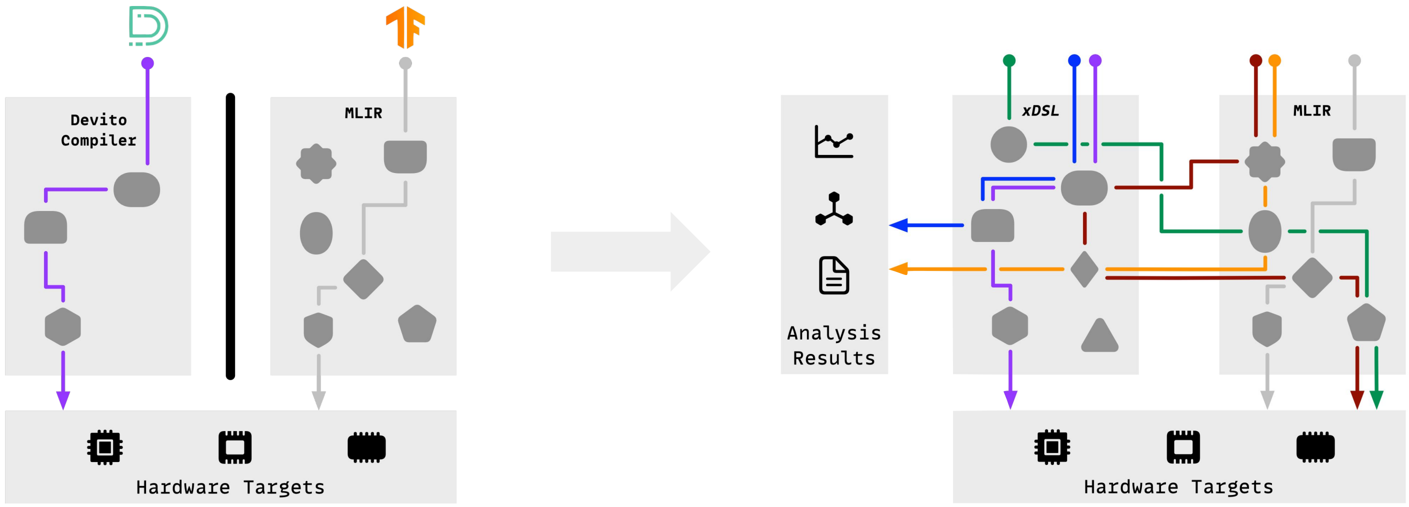


Python-native domain-specific compilation for low-level systems with **xDSL**

Tobias Grosser

UNIVERSITY OF CAMBRIDGE

General-purpose **Software** Specialized

THE C PROGRAMMING LANGUAGE

C++

NVIDIA CUDA

TensorFlow

tvm

Firedrake

How to compile
specialized software to specialized hardware?

General-purpose **Hardware** Specialized

How do we currently build specialized Compilers?

Example 1: Devito HPC DSL

devito Public

Code generation framework for automated finite difference computation

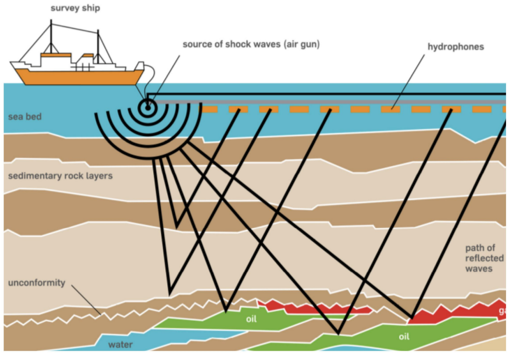
Python ☆ 407 190

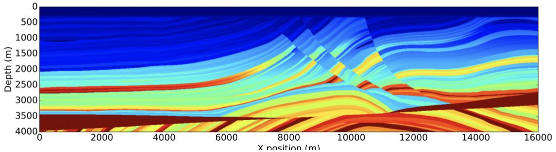
- < 50,000 lines of code
- Compiler implemented in Python
- Uses three IRs to compile
- Applies many classical loop optimizations
- Support for GPUs, no support for hardware accelerators

Usability and **performance**, portable on CPUs, GPUs, but **limited hardware support** and **optimizations**

Devito

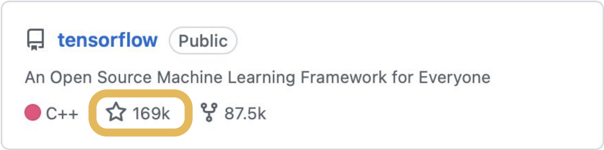
Imperial College London





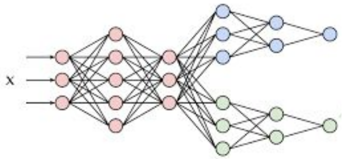
How do we currently build specialized Compilers?

Example 2: TensorFlow, Google's ML Framework



TensorFlow

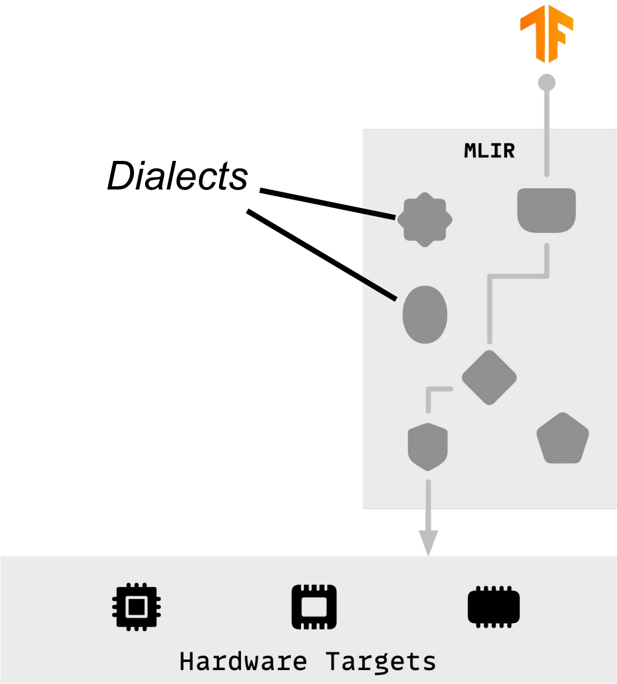
- \> 2,500,000 lines of code
- Compiler implemented in Python & C++
- Uses two IRs with > 500 different types of expressions
- Applies many classical loop optimizations
- Great Performance & Support for custom hardware: TPU



Huge effort to build and maintain, but **great performance**!

MLIR — Multi-Level Intermediate Representation

An LLVM subproject for building reusable and extensible compiler infrastructure



Simple example in MLIR

- A DSL that is aware of stacks

```
fn demo(stack: &stack<i32>) -> i32 {
     i32 c = 5;
     stack.push(c);
     return stack.pop();
}
```

Your Tool

MLIR FORMAT
(sequence of operations with child regions)
+ a selection of dialects

Simple example in MLIR

- A DSL that is aware of stacks

```
fn demo(stack: &stack<i32>) -> i32 {
     i32 c = 5;
     stack.push(c);
     return stack.pop();
}
```

Your Tool

```
func.func @demo(%stack: ???) -> i32 {



     func.return %res
}
```

Simple example in MLIR

- A DSL that is aware of stacks

```
fn demo(stack: &stack<i32>) -> i32 {
     i32 c = 5;
     stack.push(c);
     return stack.pop();
}
```

Your Tool

```
func.func @demo(%stack: ???) -> i32 {
     %c = arith.constant 5 : i32


     func.return %res
}
```

Simple example in MLIR

- A DSL that is aware of stacks

Stack Dialect

```
stack.stack<T>
```

```
stack.push %stack <- %val : T
stack.pop  %stack         : T
```

Simple example in MLIR

- A DSL that is aware of stacks

```
fn demo(stack: &stack<i32>) -> i32 {
     i32 c = 5;
     stack.push(c);
     return stack.pop();
}
```

Your Tool

```
func.func @demo(%stack: stack.stack<i32>) -> i32 {
     %c = arith.constant 5 : i32
     stack.push %stack <- %c : i32
     %res = stack.pop %stack : i32
     func.return %res
}
```

