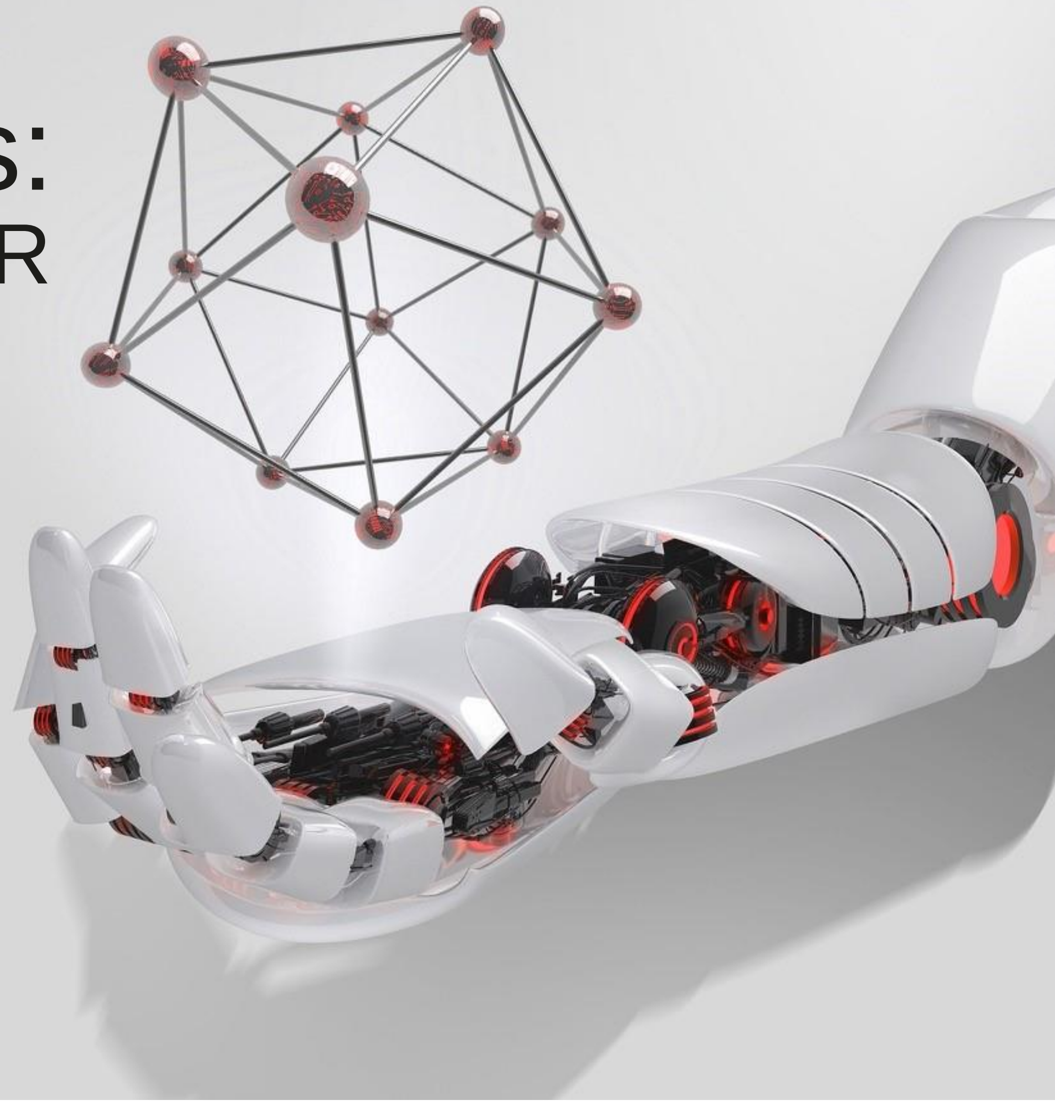
Transform-dialect schedules:
writing MLIR-lowering pipelines in MLIR



Rolf Morel

<rolf.morel@huawei.com>

2024 EuroLLVM, Vienna, April 10-11

Huawei R&D UK, Cambridge
University of Oxford (DPhil, recently defended)

UNIVERSITY OF OXFORD

2024 EURO LLVM
Vienna Austria
DEVELOPERS' MEETING

HUAWEI

What we want:

Declarative & *modular* compilers!

