Michel Steuwer • 22 November 2022

Modern DSL Compiler Development with MLIR

or: How to design the next 700 optimizing compilers

In collaboration with:
Martin Lücke, Mathieu Fehr, Michel Weber, Christian Ulmann, Alexander Lopoukhine, Tobias Grosser

THE UNIVERSITY *of* EDINBURGH

General purpose

Software

Specialized

THE C PROGRAMMING LANGUAGE

C++

NVIDIA CUDA

TensorFlow

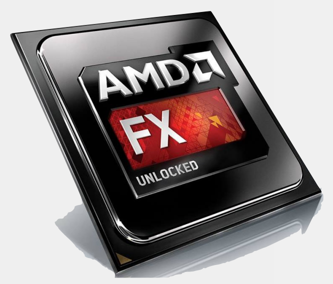
Halide

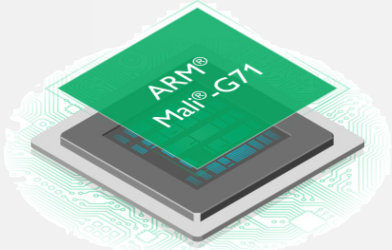
tvm

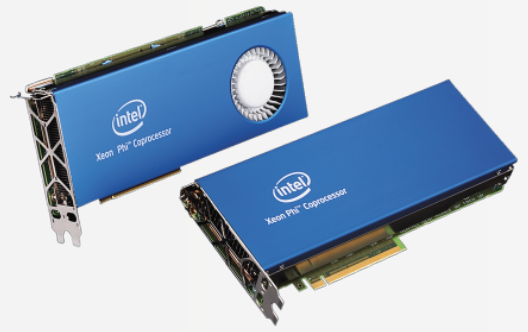
Devito

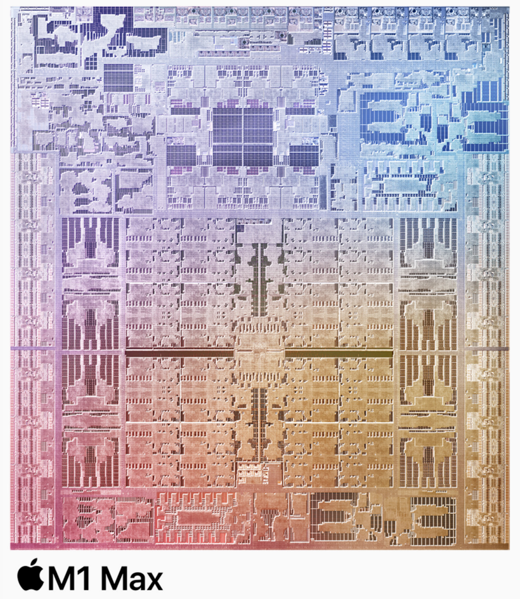
Firedrake

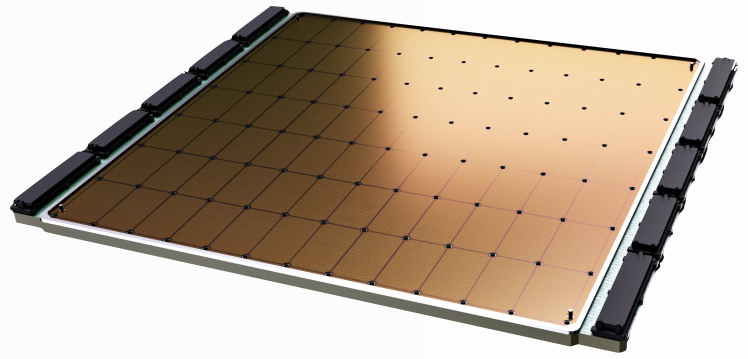
How do we build compilers to (automatically) optimise specialised software for specialized hardware?









General purpose

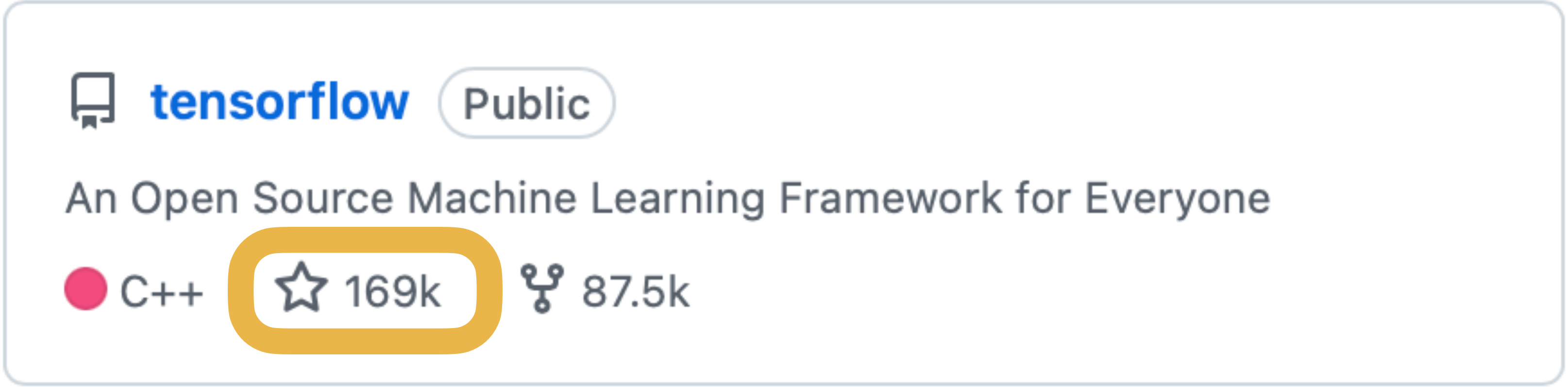
Hardware

Specialized

How Do We Currently Build Specialized Compilers?

Example 1: TensorFlow

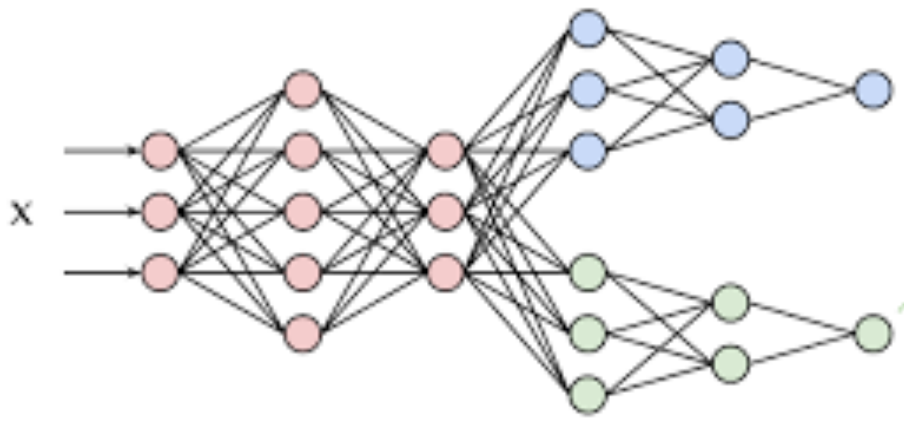
Popular machine learning framework developed by Google (and others)



TensorFlow

- ⊖ >2,500,000 lines of code
- ⊖ >500 different types of expressions represented in the TF IR
- ⊖ >50 different types of expression represented in the XLA IR
- ⊖ Compiler implemented in Python & C++ makes it hard to contribute
- ⊕ Great Performance & Support for custom hardware: TPU

XLA



Hughe effort to build and maintain, but delivering great performance

How can we benefit from the investment in ML compilers and reuse *intermediate representations & optimizations* across compilers?

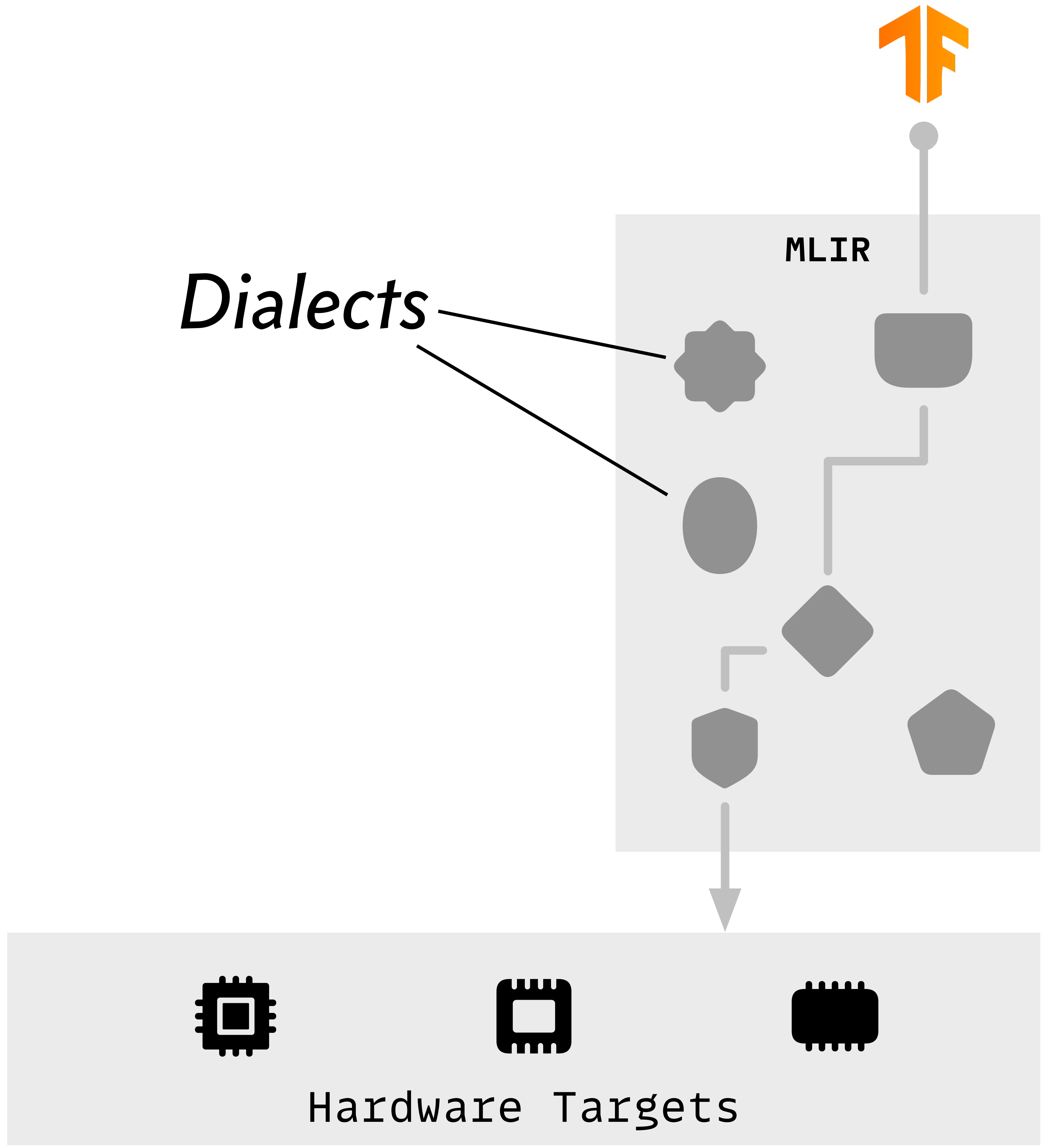
MLIR — Multi-Level Intermediate Representation

A LLVM subproject for building reusable and extensible compiler infrastructure

- MLIR is a (fairly) novel framework to facilitate the sharing of compiler intermediate representations (IRs) and optimizations

- Common abstractions are bundled in *Dialects* that can easily be combined to express programs at various levels

- Examples of dialects are:
  - `tf` - Tensor Flow abstractions
  - `affine` - Polyhedral abstractions
  - `gpu` - GPU abstractions



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
}
```

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

Types
ensure consistency of the overall program

Regions & Blocks
allow sequencing and nesting of operations

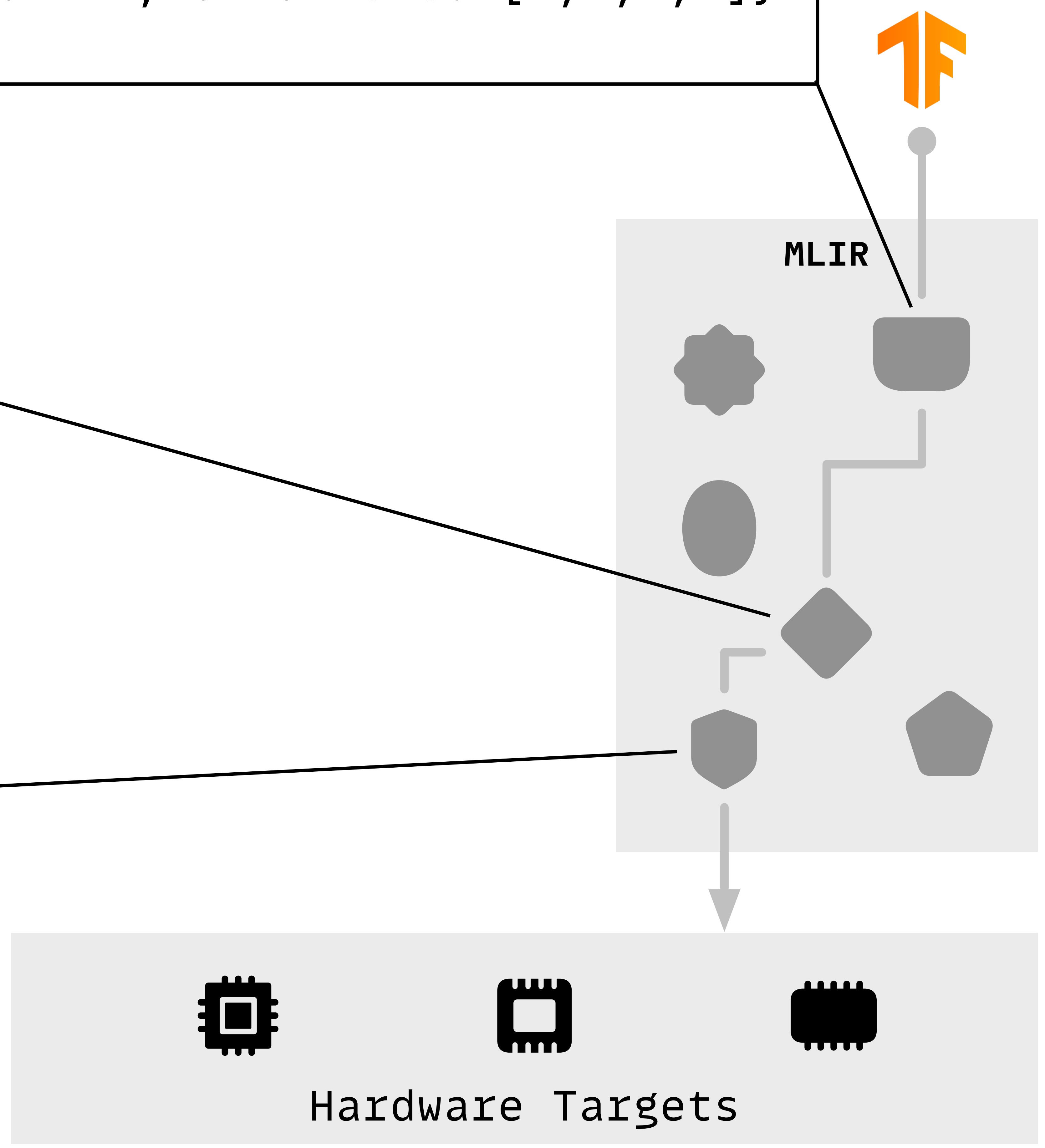
MLIR — Multi-Level Intermediate Representation

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
  …
}
```