Simple example in MLIR

- A DSL that is aware of stacks

```
func.func @demo(%stack: stack.stack<i32>) -> i32 {
     %c = arith.constant 5 : i32
     stack.push %stack <- %c : i32
     %res = stack.pop %stack : i32
     func.return %res
}
```

MLIR

```
func.func @demo(%stack: stack.stack<i32>) -> i32 {
     %c = arith.constant 5 : i32
     func.return %c
}
```

(with your domain-specific rewrite)

Simple example in MLIR

- A DSL that is aware of stacks

```
func.func @demo(%stack: stack.stack<i32>) -> i32 {
     %c = arith.constant 5 : i32
     func.return %c
}
```

MLIR

```
llvm.func @demo(%stack: llvm.ptr) -> i32 {
     %c = llvm.constant 5 : i32
     llvm.return %c
}
```

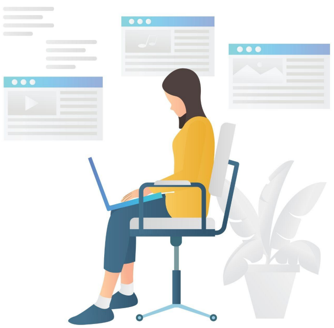
MLIR is really nice!

MLIR is really nice!

…but it's very complicated

UNIVERSITY OF CAMBRIDGE

Research



Open-Source Development

Teaching
(150 students / year)

Core Concepts

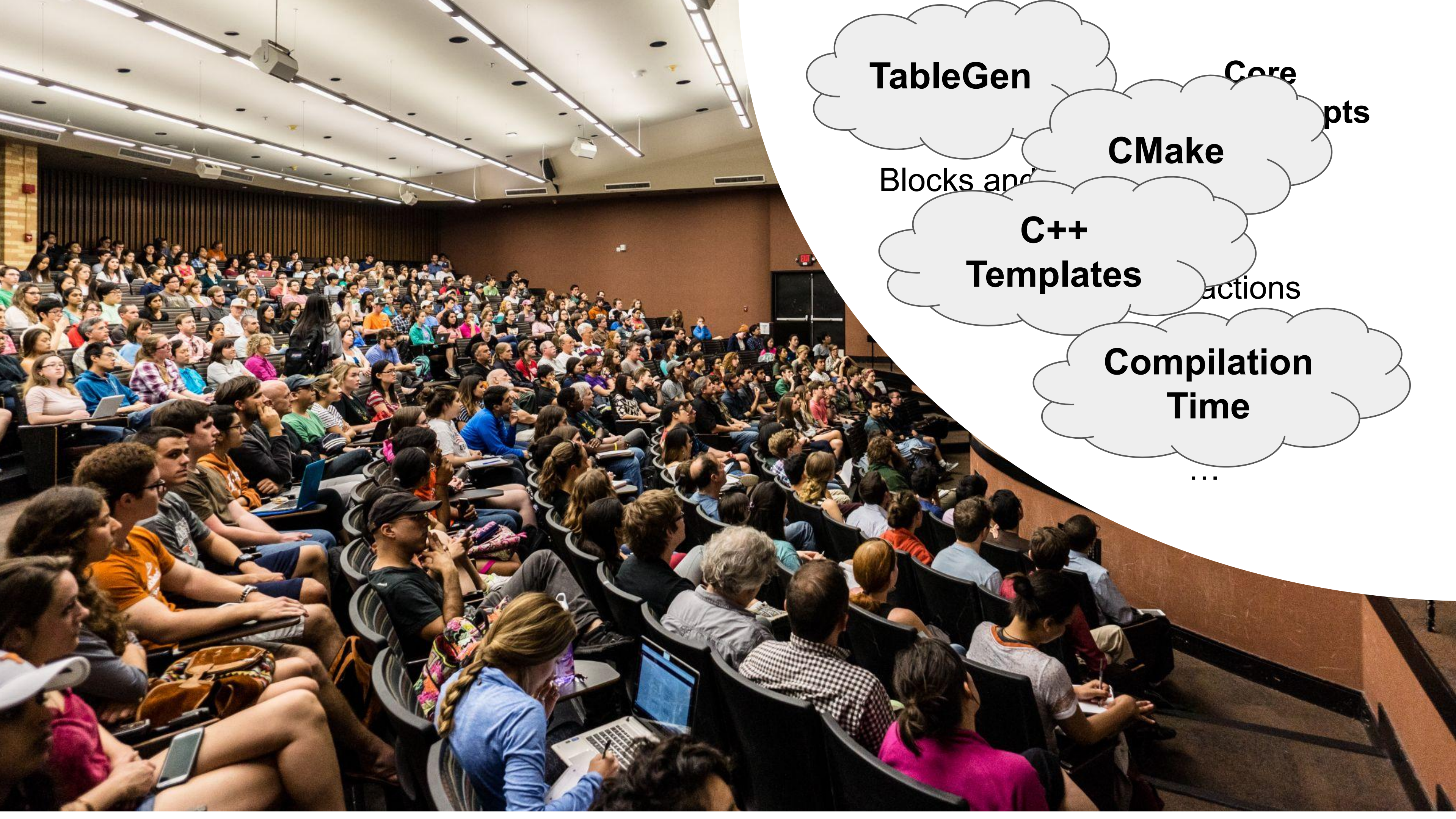
SSA and Operations

Blocks and Regions

Dialects as abstractions

Peephole rewrites

…



TableGen
Core
pts
CMake
Blocks an
C++
Templates
actions
Compilation
Time
...

How to teach MLIR?

TableGen

Core

pts

CMake

Blocks and

C++
Templates

actions

Compilation
Time

...

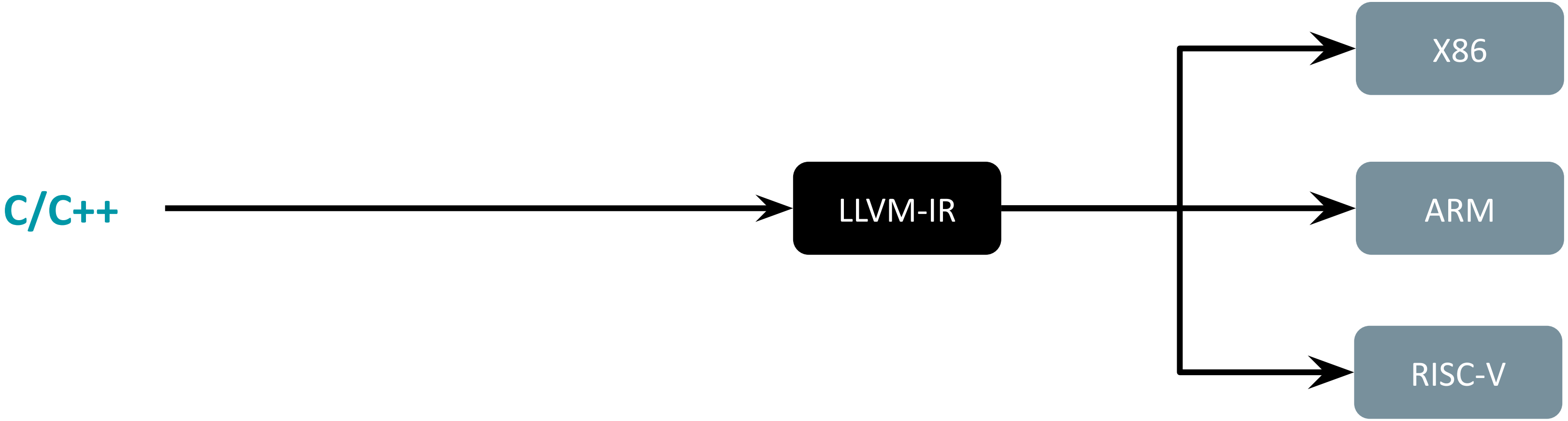
MLIR is hard to get started with!

