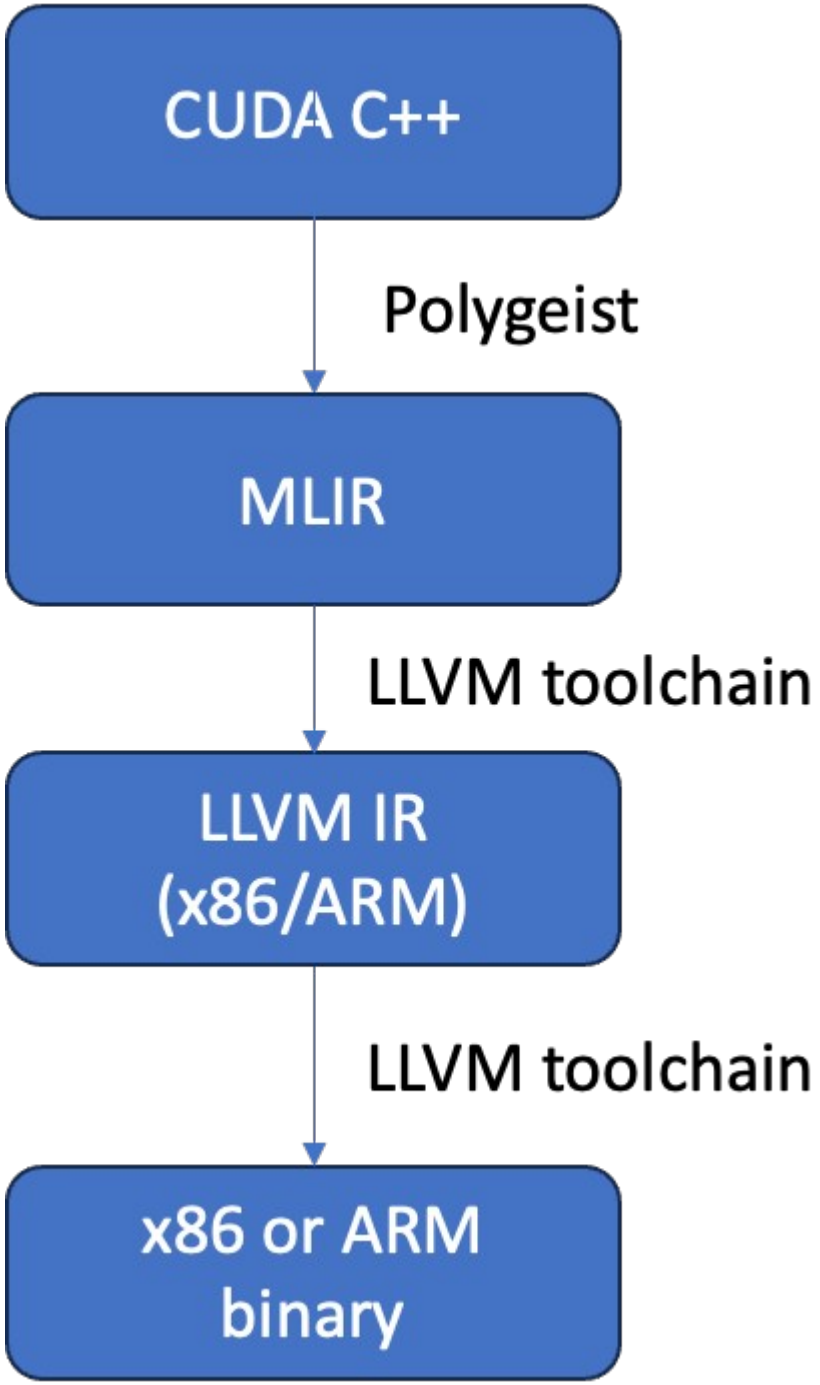
Separation of concerns: constraints checking vs. optimizing computation