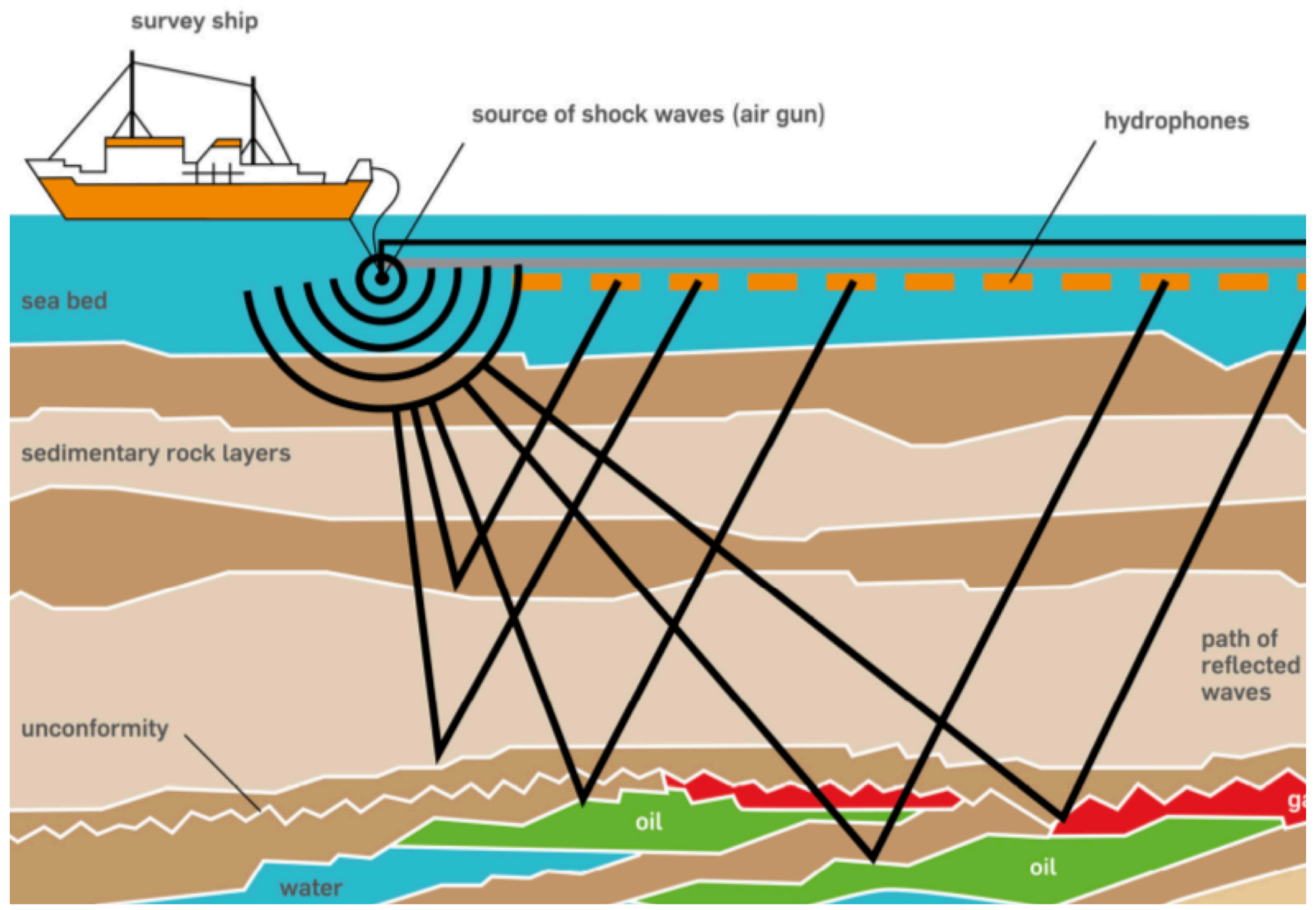
How Do We Currently Build Specialized Compilers?

Example 2: Devito

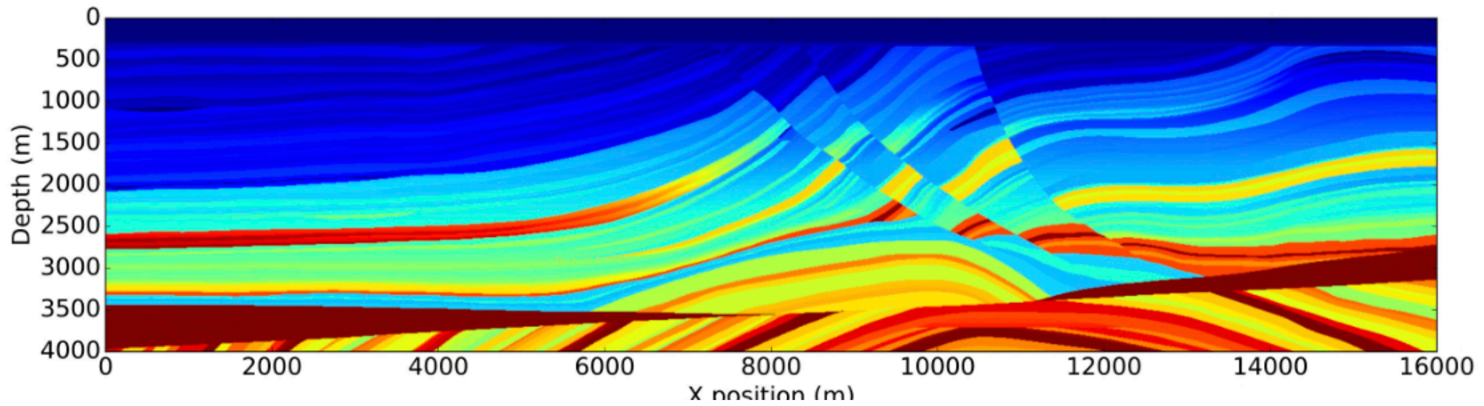
Popular HPC DSL
developed by academics **(and others)**

devito Public
Code generation framework for automated finite difference computation
Python 407 190

Devito
Imperial College London



- ⊕ < 50,000 lines of code
- ⊕ Compiler implemented in Python makes it easy to contribute
- ± Support for GPUs via OpenACC
- ⊖ Reimplementation of many classical loop optimizations
- ⊖ No support for hardware accelerators



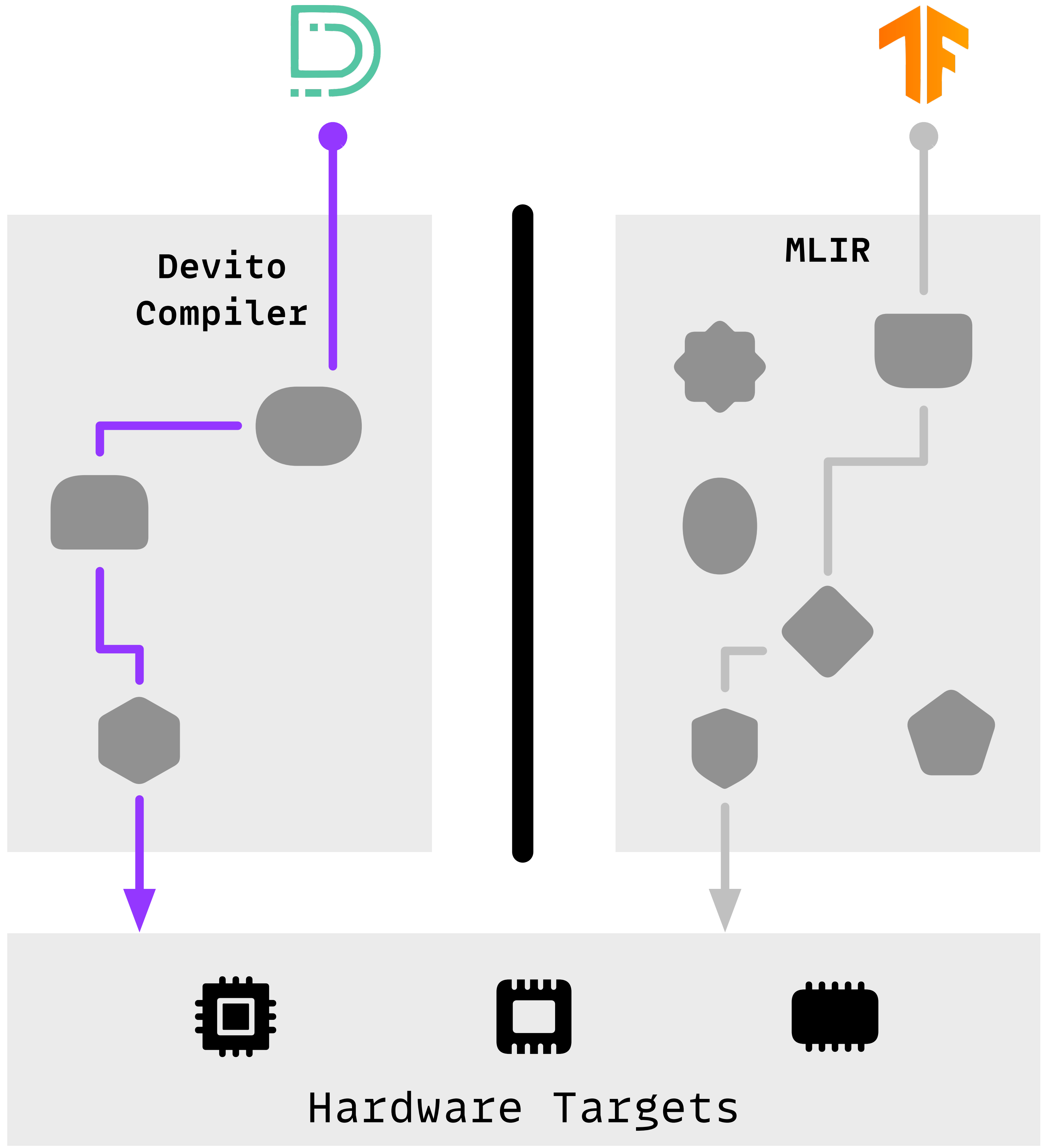
Small team delivering great usability and performance,
but limited support of advanced optimizations and hardware

Problem: Isolated Compiler Ecosystems

Each DSL reimplements the same IRs and optimizations

- Today, Devito and Tensor Flow share no code

- But, there is a huge ***opportunity*** for HPC DSLs:
    - They have some common IRs
    - They perform similar optimizations
    - They could benefit from the current investment in ML compilers



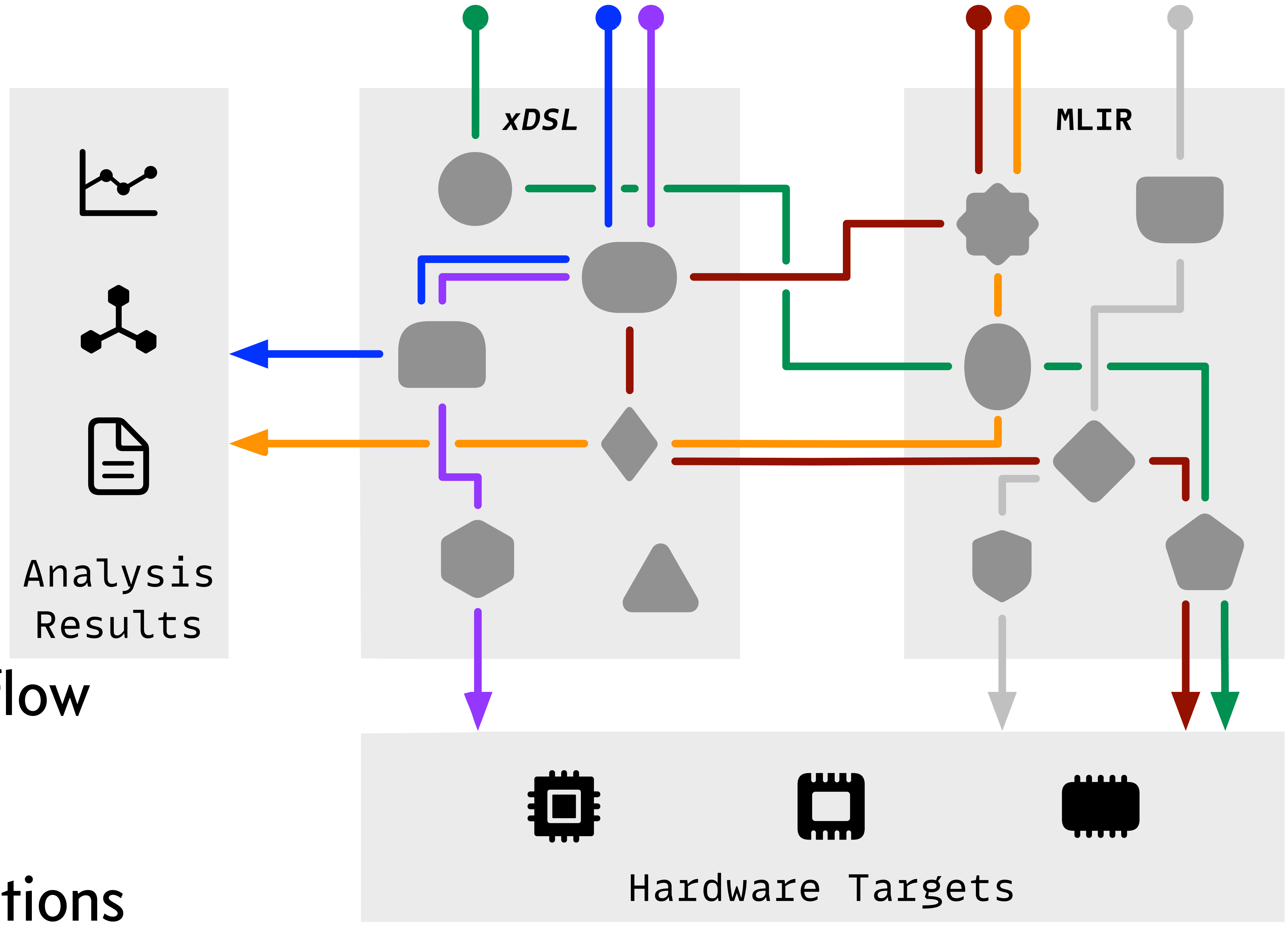
xDSL: a *Sidekick* to MLIR

Making the MLIR ecosystem accessible and extensible from Python

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

- xDSL is a Python framework we develop at the University of Edinburgh, it shares ***the same*** IR format and dialects with MLIR