Bringing in the MLIR ecosystem

xDSL is a Python implementation of the MLIR concepts

- Focus on *approachability*
- *Reuse* of existing concepts implemented in a simpler way
- *Expands* on MLIR concepts
- Making compiler frameworks *interoperable*

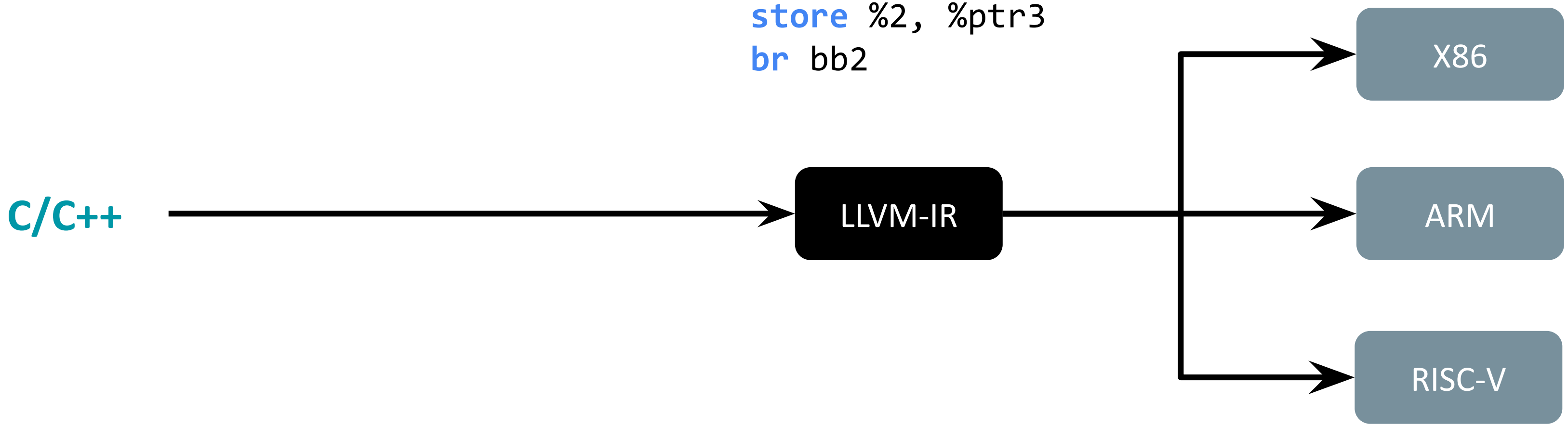
Compiler Pipelines



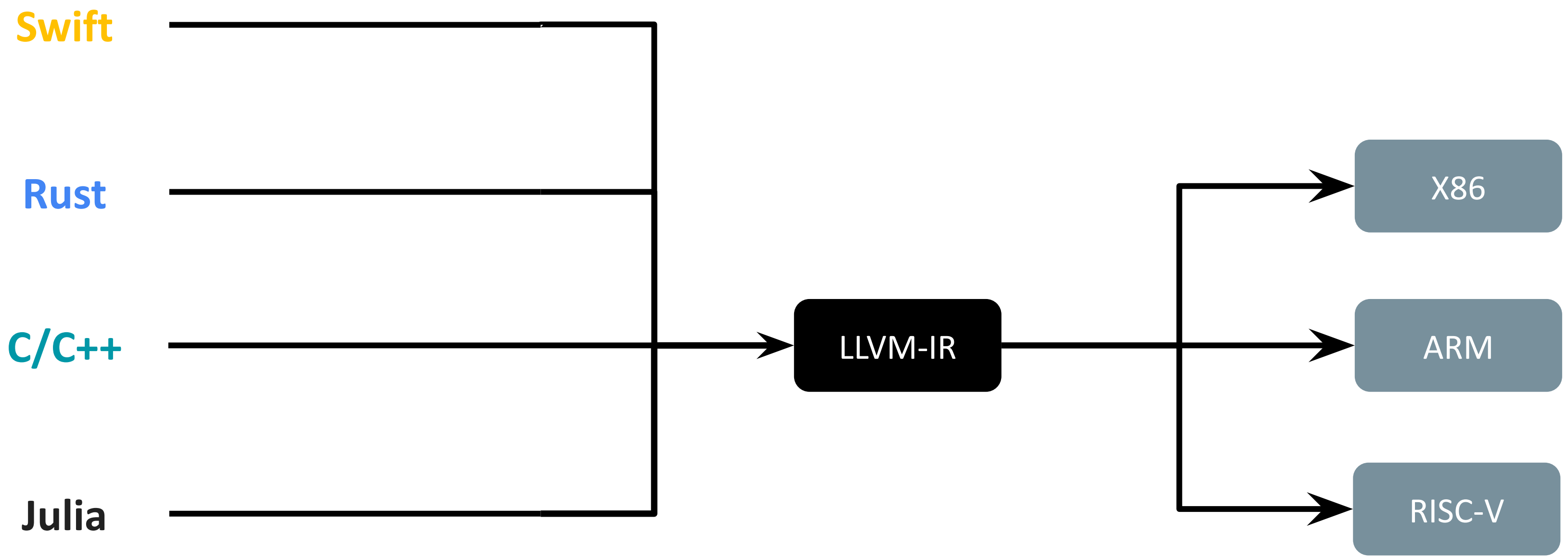
Compiler Pipelines

Intermediate Representations (IRs) are the central abstractions in a compiler.

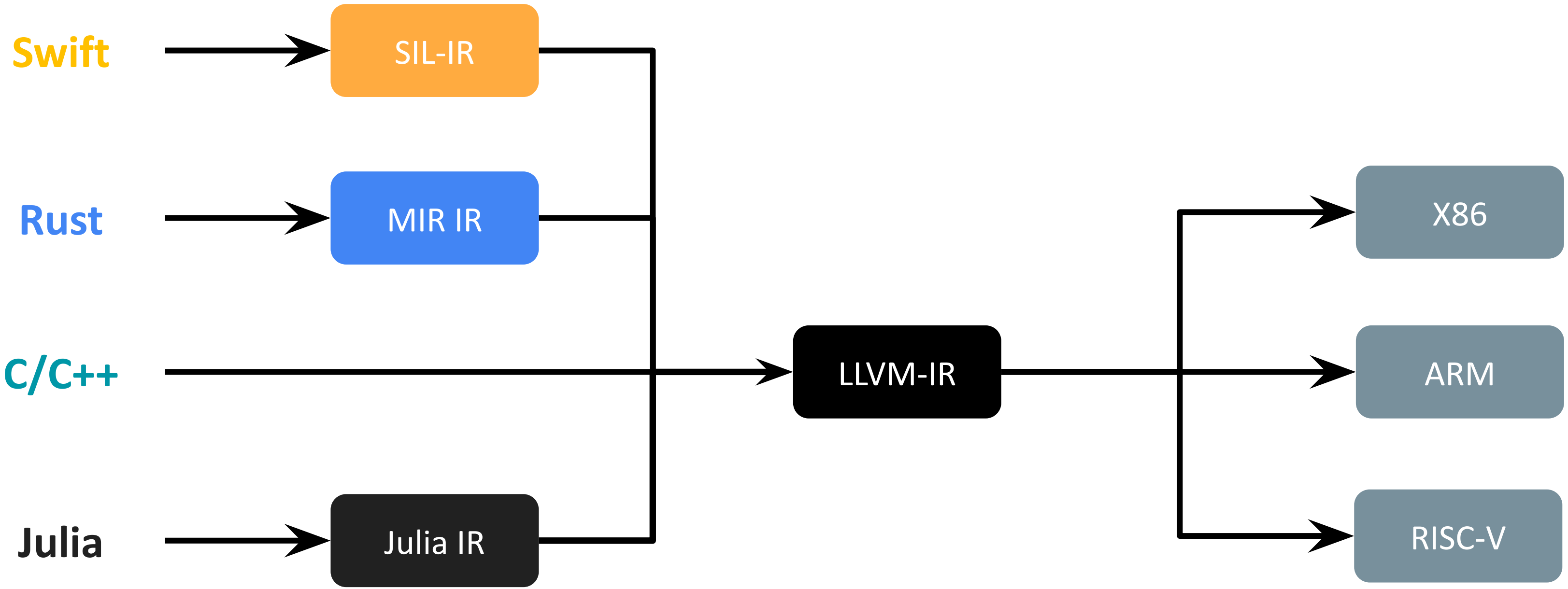
```
%0 = load %ptr
%1 = load %ptr2
%2 = add %0, %1
store %2, %ptr3
br bb2
```