NVIDIA

Polygeist

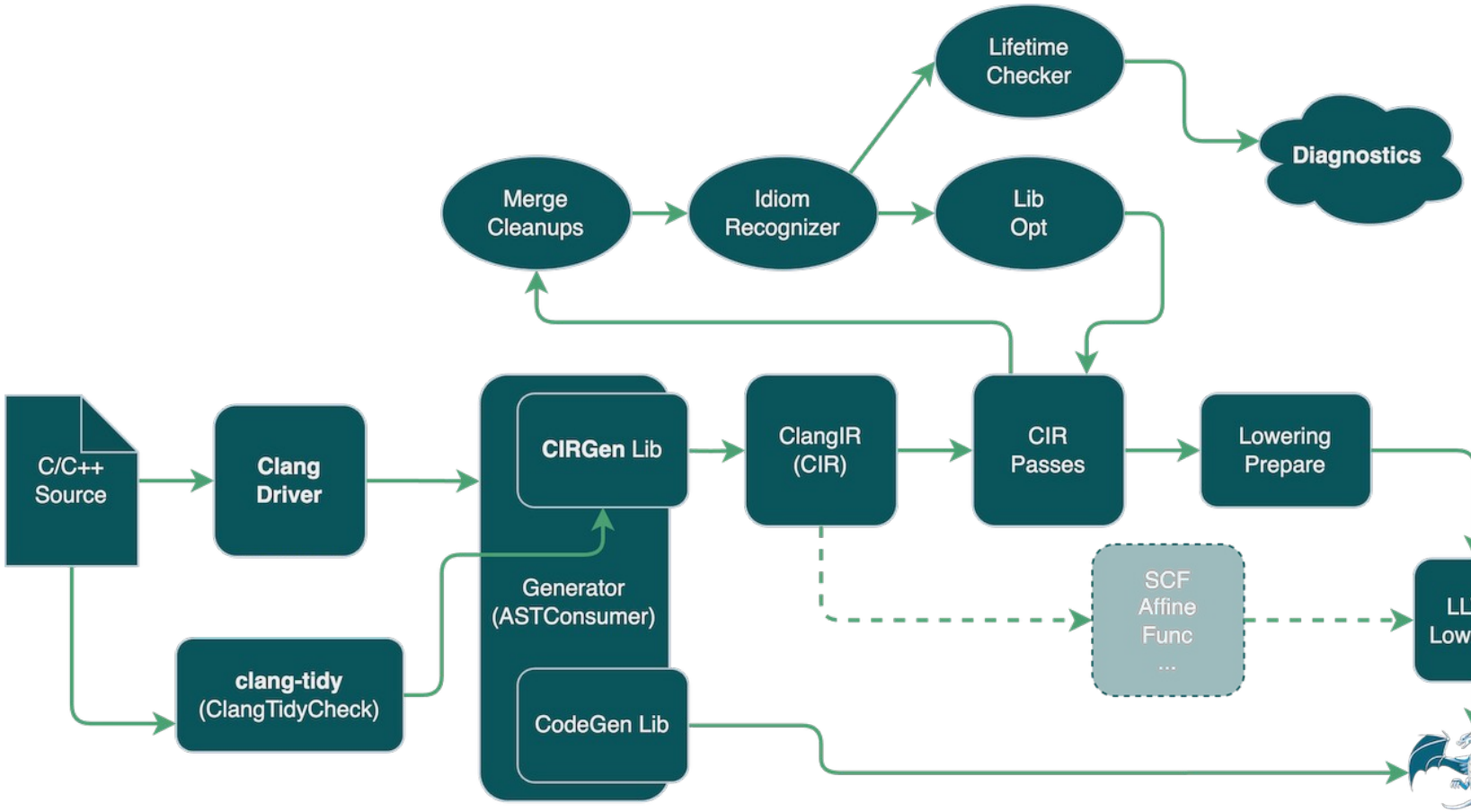
Convert C/C++/CUDA C++ to MLIR and apply polyhedral transformations to run it faster

Project: https://github.com/llvm/Polygeist



clangir

ClangIR integrates Clang's C/C++ front-end capabilities with the MLIR framework, enabling a layered approach to transforming and optimizing code