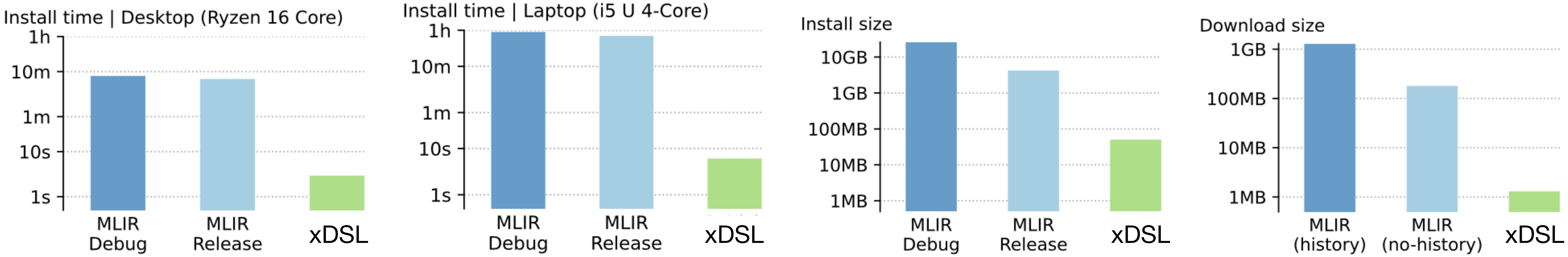
- This allows for many possible use cases:
  - Python-native end-to-end compilers
  - Prototyping new compiler design ideas
  - Building tools for analysing the compilation flow
  - Pairing high-level Python DSLs with existing low-level MLIR dialects and optimizations