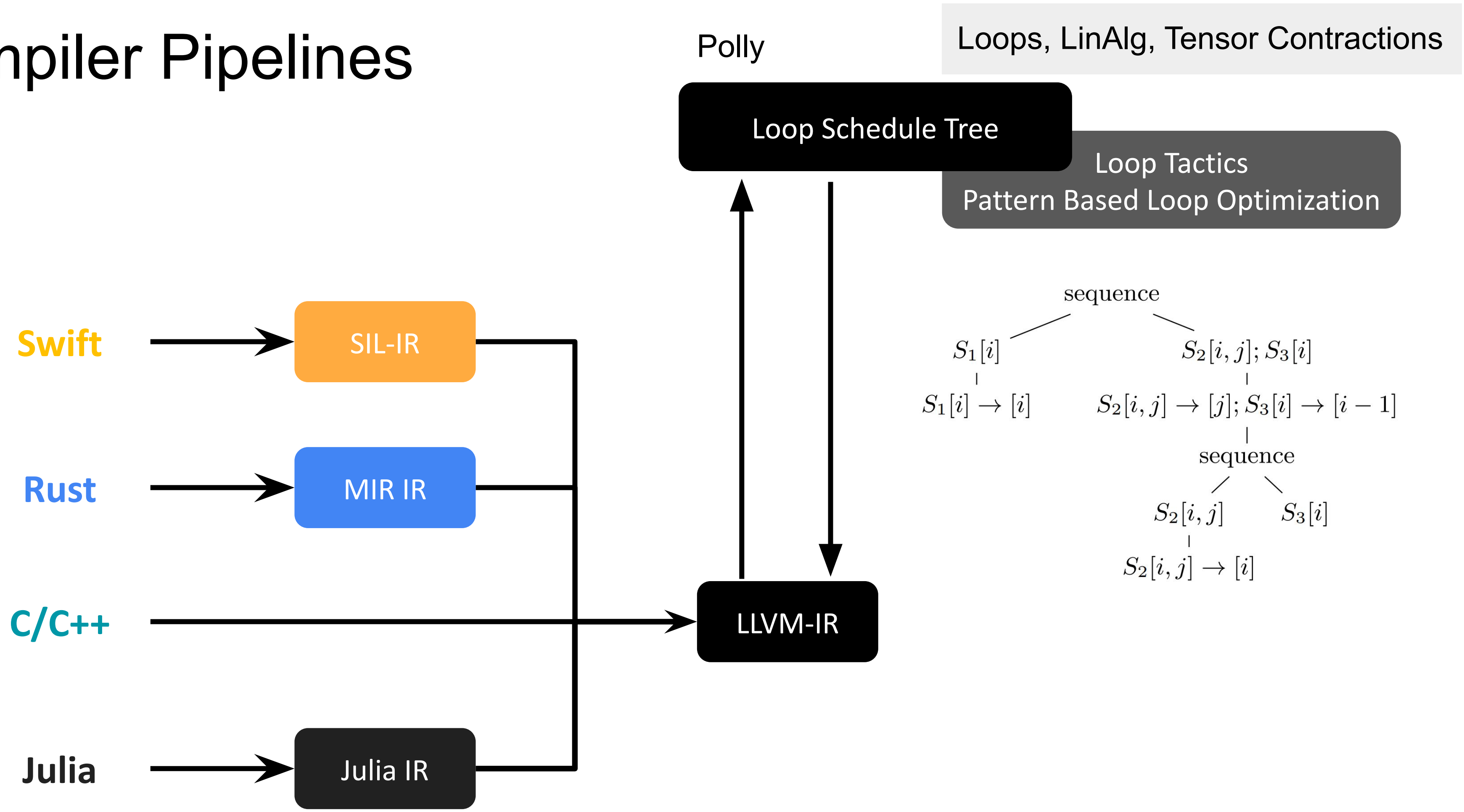
Compiler Pipelines



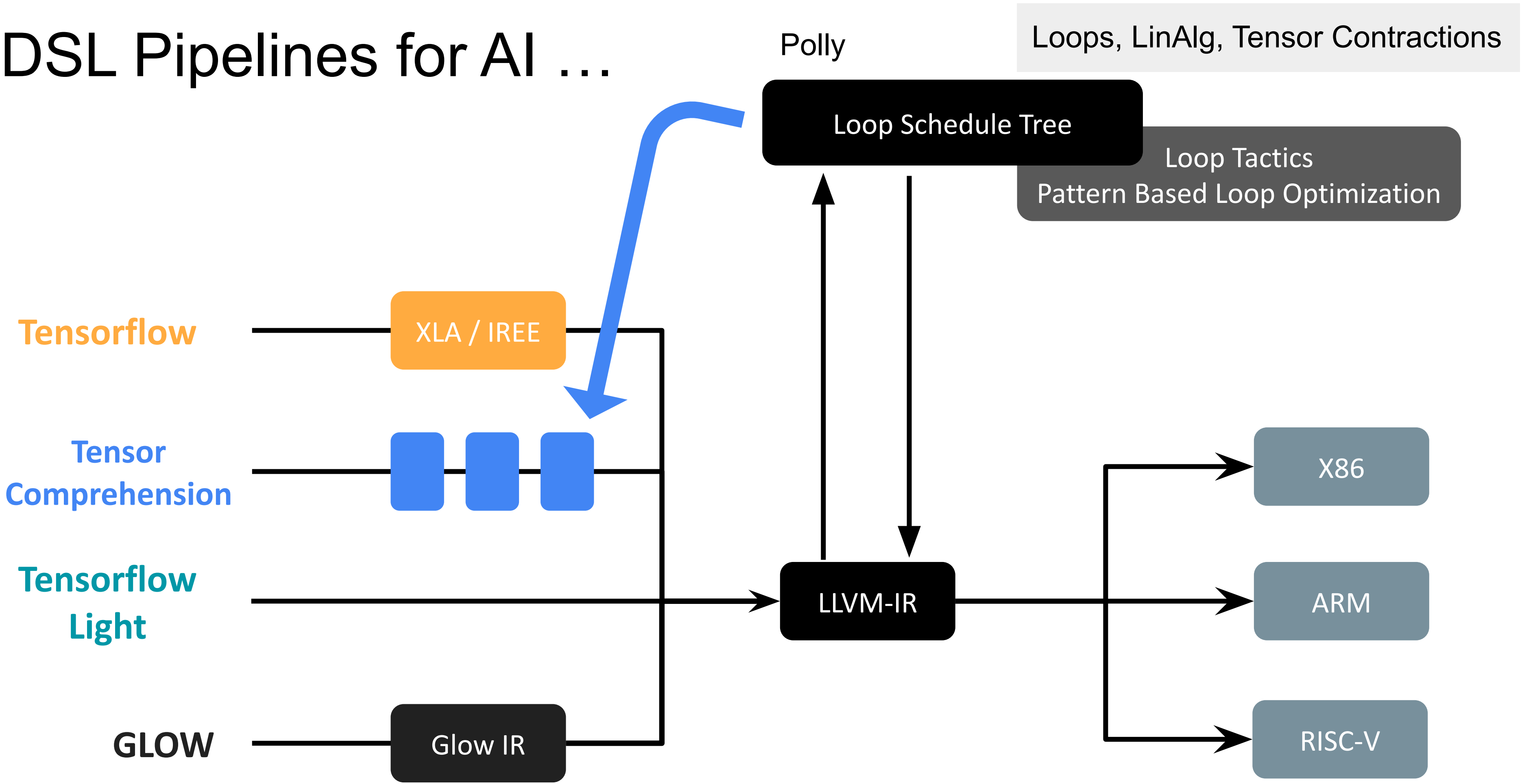
Compiler Pipelines



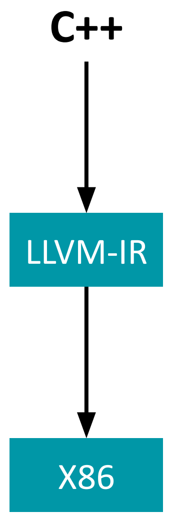
Compiler Pipelines



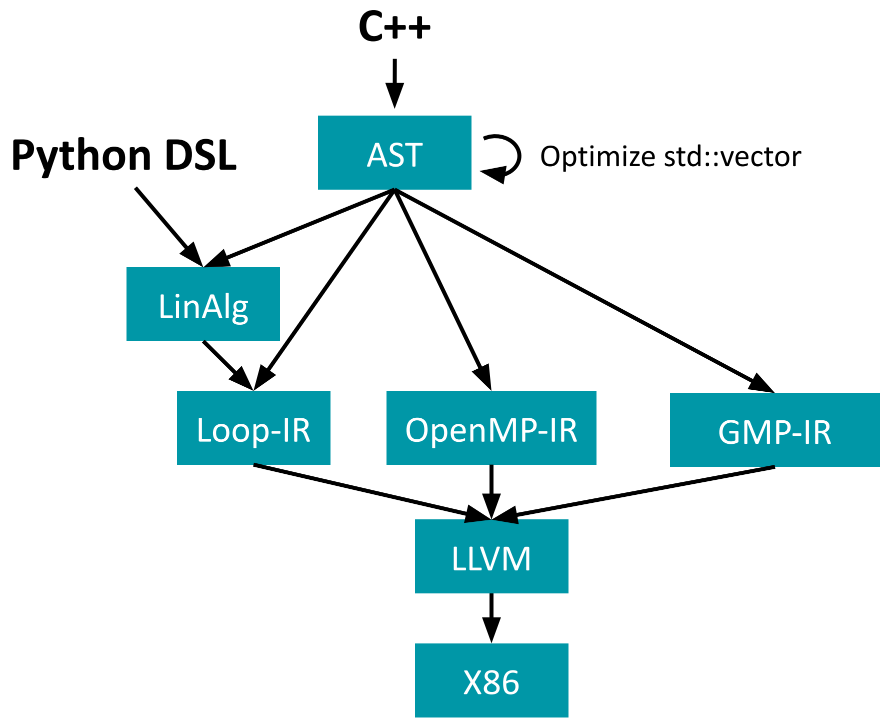
DSL Pipelines for AI …



Traditional Compilation



New: Multi-Level Rewriting



MLIR — Multi-Level Intermediate Representation

Example: Matrix Multiplication in MLIR

```
func @matmul_square(%A: memref<?x?xf32>,
                    %B: memref<?x?xf32>,
                    %C: memref<?x?xf32>) {
  %n = dim %A, 0 : memref<?x?xf32>

  affine.for %i = 0 to %n {
    affine.for %j = 0 to %n {
      store 0, %C[%i, %j]        : memref<?x?xf32>
      affine.for %k = 0 to %n {
        %a    = load %A[%i, %k] : memref<?x?xf32>
        %b    = load %B[%k, %j] : memref<?x?xf32>
        %prod = mulf %a, %b     : f32
        %c    = load %C[%i, %j] : memref<?x?xf32>
        %sum  = addf %c, %prod  : f32