ClangIR is inspired in the success of other languages that greatly benefit from a middle-level IR, such as Swift and Rust. Particularly, optionally attaching AST nodes to CIR operations is inspired by SIL references to AST nodes in Swift.

https://github.com/llvm/clangir
https://llvm.github.io/clangir/

Mojo 🔥 language

- Mojo combines the usability of Python with the performance of C, unlocking unparalleled programmability of AI hardware and extensibility of AI models.
- Mojo leverages MLIR, which enables Mojo developers to take advantage of vectors, threads, and AI hardware units.

https://www.modular.com/max/mojo

Example: CIRCT Project

Apply MLIR and the LLVM development methodology to the domain of hardware design tools

https://github.com/llvm/circt

Circuit IR Compilers and Tools: MLIR for Hardware Design

Amalee Wilson
Stanford University
amalee@cs.stanford.edu

Stephen Neuendorffer
Xilinx
stephenn@xilinx.com

Chris Lattner
SiFive
clattner@sifive.com

Other Collaborators
Microsoft, PNNL, ETH, EPFL
U. Edinburgh, Cornell, Illinois

Introduction

- **Designing and programming complex, heterogeneous systems-on-chip mixing general purpose and specialized components is difficult.** The EDA industry has well-known and widely used proprietary and open source tools. However, these tools are often inconsistent, have usability concerns, and were not designed together into a common platform.
- **The CIRCT project is a new effort to apply MLIR and the LLVM development methodology to the domain of hardware design tools.** A coherently designed set of abstractions leveraging best practices in compiler infrastructure and compiler design techniques will result in a new generation of tools for both designing and programming complex, heterogeneous systems-on-chip mixing general purpose and specialized components.

Aim

Create a ***comprehensive open-source infrastructure*** capable of representing ***multiple levels of abstraction*** to improve both hardware and software.

Hardware
- Accelerator Design
- Circuit Synthesis
- Memory Hierarchy
- Interconnect Design
- System Simulation

Software
- Accelerator Programming
- Code Generation
- Memory Allocation
- Host Code Partitioning
- JIT Execution