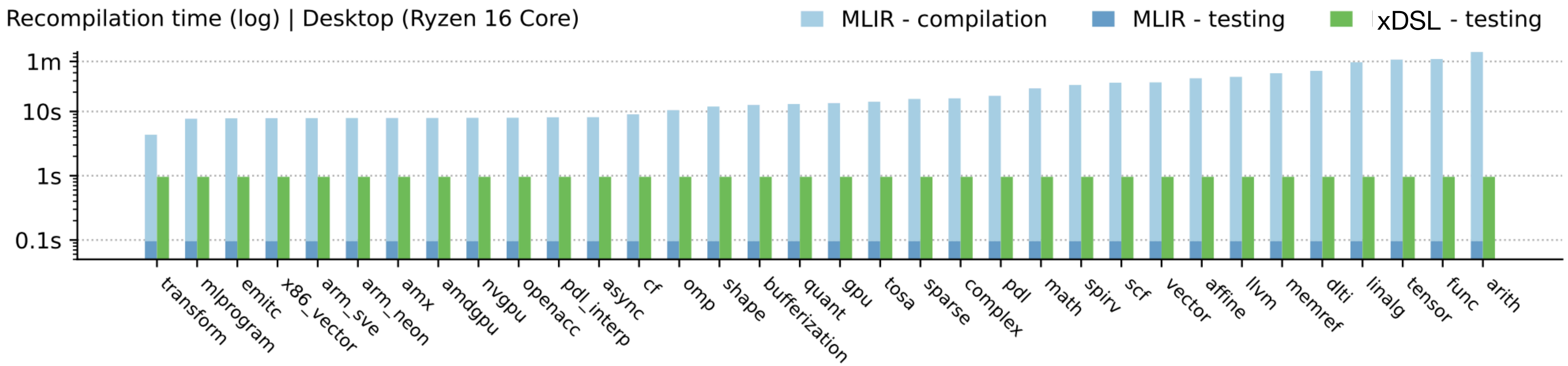


xDSL Boosts Developers Productivity

Much shorter install times | Much faster recompilation times

`pip install xdsl`

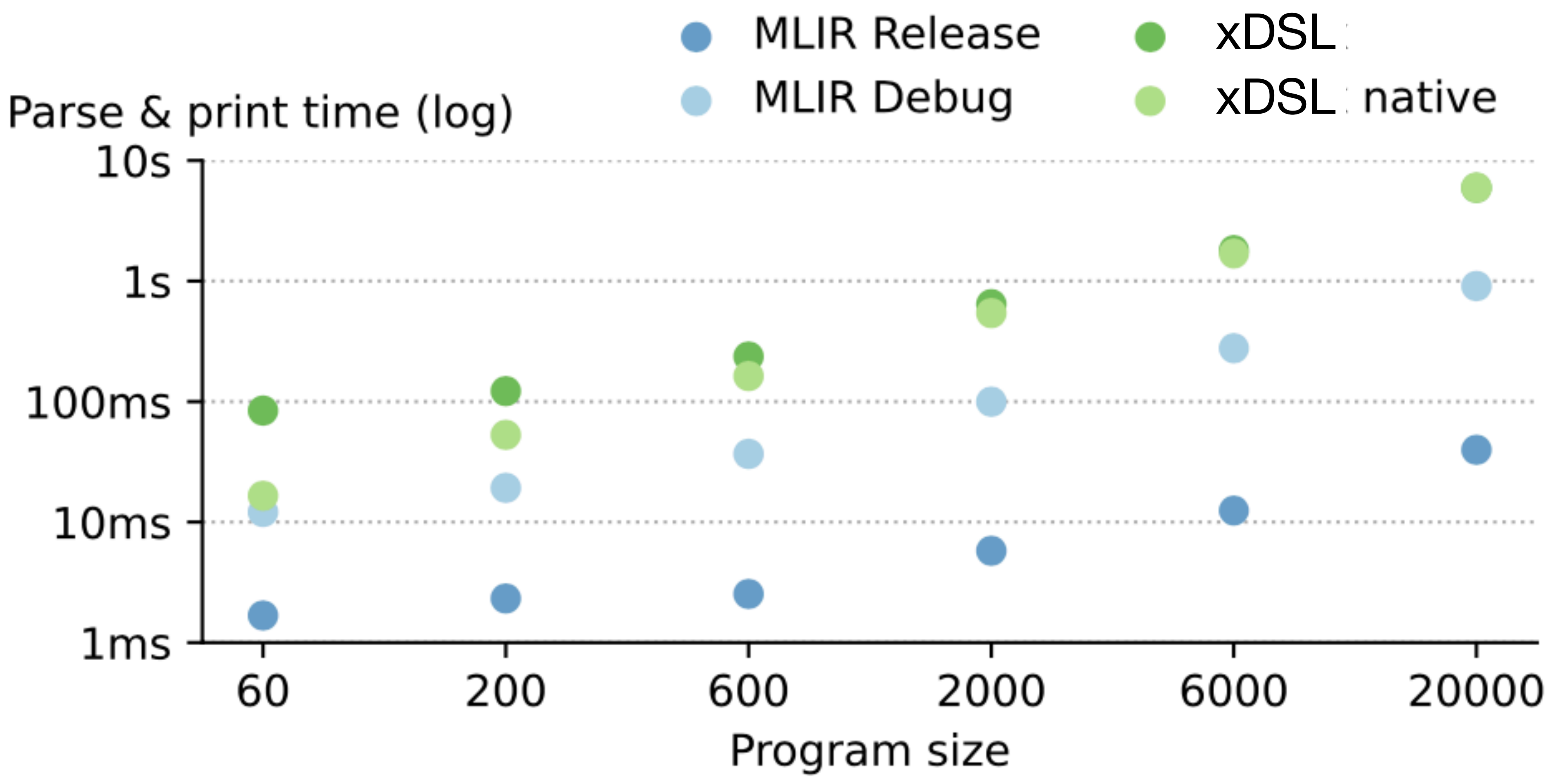


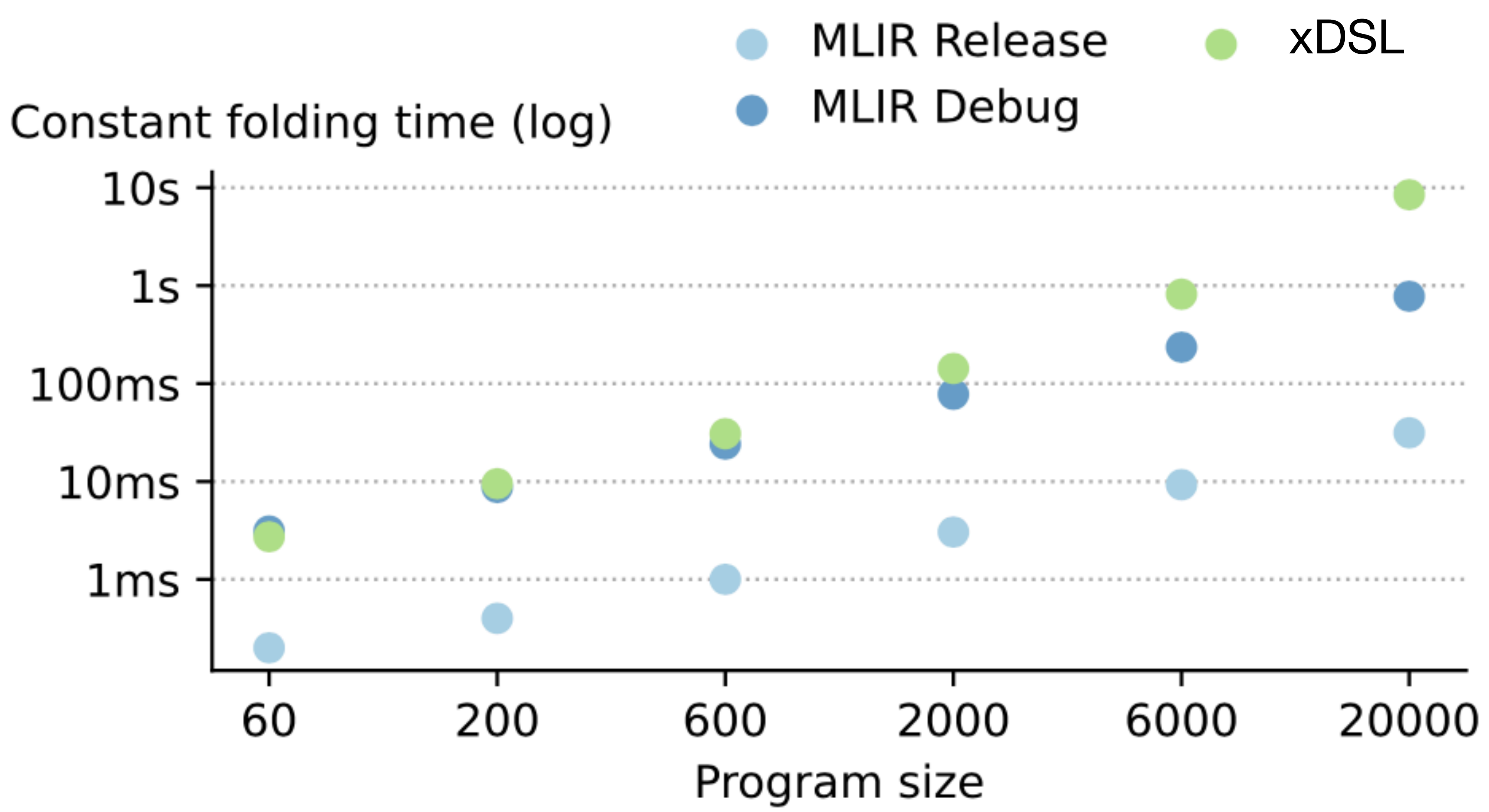


xDSL Has Reasonable Overheads Compared to MLIR

About 1 order of magnitude slower for parsing & printing

Comparable performance for constant folding

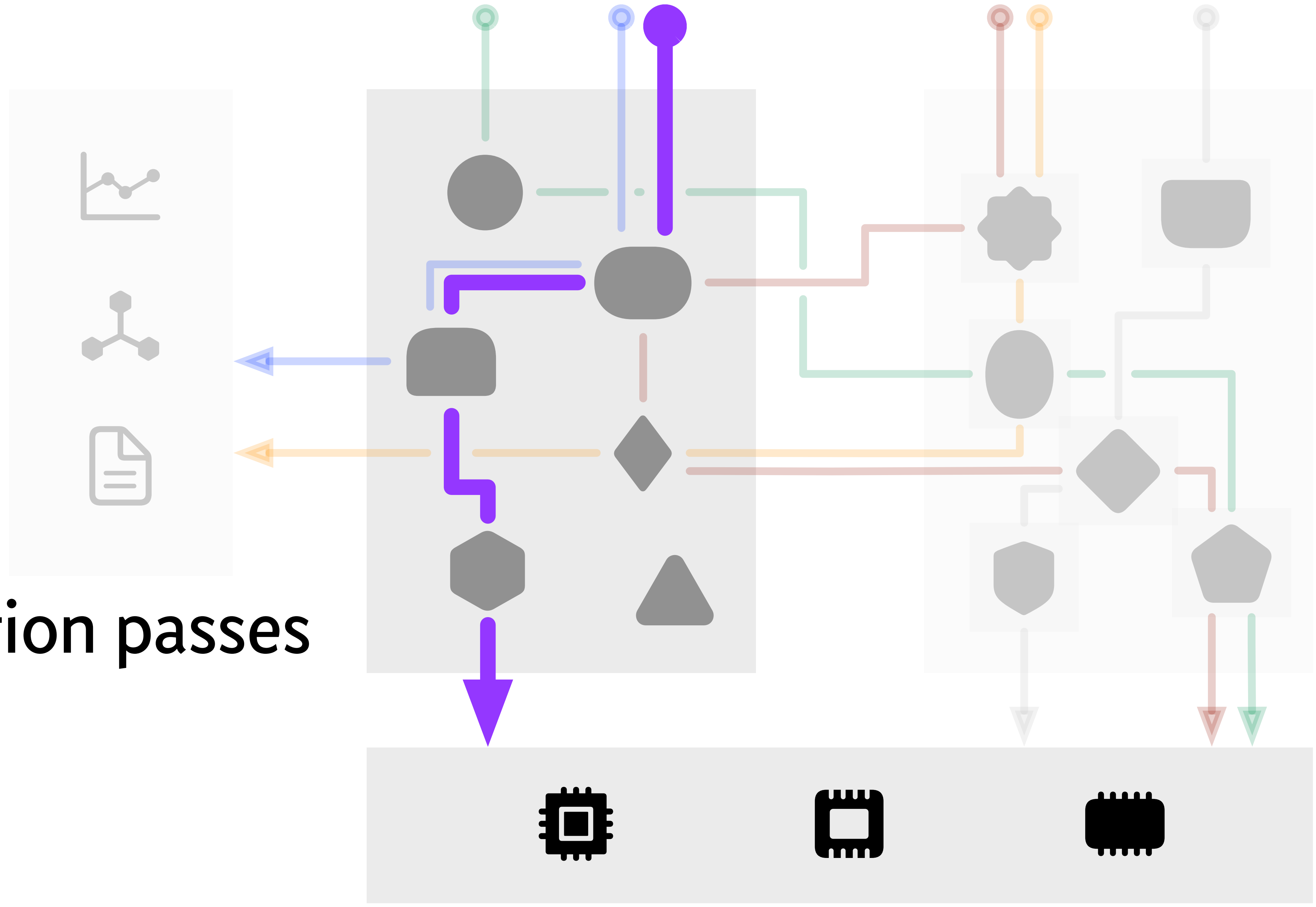




Use Case 1

Teaching compilation with ChocoPy

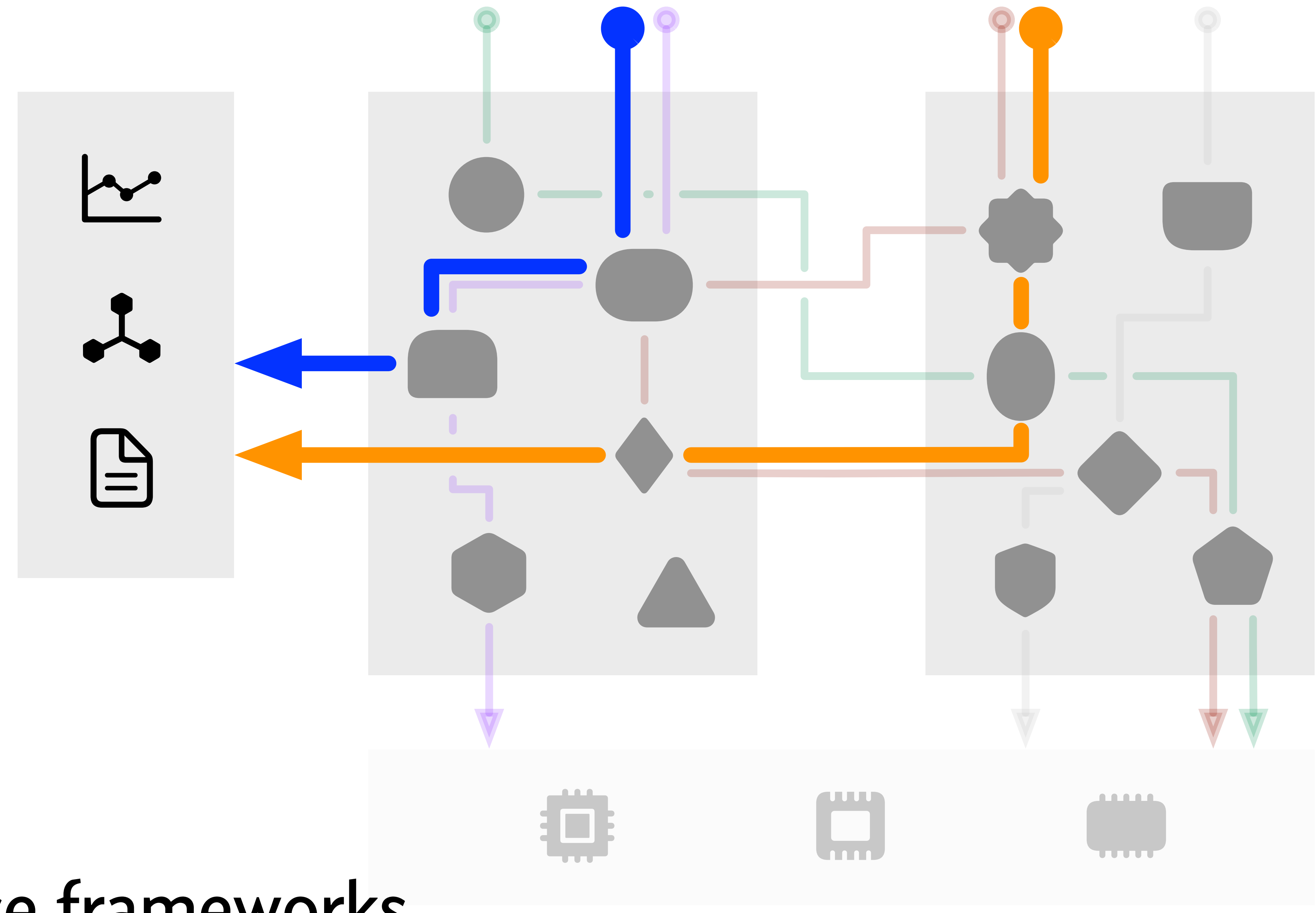
- *User:*
  - Undergraduate students familiar with Python
- *Needs:*
  - Quick and easy installation and build systems
  - Compile time performance is less important
- *Existing Workflows:*
  - Students design ad-hoc IRs, data structures, and optimization passes
- *The xDSL Approach:*
  - Students learn core concepts of SSA-based compilers and can easy transition to MLIR afterwards



Use Case 2

Data-driven compiler design

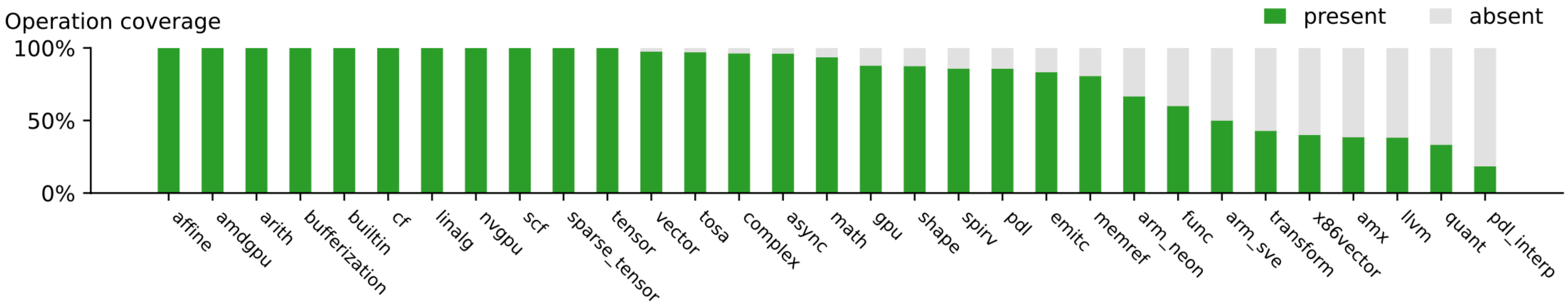
- *User:*
    - Compiler engineers trying to understand their code base
- *Needs:*
    - Scripting languages with good data science workflows
- *Existing Workflows:*
    - Lack of an integrated environment to build analysis tools
- *The xDSL Approach:*
    - xDSL makes MLIR dialects easily accessible from Python
    - Provides a good environment to integrate with data science frameworks



Use Case 2

Data-driven compiler design

- *User:*
  - Compiler engineers trying to understand their code base



- *The xDSL Approach:*

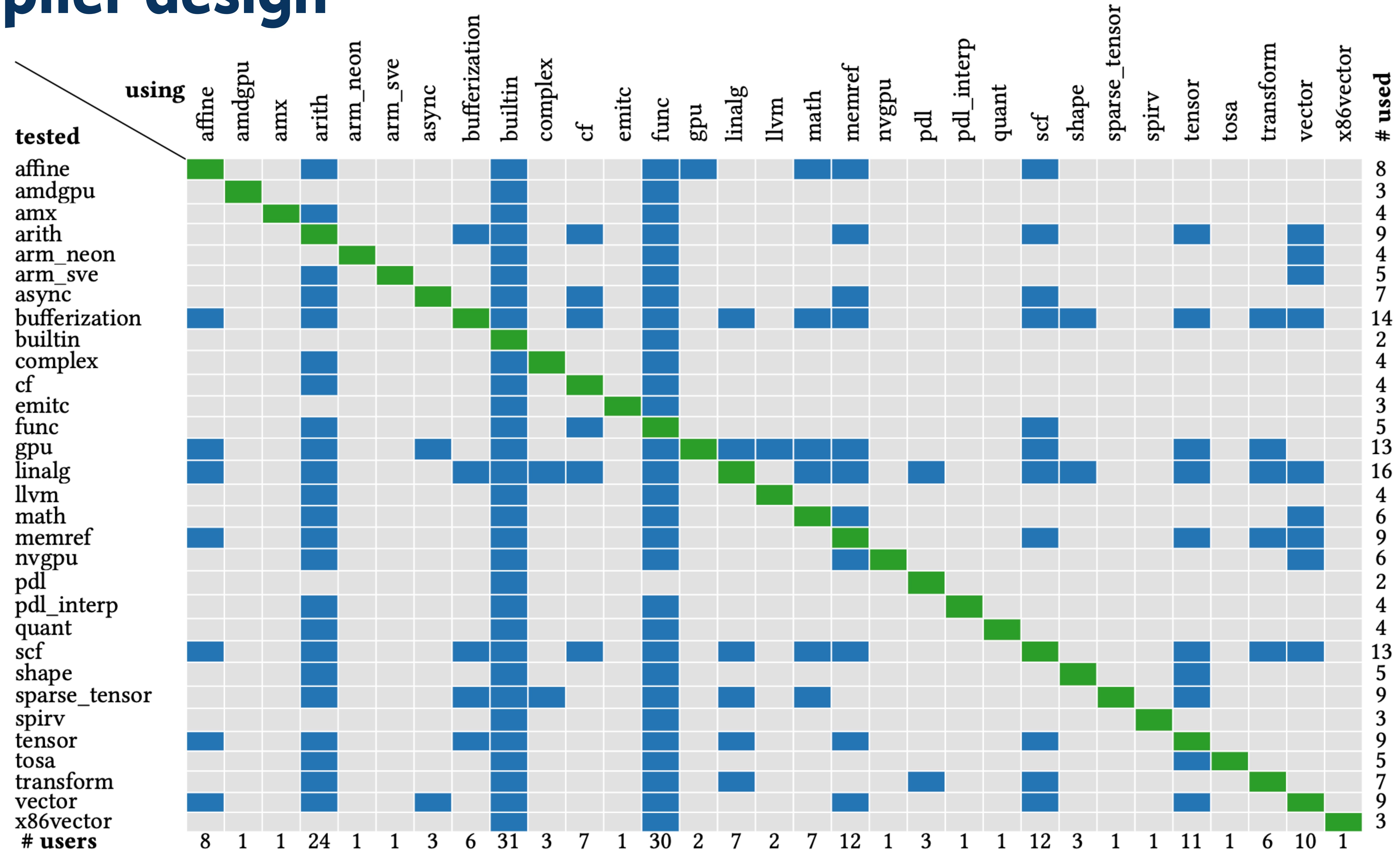
With xDSL we quickly analysed the test coverage of operations of various MLIR dialects

  - Provides a good environment to integrate with data science frameworks

Use Case 2

Data-driven compiler design

- *User:*
  - Compiler engin
- *Needs:*
  - Scripting langua
- *Existing Workflow*
  - Lack of an integ
- *The xDSL Approac*
  - xDSL makes MI
  - Provid