        store %sum, %C[%i, %j]  : memref<?x?xf32>
      }
    }
  }
  return
}
```

Attributes
represent additional static information

Operations
represent computations

Regions & Blocks
allow sequencing and nesting of operations

Types
ensure consistency of the overall program

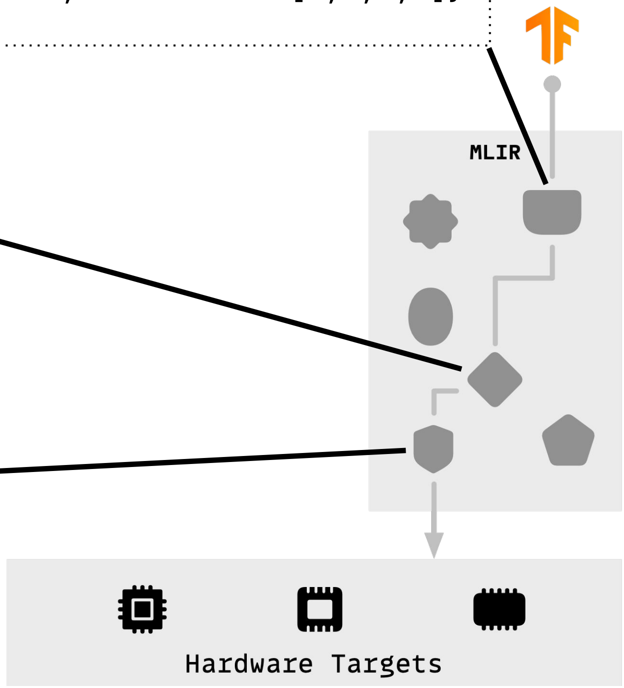
MLIR — Multi-Level Intermediate Representation

Progressive Lowering from Application Domain to Hardware

```
%x = tf.Conv2d(%input, %filter) {strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}
      : (tensor<*xf32>, tensor<*xf32>) → tensor<*xf32>
```

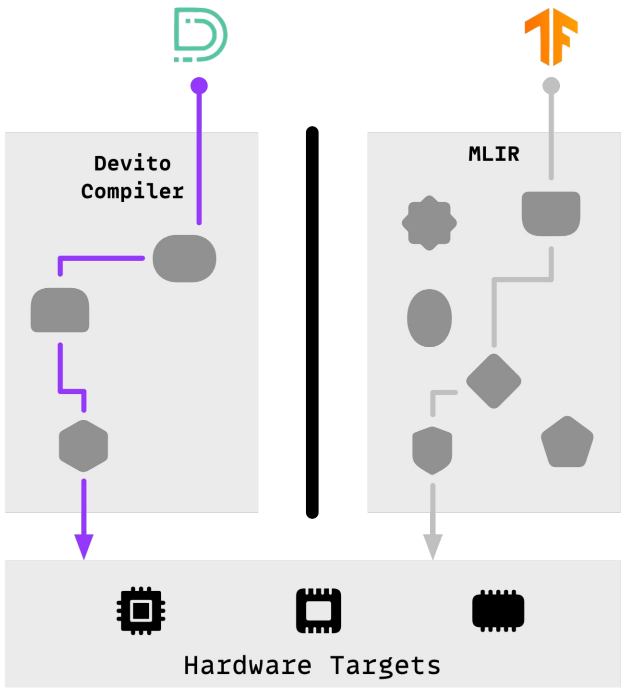
```
affine.for %i = 0 to %n {
  …
  %sum  = addf %a, %b : f32
  …
}
```

```
gpu.launch(%gx,%gy,%c1,%lx,%c1,%c1) {
  ^bb0(%bx: index, %by: index, %bz: index,
       %tx: index, %ty: index, %tz: index,
       %num_bx: index, %num_by: index, %num_bz: index,
       %num_tx: index, %num_ty: index, %num_tz: index)
  …
  %sum  = addf %a, %b : f32
  …
}
```