Our approach:

- Multi-level rewriting, hence *MLIR*
- Declarative rewriting, hence *schedules*

In this talk:

Schedules for complete lowering & optimization *of* MLIR
obtained by composing small schedules written *in* MLIR

What are schedules?

program = algorithm + schedule

Image processing à la Halide

algorithm

```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)
out(x,y) = blurx(x,y-1)
         + blurx(x,y)
         + blurx(x,y+1)
```

program

```
par for out.yₒ in 0..out.y.extent/4
 for out.xₒ in 0..out.x.extent/4
  alloc blurx[blurx.y.extent][blurx.x.extent]
  for out.yᵢ in 0..4
   let blurx.y.min = 4*out.yₒ.min + out.yᵢ.min – 1
   for blurx.y in blurx.y.min..blurx.y.max
    for blurx.xₒ in blurx.x.min/4..blurx.x.max/4
     vec for blurx.xᵢ in 0..4
      blurx[blurx.y.stride*blurx.y+...] =
          in[in.y.stride*(blurx.y.min+blurx.y)
             +4*blurx.xₒ+ramp(4)] + ...
   vec for out.xᵢ in 0..4
    out[out.y.stride*(4*(out.yₒ-out.yₒ.min)+out.yᵢ)+...] =
      blurx[blurx.y.stride*(out.yᵢ-1-blurx.y.min)
            + out.xᵢ - blurx.x.min] + ...
```

Halide (Ragan-Kelley et al, 2013)

Image processing à la Halide

algorithm (high-level code)

```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)
out(x,y) = blurx(x,y-1)
         + blurx(x,y)
         + blurx(x,y+1)
```

schedule

```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_0
       compute at out.y_i
out: split x by 4 → x_o, x_i
     split y by 4 → y_o, y_i
     reorder: y_o, x_o, y_i, x_i
     parallelize: y_o
     vectorize: x_i
```

program (lowered & optimized code)

```
par for out.y_o in 0..out.y.extent/4
 for out.x_o in 0..out.x.extent/4
  alloc blurx[blurx.y.extent][blurx.x.extent]
  for out.y_i in 0..4
   let blurx.y.min = 4*out.y_o.min + out.y_i.min - 1
   for blurx.y in blurx.y.min..blurx.y.max
    for blurx.x_o in blurx.x.min/4..blurx.x.max/4
     vec for blurx.x_i in 0..4
      blurx[blurx.y.stride*blurx.y+...] =
          in[in.y.stride*(blurx.y.min+blurx.y)
             +4*blurx.x_o+ramp(4)] + ...
   vec for out.x_i in 0..4
    out[out.y.stride*(4*(out.y_o-out.y_o.min)+out.y_i)+...] =
      blurx[blurx.y.stride*(out.y_i-1-blurx.y.min)
            + out.x_i - blurx.x.min] + ...
```

Halide (Ragan-Kelley et al, 2013)

Many scheduling DSLs

… in support of
optimizing BLAS / tensor programs:

Tensor Comprehensions
(Vasilache et al, 2018)

TTile
(Tollenaere et al, 2021)

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024), lambda y, x:
            t.sum(A[k, y] * B[k, x], axis=k))
s = t.create_schedule(C.op)
```

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

+ Loop Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

Fireiron
(Hagedorn et al, 2020)

Tiramisu
(Baghdadi et al, 2019)

TVM (Chen et al, 2018)

All share the same *main idea*:
declarative descriptions of how to transform code

What is the Transform Dialect?

MLIR’s Transform *meta*-Dialect

transform IR describes transformations on ***payload IR***

payload IR:

```
func.func @gemv(%α, %A, %x, %β, %y) {
 %βy = linalg.generic attrs {iter_types = ["par"]} ... {
  %βy_elem = arith.mulf %β_elem, %y_elem
  linalg.yield %βy_elem : f32
 } -> tensor<?xf32>
 ...
 %αAx_plus_βy = linalg.generic
   { iter_types = ["par", "reduction"]} ... {
    ...
 } -> tensor<?x?xf32>
 return %αAx_plus_βy : tensor<?x?xf32>
}
```
.mlir

transform IR:

```
transform.sequence ... {
^bb1(%arg: !transform.any_op):
  %elemwise = transform.structured.match
                 attrs { iter_types = ["par"] } %arg
  transform.structured.tile_using_for %elemwise [4]
}
```
.mlir

Transform interpreter

transformed IR:

```
func.func @gemv(%α, %A, %x, %β, %y) {
 %dim = tensor.dim %y, %c0
 %βy = scf.for %i = %c0 to %dim step %c4 ... {
  %ex_slice = tensor.extract_slice ...
  ... = linalg.generic attrs {iter_types = ["par"]} {
   %βy_slice_elem = arith.mulf %β_elem, %y_slice_elem
   linalg.yield %βy_slice_elem : f32
  } -> tensor<?xf32>
  %in_slice = tensor.insert_slice ...
  scf.yield %in_slice
 }
 ...
 %αAx_plus_βy = linalg.generic
   { iter_types = ["par", "reduction"]} ... {
    ...
 } -> tensor<?x?xf32>
 return %αAx_plus_βy : tensor<?x?xf32>
}
```
.mlir

MLIR's Transform *meta*-Dialect

transform IR describes transformations on ***payload IR***

payload IR *and* **transform IR:**