Analysis of dependencies between MLIR dialects in the MLIR test suite

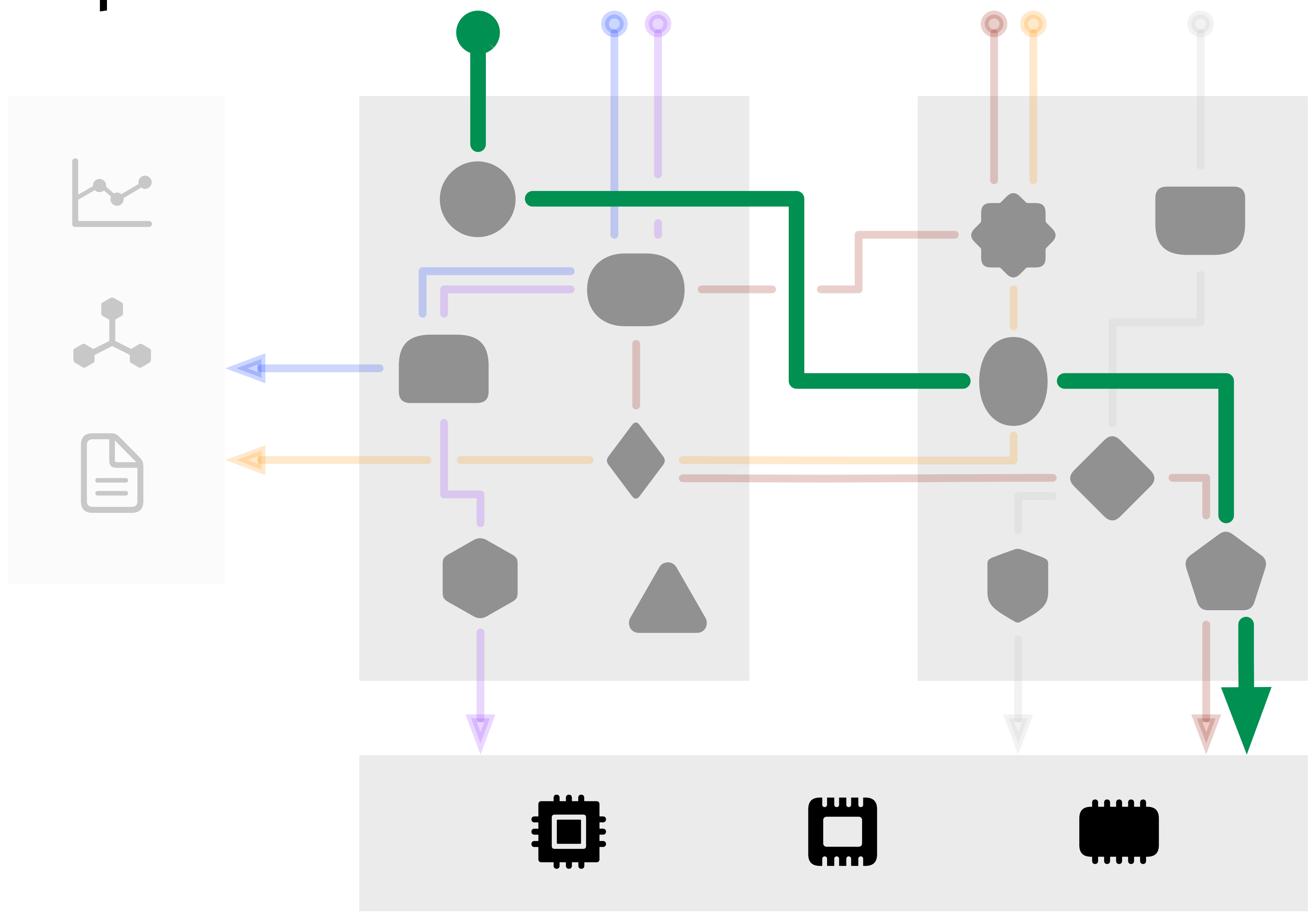
Use Case 3

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

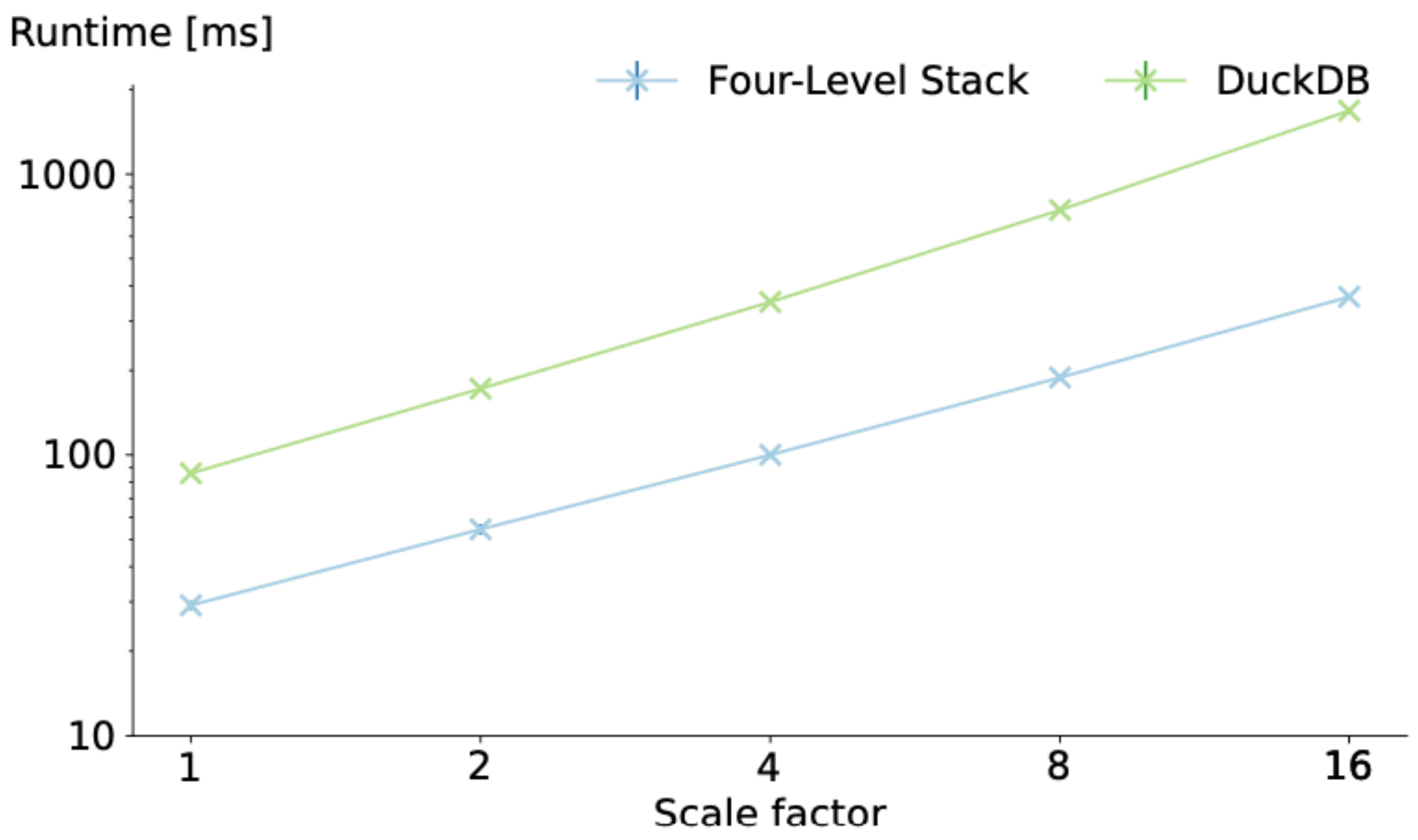
- *Existing Workflows:*
  - Build isolated compiler ecosystem (such as Devito)



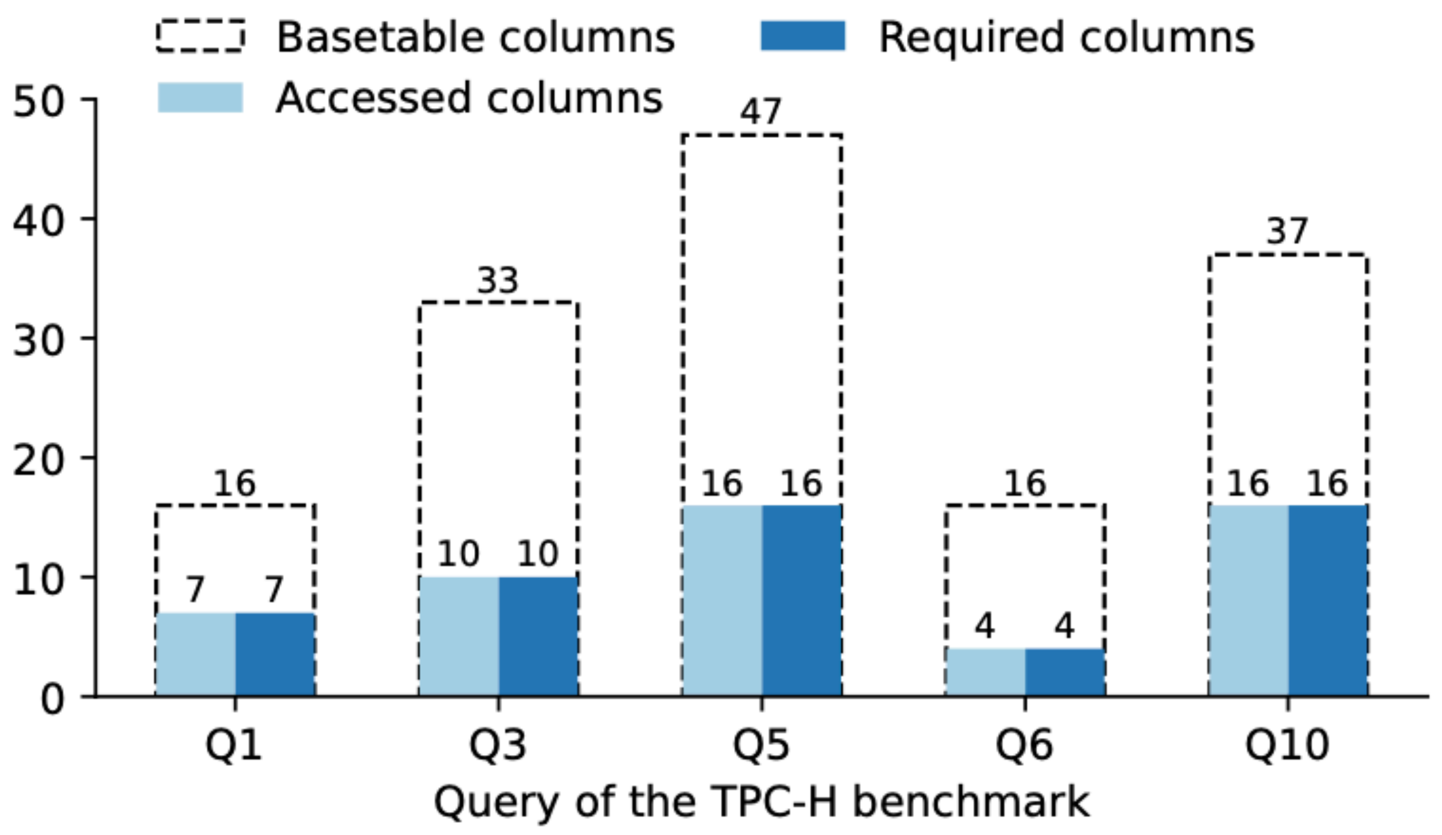
- *The xDSL Approach:*
  - Embed high-level DSL in Python for ease of use
  - Use xDSL dialects in Python and then lower to common dialects that are optimized in MLIR

Use Case 3

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



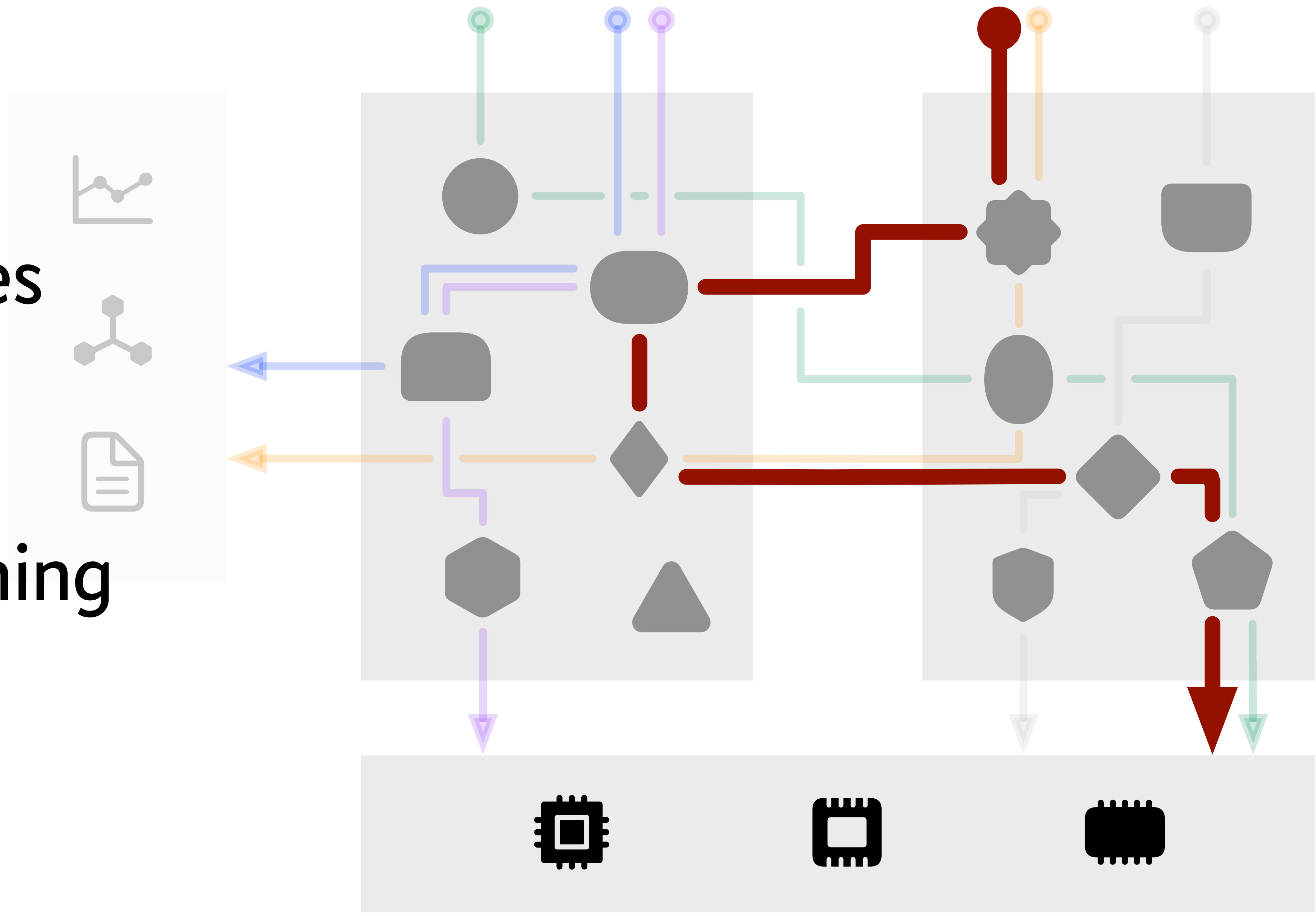
Reduction of basetable column accesses implemented as a compiler optimization pass in Python with xDSL

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

Use Case 4

Prototyping new MLIR features

- *User:*
  - Compiler researchers and engineers
- *Needs:*
  - Prototyping many design; quick incremental build times
- *Existing Workflows:*
  - Directly modify MLIR and LLVM which is time consuming
- *The xDSL Approach:*
  - Prototype new ideas in Python with xDSL
  - Integrate with MLIR for realistic tests and benchmarks



How To Optimize Programs in MLIR Today?