Problem: Isolated Compiler Ecosystems

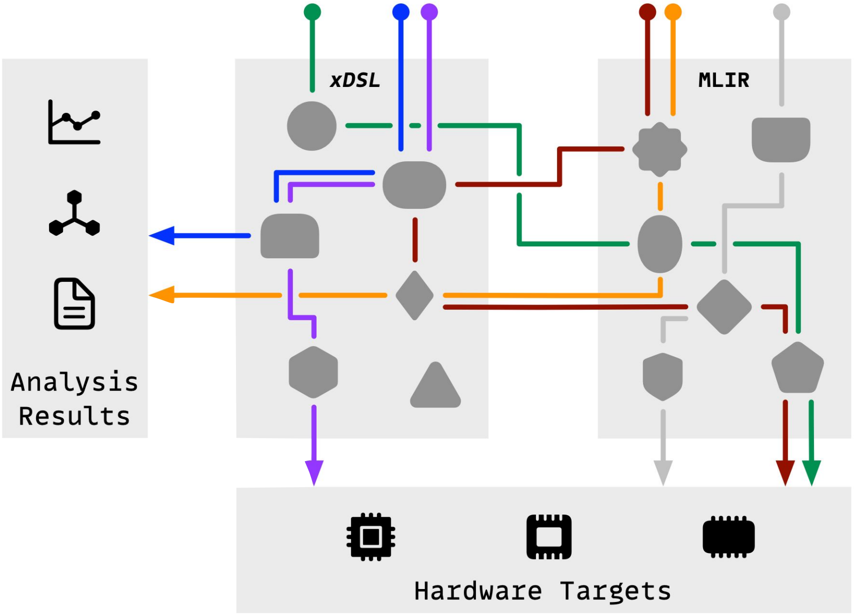
Each DSL reimplements the same IRs and optimizations



https://github.com/xdslproject/xdsl/

xDSL: a ***Sidekick*** to MLIR

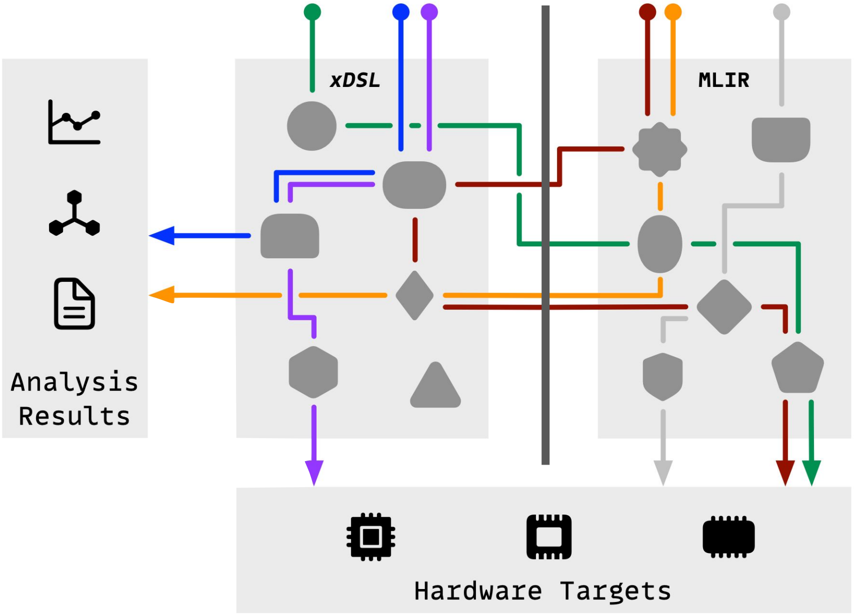
Making the MLIR ecosystem accessible and extensible from Python



- Python-native end-to-end compilers
- Prototyping new compiler design ideas
- Analysing the compilation flow
- Pairing high-level Python DSLs with existing low-level MLIR dialects and optimizations

xDSL: a *Sidekick* to MLIR

How does it work?



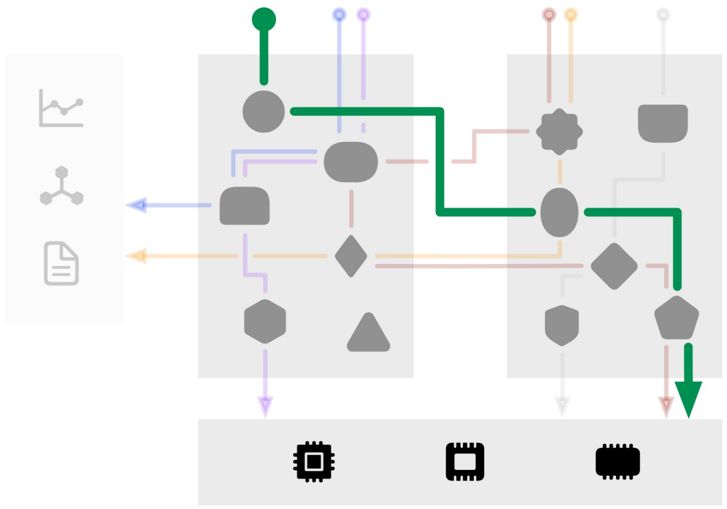
Translation consists of:

- Programs ✔
- IR Definitions ✔
- Transformations ?

Use Case

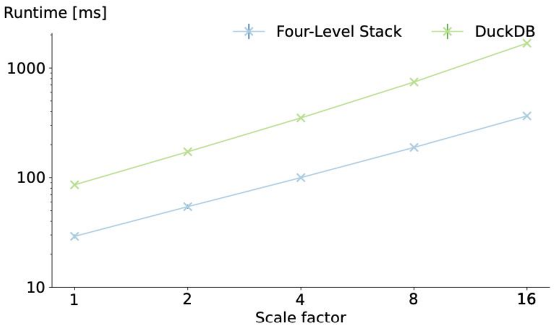
Building a high-level Python DSL with existing low-level MLIR dialects

- *User:*
  - Domain experts, e.g., computational scientists or database experts
- *Needs:*
  - Productivity is (often) more important than compilation speed
- *Existing Workflows:*
  - Build isolated compiler ecosystem (such as Devito)
- *The xDSL Approach:*
  - Embed high-level DSL in Python for ease of use
  - Use xDSL dialects in Python and then lower to common dialects that are optimized in MLIR

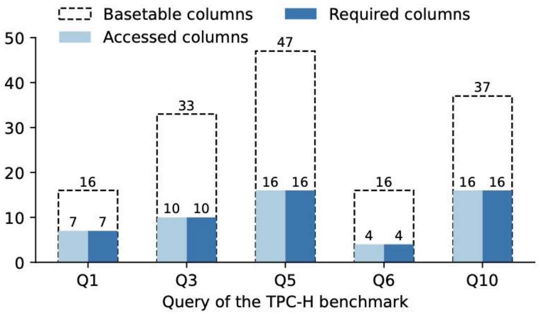


Use Case

Building a high-level Python DSL with existing low-level MLIR dialects



We implemented a database DSL using xDSL outperforming the in-memory database DuckDB



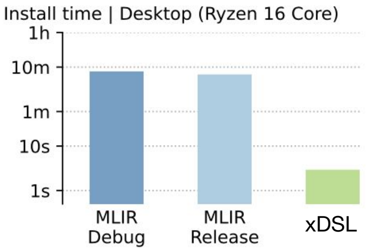
Reduction of basetable column accesses implemented as a compiler optimization pass in Python with xDSL

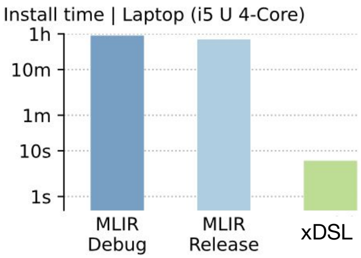
\+

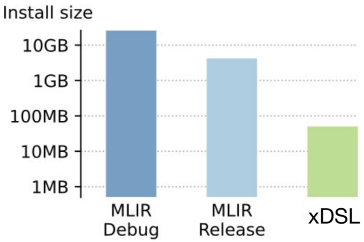
We currently work with colleagues from Imperial to integrate Devito & MLIR with xDSL