Next generation open source synthesis infrastructure

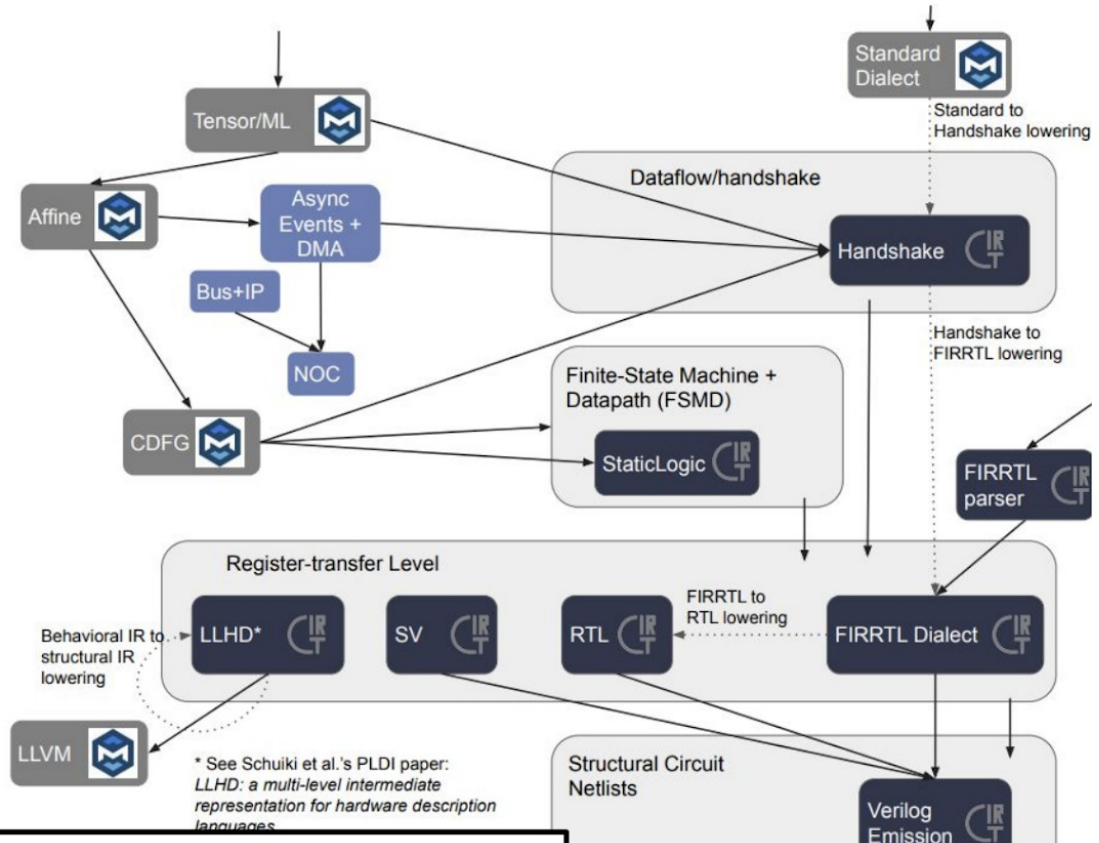
- LLVM incubator project

Focus on RTL level and above

- Interfaces with SystemVerilog, Chisel, C++
- FPGA+ASIC targets
- Integrated simulation through LLVM backends

Leverage unique MLIR Capabilities

- **Parallel Compilation => Reduced design time**
- **Multiple Abstractions/Dialect => Improved predictability**
- **Unified Framework => Better integration** between high level and low level tools
- **Cyclic SSA graphs** (contributed by CIRCT developers) **=> hardware-oriented semar**



Poster from LLVM Dev Meeting'20
https://llvm.org/devmtg/2020-09/

Toy Tutorial

This tutorial runs through the implementation of a basic toy language on top of MLIR. The goal of this tutorial is to introduce the concepts of MLIR; in particular, how dialects can help easily support language specific constructs and transformations while still offering an easy path to lower to LLVM or other codegen infrastructure. This tutorial is based on the model of the LLVM Kaleidoscope Tutorial.

Another good source of introduction is the online recording from the 2020 LLVM Dev Conference (slides).

This tutorial assumes you have cloned and built MLIR; if you have not yet done so, see Getting started with MLIR.

This tutorial is divided in the following chapters:

- Chapter #1: Introduction to the Toy language and the definition of its AST.
- Chapter #2: Traversing the AST to emit a dialect in MLIR, introducing base MLIR concepts. Here we show how to start attaching semantics to our custom operations in MLIR.
- Chapter #3: High-level language-specific optimization using pattern rewriting system.
- Chapter #4: Writing generic dialect-independent transformations with Interfaces. Here we will show how to plug dialect specific information into generic transformations like shape inference and inlining.
- Chapter #5: Partially lowering to lower-level dialects. We'll convert some of our high level language specific semantics towards a generic affine oriented dialect for optimization.
- Chapter #6: Lowering to LLVM and code generation. Here we'll target LLVM IR for code generation, and detail more of the lowering framework.
- Chapter #7: Extending Toy: Adding support for a composite type. We'll demonstrate how to add a custom type to MLIR, and how it fits in the existing pipeline.

https://mlir.llvm.org/docs/Tutorials/Toy/

Next time

- Connection with LLVM
- Available transformations and optimizations overview
- Custom dialects
- Writing a pass

Test

https://forms.gle/bR6K58M7p9YH6QxS8

Submission time: **10 minutes**

В чем состоят отличия строения MLIR и LLVM IR?

Your answer

Каковы преимущества и недостатки MLIR?

Your answer

Backup: me@gooddoog.ru

Extra materials

- LLVM Compiler Infrastructure in HPC Workshop - https://www.youtube.com/watch?v=0bxyZDGs-aA
- All MLIR talks: https://mlir.llvm.org/talks/
- Toy language (MLIR tutorial): https://mlir.llvm.org/docs/Tutorials/Toy/Ch-1/
- 2023 EuroLLVM - What's new in MLIR? https://www.youtube.com/watch?v=LPlRLt9w4b0