- MLIR provides an infrastructure to express program transformations as *Pattern Rewrites*
- Such rewrites are performed once a pattern has matched in the code
- *Example*: splitting a loop:

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19

scf.for %i=%cst0 to %cst19{
  memref.store %v, %a[%i]
}



...
```

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19
%cst16 = arith.constant  16
scf.for %i=%cst0 to %cst16{
  memref.store %v, %a[%i]
}
scf.for %i=cst16 to %cst19{
  memref.store %v, %a[%i]
}
...
```

Pattern Rewrite in MLIR

Example: Loop splitting

```cpp
struct LoopSplitPattern : public OpRewritePattern<scf::ForOp> {
public:
  using OpRewritePattern::OpRewritePattern;

  LogicalResult matchAndRewrite(scf::ForOp op, PatternRewriter &rewriter) const {
    Location loc = forOp.getLoc();
    Optional<int64_t> ub = getConstantIntValue(forOp.getUpperBound());
    Value split = rewriter.create<arith::ConstantIndexOp>(loc, ub.getValue() - 3);
    auto fst_loop = rewriter.create<scf::ForOp>(loc, forOp.getLowerBound(), split,
                                                forOp.getStep(), forOp.getIterOperands());
    rewriter.eraseBlock(fst_loop.getBody());
    rewriter.cloneRegionBefore(forOp.getRegion(), fst_loop.getRegion(),
                               fst_loop.getRegion().end());

    auto snd_loop = rewriter.create<mlir::scf::ForOp>(loc, split, ub, forOp.getStep(),
                                                      forOp.getIterOperands());
    rewriter.eraseBlock(snd_loop.getBody());
    rewriter.cloneRegionBefore(forOp.getRegion(), snd_loop.getRegion(),
                               snd_loop.getRegion().end());
    rewriter.eraseOp(forOp);
    return success();
  };
};
```

Pattern Rewrite in MLIR

Example: Loop splitting

1. Implement C++ class inheriting from *Pattern Rewriter* interface

2. Match operation

3. Create replacement

4. Erase matched operation

```
struct LoopSplitPattern : public OpRewritePattern<scf::ForOp> {
public:
  using OpRewritePattern::OpRewritePattern;

  LogicalResult matchAndRewrite(scf::ForOp op, PatternRewriter &rewriter) const {
    Location loc = forOp.getLoc();
    Optional<int64_t> ub = getConstantIntValue(forOp.getUpperBound());
    Value split = rewriter.create<arith::ConstantIndexOp>(loc, ub.getValue() - 3);
    auto fst_loop = rewriter.create<scf::ForOp>(loc, forOp.getLowerBound(), split,
                                                 forOp.getStep(), forOp.getIterOperands());
    rewriter.eraseBlock(fst_loop.getBody());
    rewriter.cloneRegionBefore(forOp.getRegion(), fst_loop.getRegion(),
                               fst_loop.getRegion().end());

    auto snd_loop = rewriter.create<mlir::scf::ForOp>(loc, split, ub, forOp.getStep(),
                                                       forOp.getIterOperands());
    rewriter.eraseBlock(snd_loop.getBody());
    rewriter.cloneRegionBefore(forOp.getRegion(), snd_loop.getRegion(),
                               snd_loop.getRegion().end());
    rewriter.eraseOp(forOp);
    return success();
  };
};
```

Composing Rewrites?

How to perform a sequence of rewrites?

- *Example*: splitting a loop + unrolling the second (+ vectorizing first) + …

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19

scf.for %i=%cst0 to %cst19{
  memref.store %v, %a[%i]
}




...
```

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19
%cst16 = arith.constant  16
scf.for %i=%cst0 to %cst16{
  memref.store %v, %a[%i]
}
scf.for %i=cst16 to %cst19{
  memref.store %v, %a[%i]
}
...
```

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19
%cst16 = arith.constant  16
scf.for %i=%cst0 to %cst16{
  memref.store %v, %a[%i]
}
memref.store %v, %a[%cst16]
memref.store %v, %a[%cst17]
memref.store %v, %a[%cst18]
...
```

⊖ In MLIR no way to describe *locations* of rewrites; Usually greedily applied everywhere

⊖ What if a rewrite fails halfway through? Mutating rewrites make *backtracking* difficult

ELEVATE — a Language for Composing Rewrites

Based on ICFP 2020 Paper: ***Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies***
by Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, Michel Steuwer

- We think of a *Rewrite* as function with a specific type:
  Either returning the transformed `IR` of the input program, or returning a `Failure`.

```
type Rewrite = IR ⇒ IR | Failure
```

- The rewrite must be immutable, i.e., they don't modify directly the input program
- Immutable rewrites with this type *compose* nicely into larger rewrites!
- **To prototype ELEVATE in MLIR: we implemented an immutable version of the MLIR IR in xDSL**
- **We describe individual rewrite rules in a declarative MLIR dialect itself!**

ELEVATE Rewrite in MLIR

Example 1: Simple arithmetic rewrite

x * 2 → x >> 1

```
rewrite.rule @mul_to_shift(%op) {
  %pattern = rewrite.pattern() {
    %x    = pdl.operand
    %cst2 = pdl.operation "arith.constant"() ["value" = 2]
    %muli = pdl.root_operation "arith.muli"(%x, %cst2) -> !i32
    rewrite.capture(%muli, %x)
  }
  rewrite.match_and_replace(%op, %pattern) {
    ^(%muli, %x):
      %cst1 = rewrite.new_op "arith.constant"() ["value" = 1] -> !i32
      %shli = rewrite.new_op "arith.shli"(%x, %cst1) -> !i32
      rewrite.return(%shli)
  }
}
```

1. We use the (extended) `pdl` dialect to match the input `%op`

2. The created replacement replaces the matched *root operation*

```
...
%cst2 = arith.constant() ["value" = 2]

%result = arith.muli(%x, %cst2)
...
```

```
...
%cst2 = arith.constant() ["value" = 2]
%cst1 = arith.constant() ["value" = 1]
%result = arith.shli(%x, %cst1)
...
```

3. If `%cst2` has no uses it will be automatically removed

ELEVATE Rewrite in MLIR

Example 2: Loop Splitting

Rewrite

```
rewrite.rule @split_loop(%op) {
 %pattern = rewrite.pattern() {
   %ub  = pdl.attribute
   %for = pdl.root_operation "scf.for"["ub"=%ub]
   rewrite.capture(%for, %ub)
 }
 rewrite.match_and_replace(%op, %pattern) {
   ^(%for, %ub):
    %3 = arith.constant 3
    %s = arith.subi %ub %3
    %fst_loop = rewrite.from_op(%for)["ub"=%s]
    %snd_loop = rewrite.from_op(%for)["lb"=%s]
    rewrite.return(%fst_loop, %snd_loop)
 }
}
```

Computational IR

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19

scf.for %i=%cst0 to %cst19{
  memref.store %v, %a[%i]
}




...
```

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19
%cst16 = arith.constant  16
scf.for %i=%cst0 to %cst16{
  memref.store %v, %a[%i]
}
scf.for %i=cst16 to %cst19{
  memref.store %v, %a[%i]
}
...
```

ELEVATE Rewrite in MLIR

Example 3: Stencil inlining

```
func @simple_stencil(%in, %out) {
  %field = stencil.load(%in)
  %tmp_field = stencil.apply(%field) {
   ^(%field):
     %lhs = stencil.access(%field) ["off"=[-1,0]]
     %rhs = stencil.access(%field) ["off"=[1,0]]
     %added = arith.addf(%lhs, %rhs)
     stencil.return(%added)
  }
  %result_field = stencil.apply(%field, %tmp_field)  {
   ^(%field, %tmp):
     %middle = stencil.access(%field) ["off"=[0,0]]
     %added_access = stencil.access(%tmp) ["off"=[1,2]]


     %result = arith.subf(%middle, %access_added)
     stencil.return(%result)
  }
  stencil.store(%result_field, %out)
  return()
}
```

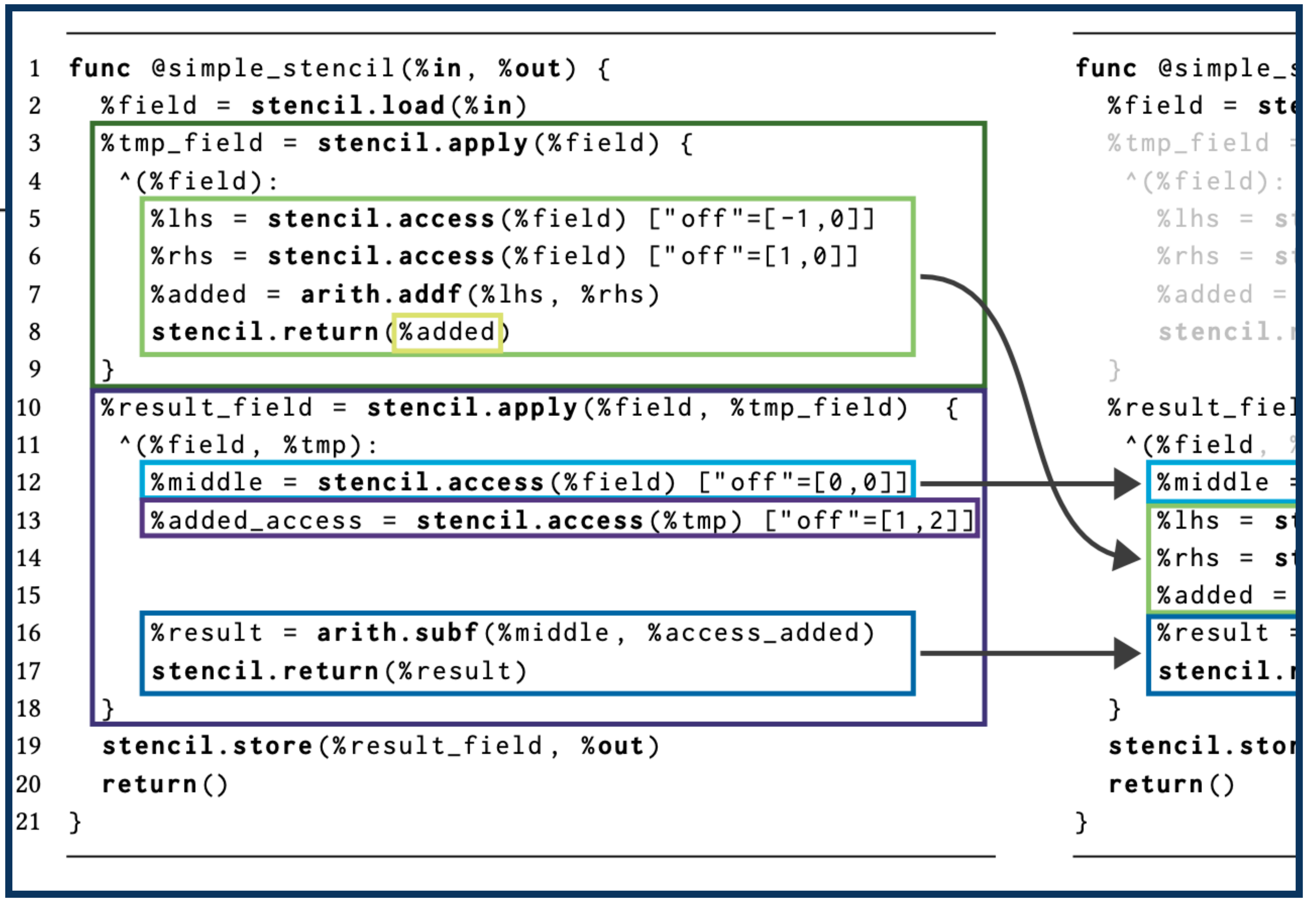
```
func @simple_stencil(%in, %out) {
  %field = stencil.load(%in)
  %tmp_field = stencil.apply(%field) {
   ^(%field):
     %lhs = stencil.access(%field) ["off"=[-1,0]]
     %rhs = stencil.access(%field) ["off"=[1,0]]
     %added = arith.addf(%lhs, %rhs)
     stencil.return(%added)
  }
  %result_field = stencil.apply(%field) {
   ^(%field, %tmp):
     %middle = stencil.access(%field) ["off" = [0, 0]]
     %lhs = stencil.access(%field) ["off" = [0, 2]]
     %rhs = stencil.access(%field) ["offset" = [2, 2]]
     %added = arith.addf(%lhs, %rhs)
     %result = arith.subf(%middle, %added)
     stencil.return(%result)
  }
  stencil.store(%result_field, %out)
  return()
}
```

Optimization implemented in the Open Earth Compiler (https://github.com/spcl/open-earth-compiler/)

ELEVATE Rewrite in MLIR

Example 3: Stencil inlining

```
rewrite.rule @inline_simplified(%op) {
  %pattern = rewrite.pattern() {
    %producer, %producer_result = pdl.operation "stencil.apply"() {
      ^(%field)
        %producer_ops    = rewrite.this_block_ops()
        %produced_value = pdl.operand
        pdl.operation "stencil.return"(%produced_value)
        rewrite.capture(%producer_ops, %produced_value)
    }
    %consumer, %consumer_result = pdl.root_operation "stencil.apply"(%producer_result) {
      ^(%field, %consumed_value):
        %stencil_access = pdl.operation "stencil.access"(%consumed_value)
        %ops = rewrite.this_block_ops()
        %consumer_ops_until = rewrite.ops_until(%ops, %stencil_access)
        %consumer_ops_after = rewrite.ops_after(%ops, %stencil_access)
        rewrite.capture(%consumer_ops_until, %stencil_access, %consumer_ops_after)
    }
    rewrite.capture(%producer, %consumer)
  }
  rewrite.match_and_replace(%op, %pattern) {
    ^(%prod_ops, %prod_value, %cons_ops_until, %stencil_access, %cons_ops_after, %prod, %cons):

  ...
```



Matching of two successive stencil operations

ELEVATE Rewrite in MLIR

Example 3: Stencil inlining

```
  ...
  rewrite.match_and_replace(%op, %pattern) {
    ^(%prod_ops, %prod_value, %cons_ops_until, %stencil_access, %cons_ops_after, %prod, %cons):

    %updated_prod_ops = rewrite.for_each(%prod_ops) { ^(%op):
      %updated_offset = ...  // compute updated offset using %stencil_access's offset
      %updated_op = rewrite.from_op(%op) ["off" = %updated_offset]
      rewrite.yield(%updated_op)
    }
    %updated_cons_ops_after = rewrite.for_each(%cons_ops_after) { ^(%op):
      %operands = ... // iterate over operands and update uses from %stencil_access to %produced_va
      %updated_op = rewrite.from_op(%op, %operands)
      rewrite.yield(%updated_op)
    }
    %new_ops      = rewrite.concat(%cons_ops_until, %updated_prod_ops, %updated_cons_ops_after)
    %new_args     = rewrite.concat_args(%prod, %cons)
    %new_block    = rewrite.new_block(%new_args, %new_ops)
    %new_region   = rewrite.region_from_blocks(%new_block)
    %new_operands = rewrite.concat_operands(%prod, %cons)
    %new_apply_op = rewrite.from_op(%cons, %new_operands, %new_region)
    rewrite.return(%new_apply_op)
  }
}
```