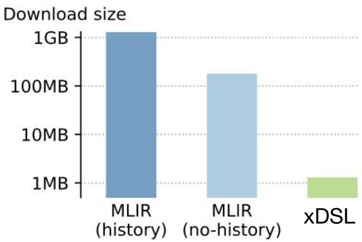
xDSL boosts Developers Productivity

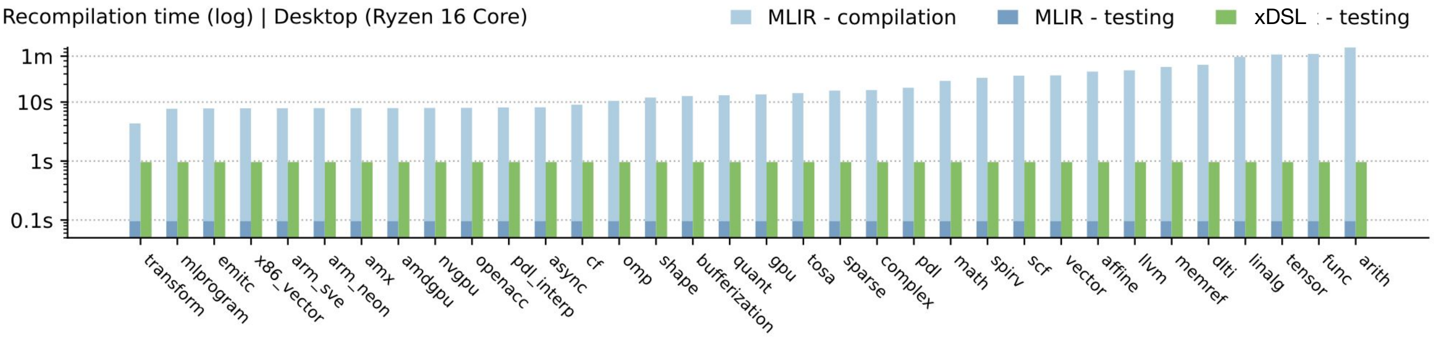
Much shorter install times | Much faster recompilation times











An MPI Abstraction for MLIR

Anton Lydike, Nick Brown, Jeff Hammond

Scope and Goals

```
char message[20];
int myrank;
```

```
myrank = 0

        message =  "Hello, there"

        MPI_Send message to rank 1
```

```
myrank = 1

        MPI_Recv into message from rank 0

        message =  "Hello, there"
```

Scope and Goals

```c
char message[20];
int myrank;

MPI_Init(NULL, NULL);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) /* code for process zero */
{
    strcpy(message, "Hello, there");
    MPI_Send(message, strlen(message) + 1, MPI_CHAR,
             1, 0, MPI_COMM_WORLD);
}
else if (myrank == 1) /* code for process one */
{
    MPI_Recv(message, 20, MPI_CHAR, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("received :%s\n", message);
}

MPI_Finalize();
```

Scope and Goals

```c
char message[20];
int myrank;

MPI_Init(NULL, NULL);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) /* code for process zero */
{
    strcpy(message, "Hello, there");
    MPI_Send(message, strlen(message) + 1, MPI_CHAR,
             1, 0, MPI_COMM_WORLD);
}
else if (myrank == 1) /* code for process one */
{
    MPI_Recv(message, 20, MPI_CHAR, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("received :%s\n", message);
}

MPI_Finalize();
```

```mlir
mpi.init

%zero = arith.constant 0 : i32
%one = arith.constant 1 : i32

%message = memref.alloca() : memref<12xi8>
%data = memref.get_global @hello_there : memref<12xi8>

%myrank = mpi.comm_rank : i32
%is_rank_zero = arith.cmpi eq, %myrank, %zero : i32
scf.if %is_rank_zero
{ // code for process zero
  memref.copy %data, %message : memref<12xi8> to memref<12xi8>
  mpi.send(%message, %one, %zero) : (memref<12xi8>, i32, i32)

} else { // code for process one
  mpi.recv(%message, %zero, %zero) : (memref<12xi8>, i32, i32)
  printf.print_format "received: {}\n" %message : memref<12xi8>

}

mpi.finalize
```

MPI_Init
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Finalize

Modelling MPI

Modelling MPI

MPI_Init ✓
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Finalize

```
MPI_Init(NULL, NULL);
```

```
mpi.init() : () -> ()
```

- Simplify default constant arguments

MPI_Init ✓
MPI_Comm_rank ✓
MPI_Send
MPI_Recv
MPI_Finalize

Modelling MPI

```
MPI_Init(NULL, NULL);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
mpi.init() : () -> ()

%myrank = mpi.comm_rank() : () -> i32
```

- Simplify default constant arguments
- Out argument becomes SSA result

Modelling MPI

MPI_Init ✓
MPI_Comm_rank ✓
MPI_Send ✓
MPI_Recv
MPI_Finalize

```
MPI_Init(NULL, NULL);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(message, strlen(message) + 1, MPI_CHAR,
         1, 0, MPI_COMM_WORLD);
```

```
mpi.init() : () -> ()

%myrank = mpi.comm_rank() : () -> i32

mpi.send(%message, %one, %zero)
  : (memref<12xi8>, i32, i32) -> ()
```

- Simplify default constant arguments
- Out argument becomes SSA result
- Pointer + Size + Datatype = memref

Modelling MPI

MPI_Init ✔
MPI_Comm_rank ✔
MPI_Send ✔
MPI_Recv ✔
MPI_Finalize

```c
MPI_Init(NULL, NULL);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(message, strlen(message) + 1, MPI_CHAR,
         1, 0, MPI_COMM_WORLD);

MPI_Recv(message, 20, MPI_CHAR, 0, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
mpi.init() : () -> ()

%myrank = mpi.comm_rank() : () -> i32

mpi.send(%message, %one, %zero)
  : (memref<12xi8>, i32, i32) -> ()

mpi.recv(%message, %zero, %zero)
  : (memref<12xi8>, i32, i32) -> ()
```

- Simplify default constant arguments
- Out argument becomes SSA result
- Pointer + Size + Datatype = memref

Modelling MPI

MPI_Init ✔
MPI_Comm_rank ✔
MPI_Send ✔
MPI_Recv ✔
MPI_Finalize ✔

```
MPI_Init(NULL, NULL);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(message, strlen(message) + 1, MPI_CHAR,
         1, 0, MPI_COMM_WORLD);

MPI_Recv(message, 20, MPI_CHAR, 0, 0,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Finalize();
```

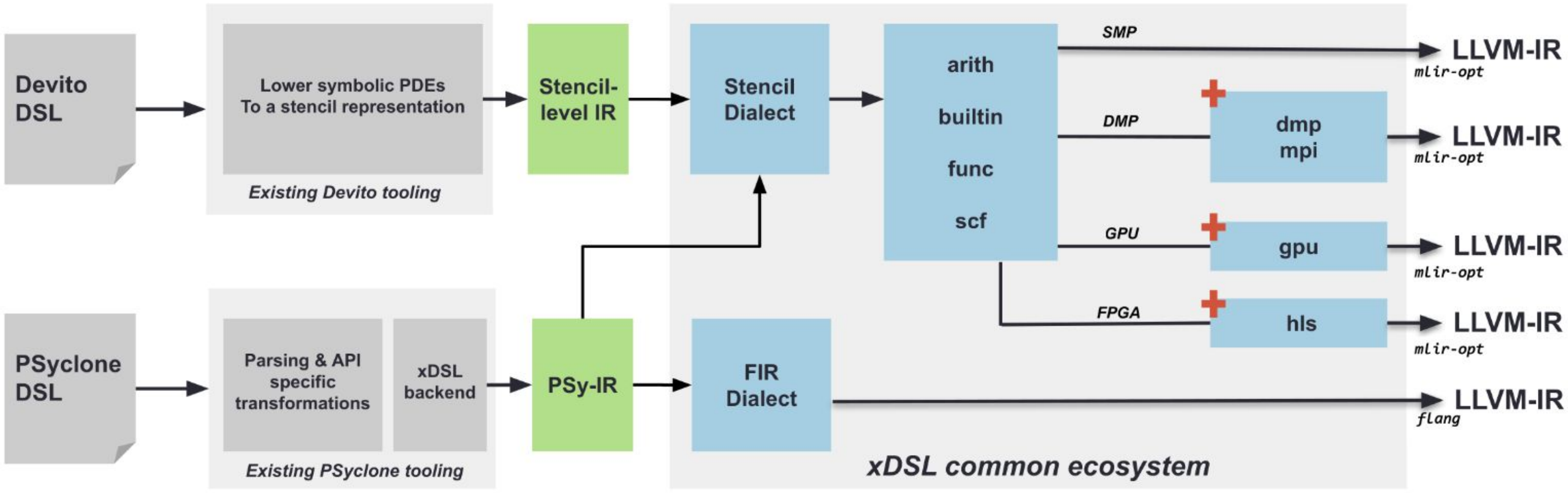
```
mpi.init() : () -> ()

%myrank = mpi.comm_rank() : () -> i32

mpi.send(%message, %one, %zero)
  : (memref<12xi8>, i32, i32) -> ()

mpi.recv(%message, %zero, %zero)
  : (memref<12xi8>, i32, i32) -> ()

mpi.finalize() : () -> ()
```