```
func @simple_stencil(%in, %out) {
  %field = stencil.load(%in)
  %tmp_field = stencil.apply(%field) {
   ^(%field):
     %lhs = stencil.access(%field) ["off"=[-1,0]]
     %rhs = stencil.access(%field) ["off"=[1,0]]
     %added = arith.addf(%lhs, %rhs)
     stencil.return(%added)
  }
  %result_field = stencil.apply(%field) {
   ^(%field, %tmp):
     %middle = stencil.access(%field) ["off" = [0, 0]]
     %lhs = stencil.access(%field) ["off" = [0, 2]]
     %rhs = stencil.access(%field) ["offset" = [2, 2]]
     %added = arith.addf(%lhs, %rhs)
     %result = arith.subf(%middle, %added)
     stencil.return(%result)
  }
  stencil.store(%result_field, %out)
  return()
}
```

Our declarative rewrite replaces about 400 lines of imperative C++ code!

https://github.com/spcl/open-earth-compiler/blob/master/lib/Dialect/Stencil/StencilInliningPass.cpp

Combinators and Traversals in ELEVATE

- **Combinators** allow to build more complex strategies from simple once, e.g.:
  - **`s1;s2`** *(Sequential Composition)*: apply second strategy `s1` to result of the first `s2`
  - **`try {s1} else {s2}`** (*Left Choice*): apply second strategy `s2` if first strategy `s1` fails
- **Traversals** allow to describe precise locations in the IR, e.g.:
  - **`top_to_bottom {s}`**: apply strategy `s` to the IR line by line, top to bottom
  - **`regionN[n]{s}`**, **`blockN[n]{s}`**, **`opN[n]{s}`**: apply strategy `s` to n-*th* region/block/op

Composing Rewrites in ELEVATE

```
rewrite.strategy @split_and_unroll_snd() {
  rewrite.apply @split_loop
  rewrite.top_to_bottom {
  rewrite.skip 1 {
    rewrite.if "scf.for" {
      rewrite.apply @unroll_loop
    }
  }
}
```

sequential composition

traversals & predicates to describe locations

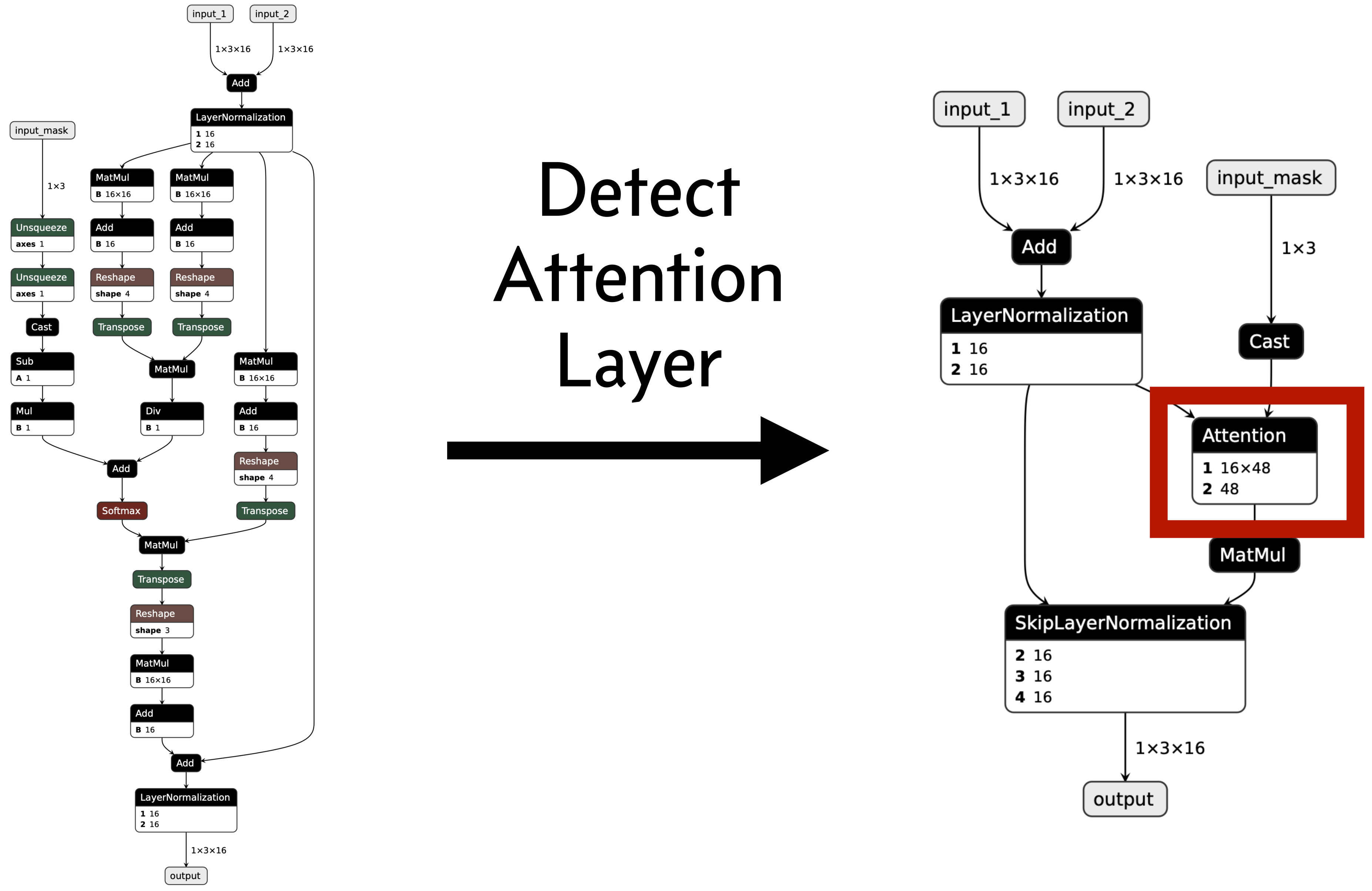
```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19

scf.for %i=%cst0 to %cst19{
  memref.store %v, %a[%i]
}



...
```

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19
%cst16 = arith.constant  16
scf.for %i=%cst0 to %cst16{
  memref.store %v, %a[%i]
}
scf.for %i=cst16 to %cst19{
  memref.store %v, %a[%i]
}
...
```

```
...
%cst0  = arith.constant  0
%cst19 = arith.constant  19
%cst16 = arith.constant  16
scf.for %i=%cst0 to %cst16{
  memref.store %v, %a[%i]
}
memref.store %v, %a[%cst16]
memref.store %v, %a[%cst17]
memref.store %v, %a[%cst18]
...
```

Use Cases for Composable Rewrites

Detection of Layers in ML models



- Enables experts to optimize ML layers specially
- Many slightly different cases could easily be described by composing individual rewrites
- Imperative C++ or Python matching code written by expert compiler engineers, e.g., at Microsoft

github.com

```python
    def fuse(self, normalize_node, input_name_to_nodes, output_name_to_node):
        # Sometimes we can not fuse skiplayernormalization since the add before layernorm has an output that used by nodes outside skiplayernorm
        # Conceptually we treat add before layernorm as skiplayernorm node since they share the same pattern
        start_node = normalize_node
        if normalize_node.op_type == 'LayerNormalization':
            add_before_layernorm = self.model.match_parent(normalize_node, 'Add', 0)
            if add_before_layernorm is not None:
                start_node = add_before_layernorm
            else:
                return

        # SkipLayerNormalization has two inputs, and one of them is the root input for attention.
        qkv_nodes = self.model.match_parent_path(start_node, ['Add', 'MatMul', 'Reshape', 'Transpose', 'MatMul'],
                                                 [None, None, 0, 0, 0])
        einsum_node = None
        if qkv_nodes is not None:
            (_, matmul_qkv, reshape_qkv, transpose_qkv, matmul_qkv) = qkv_nodes
        else:
            # Match Albert
            qkv_nodes = self.model.match_parent_path(start_node, ['Add', 'Einsum', 'Transpose', 'MatMul'],
                                                     [1, None, 0, 0])
            if qkv_nodes is not None:
                (_, einsum_node, transpose_qkv, matmul_qkv) = qkv_nodes
```

200 lines of arbitrary imperative Python code

- Declarative rewrite written by PhD student

```
%FuseAttentionLayer : !strategy = elevate.strategy() ["strategy_name"="FuseAttentionLayer"] {
  ^strategy(%op : !operation):
   %pattern : !pattern = match.pattern() {
    // input to the attention layer
    %layer_norm_cst_0 : !value = pdl.operand()
    %layer_norm_cst_weight : !value = pdl.operand()
    %layer_norm_cst_bias : !value = pdl.operand() []

    (%add2, %add2_result) = pdl.operation() ["name"="onnx.Add"]
    (%layer_norm1, %layer_norm1_result) = pdl.operation(%add2_result, %layer_norm_cst_weight, %layer_norm_cst

    // detect mask nodes
    %input_mask = pdl.operand() []
    (%unsqueeze1_mask, %unsqueeze1_mask_result) = pdl.operation(%input_mask : !value) ["name"="onnx.Unsque
    (%unsqueeze0_mask, %unsqueeze0_mask_result) = pdl.operation(%unsqueeze1_mask_result : !value) ["name"=
    (%cast_mask, %cast_mask_result) = pdl.operation(%unsqueeze0_mask_result : !value) ["name"="onnx.Cast"]
```

< 100 lines of declarative dialect could easily be generated

Use Cases for Composable Rewrites

Halide-Style *Schedules* as composition of rewrites

- ICFP 2020 paper demonstrates how to use combinators and traversals to build a *Schedule* describing a specific way to optimize a program
- Gives performance experts precise control over the optimizations applied to a program


ICFP-2020.pdf
Page 1 of 29
Search

ICFP 2020

Achieving High-Performance the Functional Way

A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies

BASTIAN HAGEDORN, University of Münster, Germany
JOHANNES LENFERS, University of Münster, Germany
THOMAS KŒHLER, University of Glasgow, UK
XUEYING QIN, University of Glasgow, UK
SERGEI GORLATCH, University of Münster, Germany
MICHEL STEUWER, University of Glasgow, UK

Optimizing programs to run efficiently on modern parallel hardware is hard but crucial for many applications. The predominantly used imperative languages - like C or OpenCL - force the programmer to intertwine the code describing functionality and optimizations. This results in a portability nightmare that is particularly problematic given the accelerating trend towards specialized hardware devices to further increase efficiency.

ELEVATE

```
val loopPerm = (
  tile(32,32)         ‘@‘ outermost(mapNest(2))     ‘;;‘
  fissionReduceMap ‘@‘ outermost(appliedReduce) ‘;;‘
  split(4)            ‘@‘ innermost(appliedReduce) ‘;;‘
  reorder(Seq(1,2,5,3,6,4))                        ‘;;‘
  vectorize(32)       ‘@‘ innermost(isApp(isApp(isMap))))
(loopPerm ‘;‘ lowerToC)(mm)
```

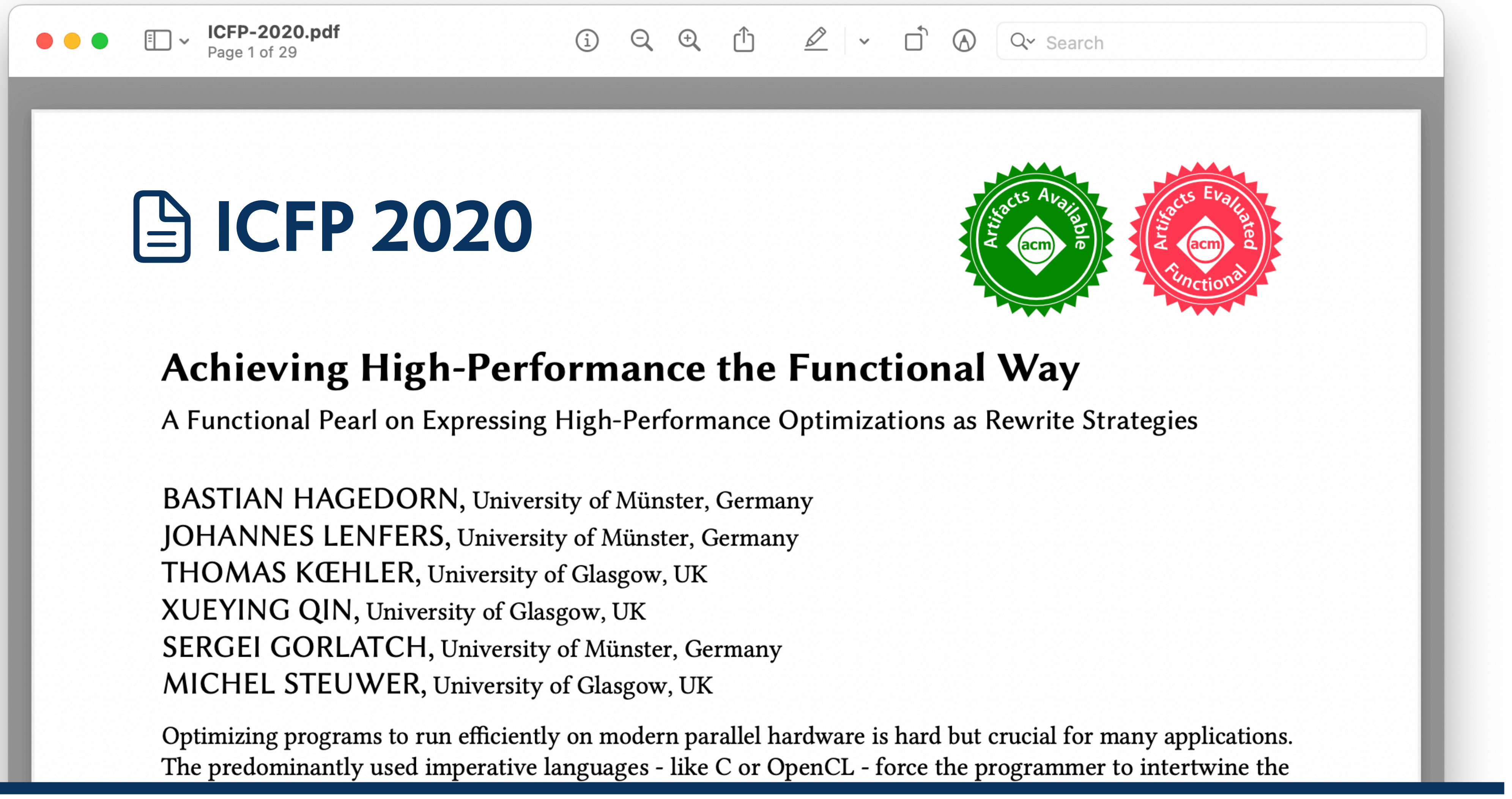
tvm

```
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0],C.op.axis[1],32,32)
k,                = s[C].op.reduce_axis
ko, ki            = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)
```

Use Cases for Composable Rewrites

Halide-Style *Schedules* as composition of rewrites

- ICFP 2020 expresses equivalent TVM schedules purely as compositions of rewrites in ELEVATE
- Demonstrate same performance as TVM compiler


ICFP-2020.pdf
Page 1 of 29

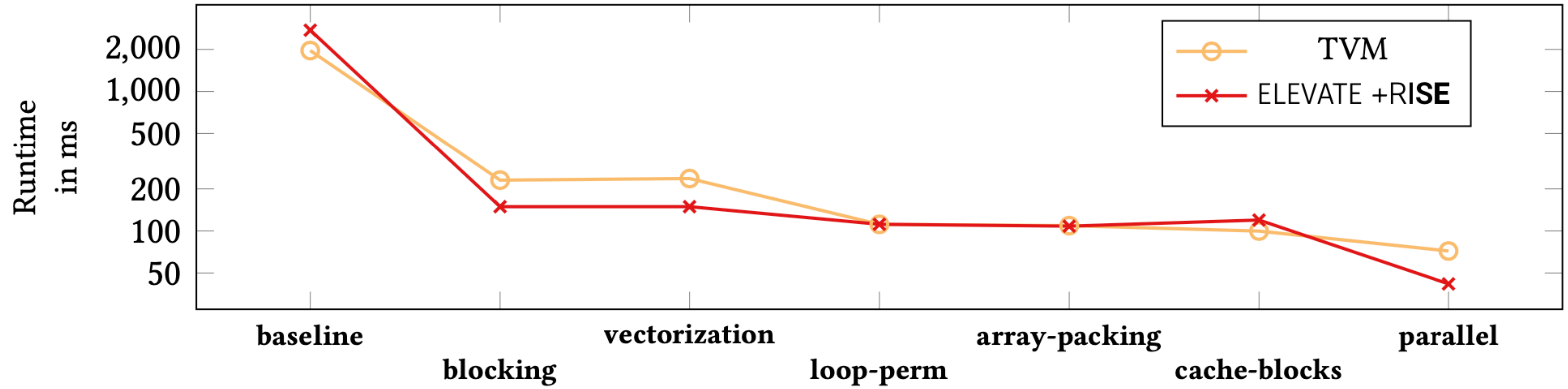
ICFP 2020

Achieving High-Performance the Functional Way

A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies

BASTIAN HAGEDORN, University of Münster, Germany
JOHANNES LENFERS, University of Münster, Germany
THOMAS KŒHLER, University of Glasgow, UK
XUEYING QIN, University of Glasgow, UK
SERGEI GORLATCH, University of Münster, Germany
MICHEL STEUWER, University of Glasgow, UK

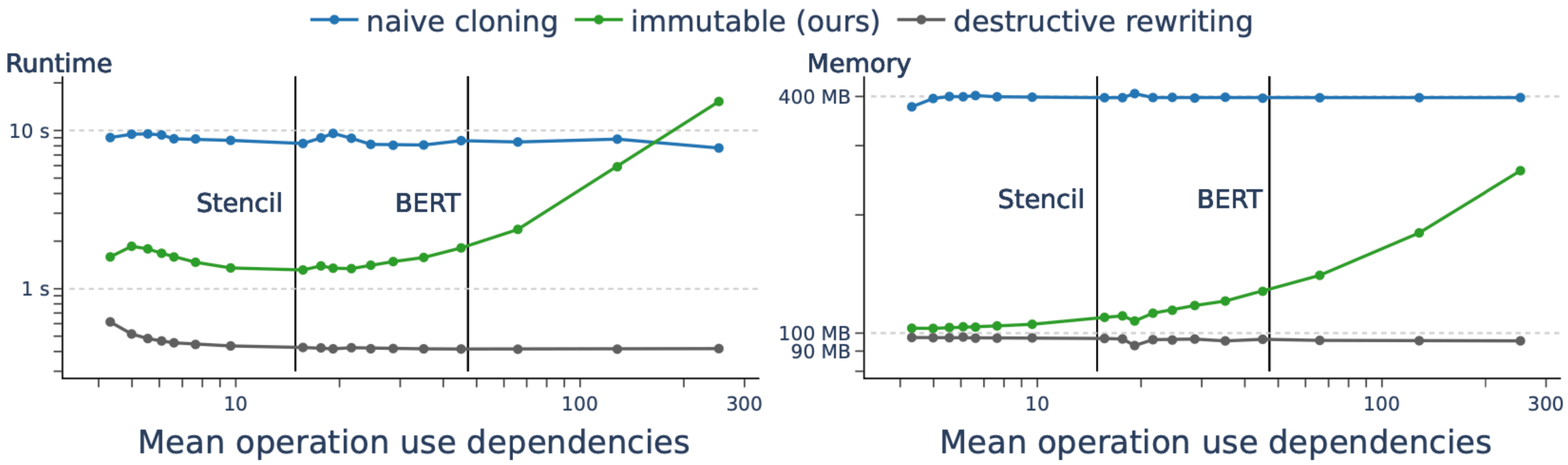
Optimizing programs to run efficiently on modern parallel hardware is hard but crucial for many applications. The predominantly used imperative languages - like C or OpenCL - force the programmer to intertwine the



What’s Next for ELEVATE in MLIR?

Bring all of ELEVATE capabilities to MLIR for expressing rewrites as compositions

- We have a working prototype implementation in xDSL, we are interested in a C++ MLIR implementation
- xDSL is a great prototyping framework!
- Overheads of immutable rewriting are reasonable for many use cases
- Rewriting with an immutable IR is much more efficient than naive cloning for supporting backtracking



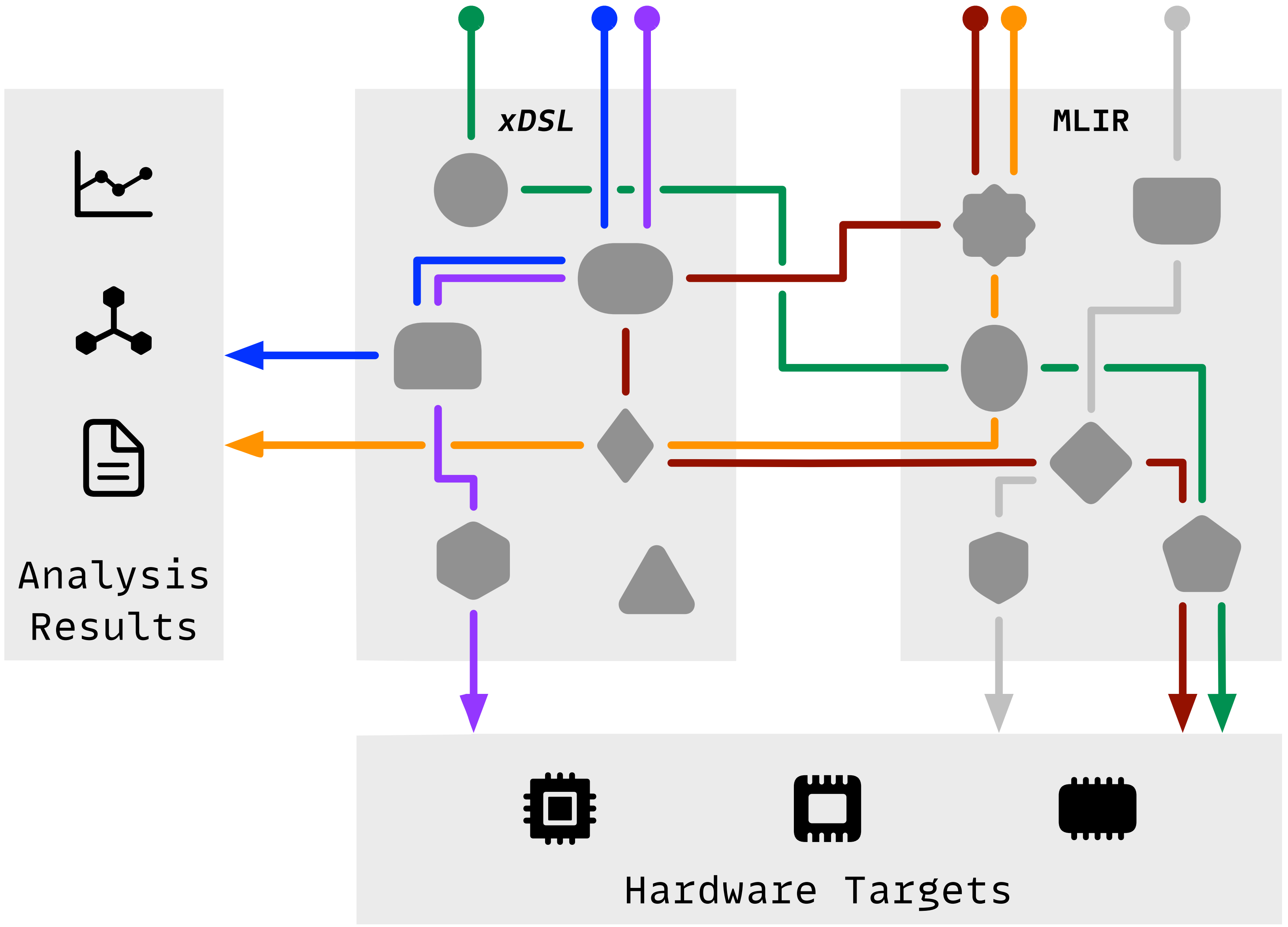
Summary

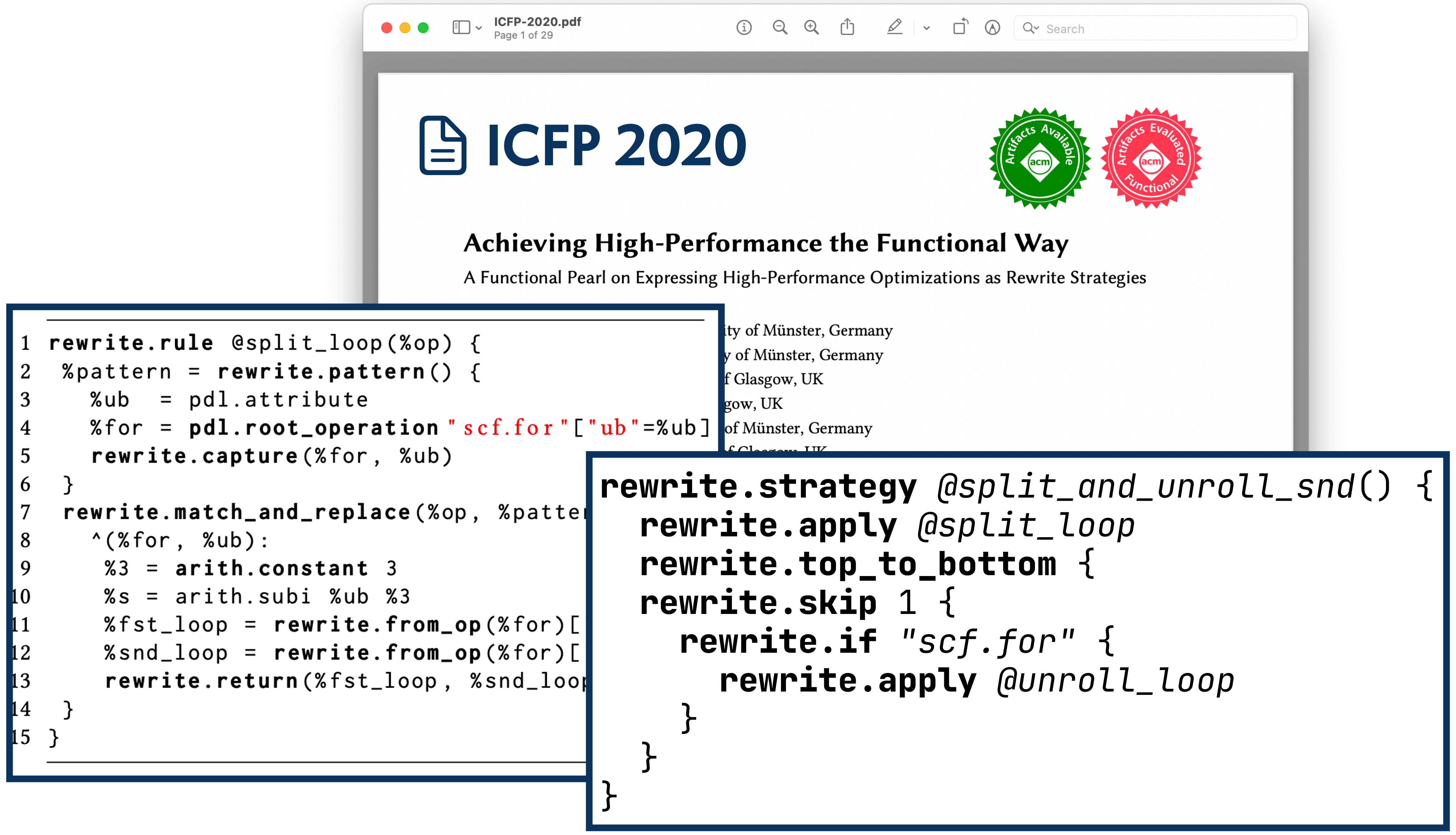
xDSL — a Python *Sidekick* to MLIR | ELEVATE — a language for composing rewrites

- MLIR provides great opportunities to share compiler infrastructure
- Many DSL developers prefer Python and are not part of the MLIR ecosystem
- **xDSL —** a *sidekick* of MLIR enables many deeply integrated use cases leveraging MLIR
- **ELEVATE** — a language for composing rewrites allows describing complex optimizations easily and opens up interesting use cases by providing control over the rewrite process

Michel Steuwer — Modern DSL Compiler Development With MLIR

xDSL — a Python *Sidekick* to MLIR | ELEVATE — a language for composing rewrites





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

https://elevate-lang.org

https://michel.steuwer.info michel.steuwer@ed.ac.uk