- Simplify default constant arguments
- Out argument becomes SSA result
- Pointer + Size + Datatype = memref

Devito and PSYCLONE on xDSL/MLIR

Cambridge, Imperial & EPCC

A Joint MLIR-Based Compilation Pipeline



From Stencils to Distributed MPI

Stencil level IR → Global to Local → DMP level IR → DMP to MPI → MPI level IR

Stencil level IR

```
%source = stencil.load(%114) : (!field<[0,128]xf64>)
                          -> !temp<?xf64>
%out = stencil.apply(%arg = %source : !temp<?xf64>)
                  -> !temp<?xf64> {
  %l = stencil.access %arg[-1] : f64
  %c = stencil.access %arg[0] : f64
  %r = stencil.access %arg[1] : f64
  // %v = %l + %r - 2.0 * %c
  stencil.return %v : f64
}

stencil.store %out to %target([1]:[127])
```

Global Domain

DMP level IR

```
%ref = builtin.unrealized_conversion_cast %114 :
    !field<[0,64]xf64> to memref<64xf62>
dmp.swap(%ref) {
  "grid" = #dmp.grid<2>,
  "swaps" = [
    #dmp.exchange<at [0] size [1]
                  source offset [1] to [-1]>,
    #dmp.exchange<at [64] size [1]
                  source offset [-1] to [1]>
  ]
} : (memref<64xf64>) -> ()
%source = stencil.load(%114) ...
%out = stencil.apply(%source) ...
stencil.store %out to %target([1]:[64])
```

Exchanged area

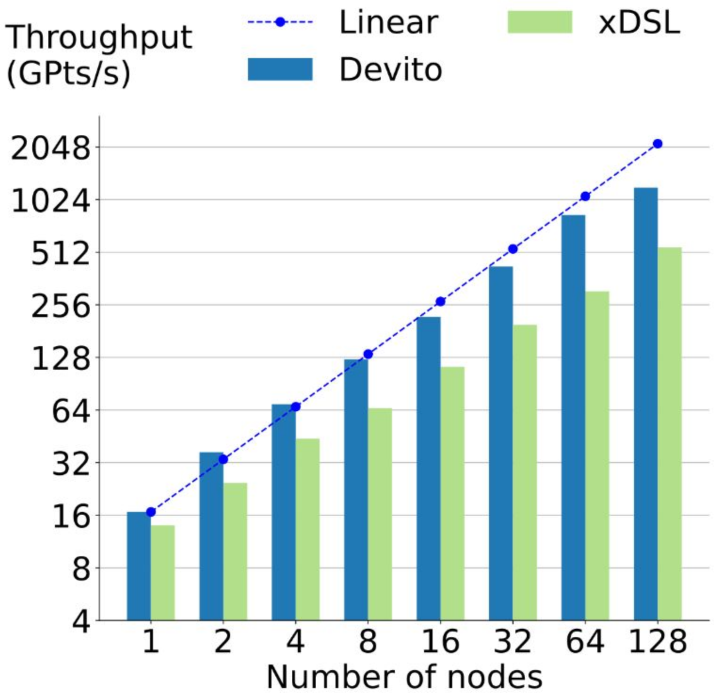
Local domain

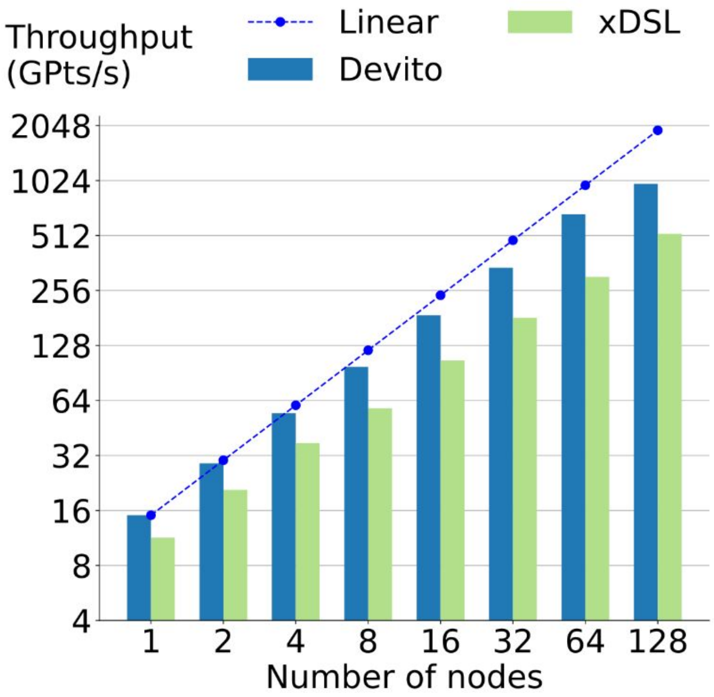
Local Domains with halo exchanges highlighted

MPI level IR

```
%rank = mpi.comm_rank : i32
// First swap communication calls
%dest = arith.add %rank, %minus_one : i32
%is_in_bounds = arith.cmpi sge, %dest, %zero
scf.if %is_in_bounds {
  %view = memref.subview %ref[0][1][1] : memref<64xf64>
                                         to memref<1xf64>
  // copy data into send buffer and set up communication
  // (omitted for clarity)
  mpi.isend %sptr, %count, %dtype, %dest, %tag, %send_req
  mpi.irecv %rptr, %count, %dtype, %dest, %tag, %recv_req
}
// Second swap
// ...
mpi.waitall %requests, %four  // synchronization barrier
// First swap copy back
scf.if %is_in_bounds {
  %view = memref.subview %ref[1][1][1] : memref<64xf64>
                                         to memref<1xf64>
  memref.copy %recv_buffer_1 to %view
}
// Second swap copy back
// Lowered stencil comes here
```

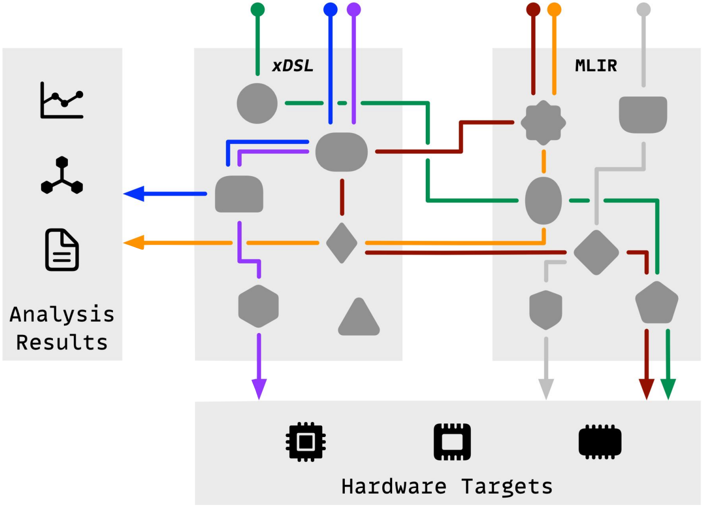
Scalable Parallelism with xDSL/MLIR





xDSL: A Compiler Infrastructure for DSLs

A framework to write domain-specific compilers



https://xdsl.dev/

https://github.com/xdslproject/xdsl/

George Bisbas, Emilien Bauer, Nick Brown, Théo Degioanni, Mathieu Fehr, Gerard Gorman, Tobias Grosser, Paul Kelly, Sasha Lopoukhine, Martin Lücke, Anton Lydike, George Mitenkov, Michel Steuwer, Larisa Stoltzfus, Christian Ulmann, Alexander Viand, Michel Weber, …