```
.mlir
module {
 func.func @name (%arg0, ...) {
   %0 = some.op(...)
   %1 = some.other.op(%0, %arg0)
   ...
 }
}
transform.sequence failures(...) {
 ops for matching ops in payload
 ...
 ops for optimizing matched ops
 AND/OR
 ops for lowering matched ops
 ...
}
```

Transform interpreter

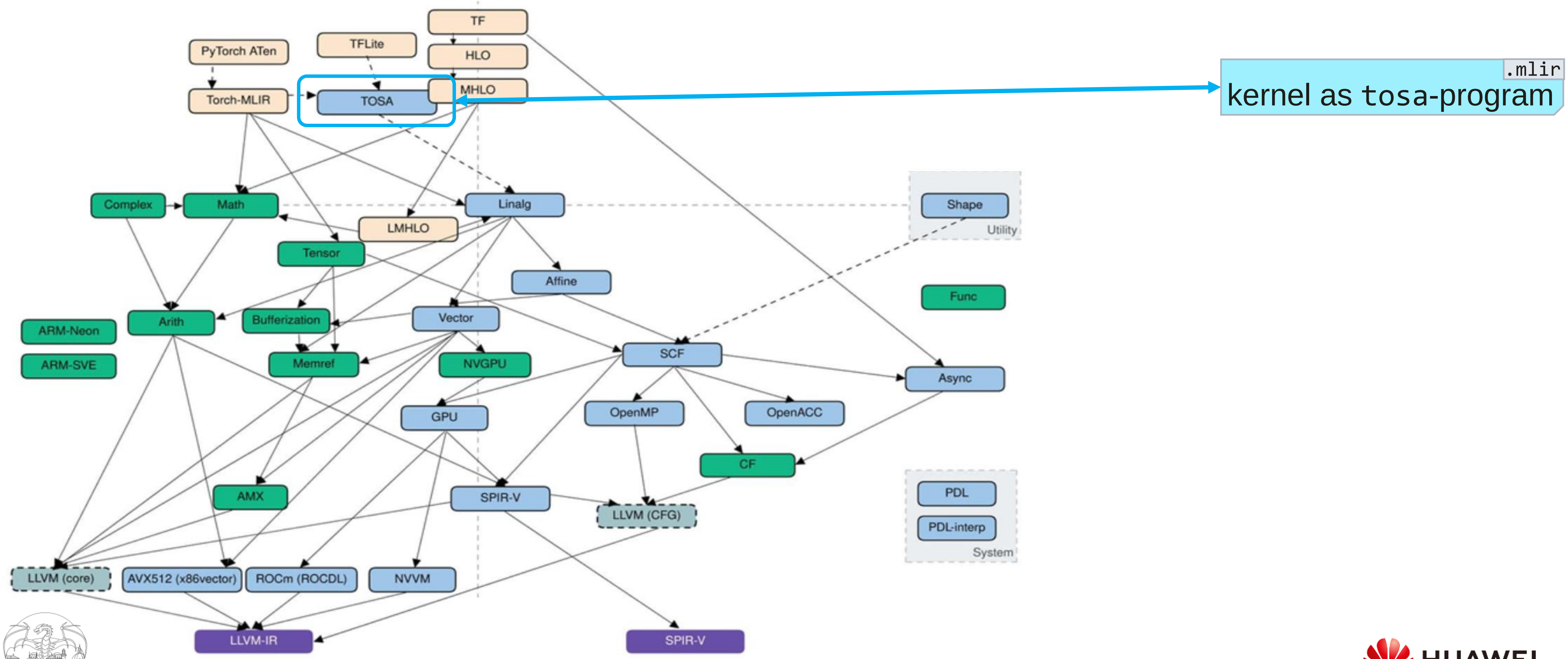
transformed IR:

```
.mlir
module {
 func.func @name (%arg0, ...) {
   %0 = some.op(...)
   ...
   %1 = scf.for %i = %c0 to %arg0 step %k {
     %2 = some.other.op(%0, %i)
     ...
     scf.yield
   }
   ...
 }
}
```

Tutorial: Controllable Transformations in MLIR, *Alex Zinenko, EuroLLVM 2023*
& https://mlir.llvm.org/docs/Tutorials/transform, *Alex Zinenko*

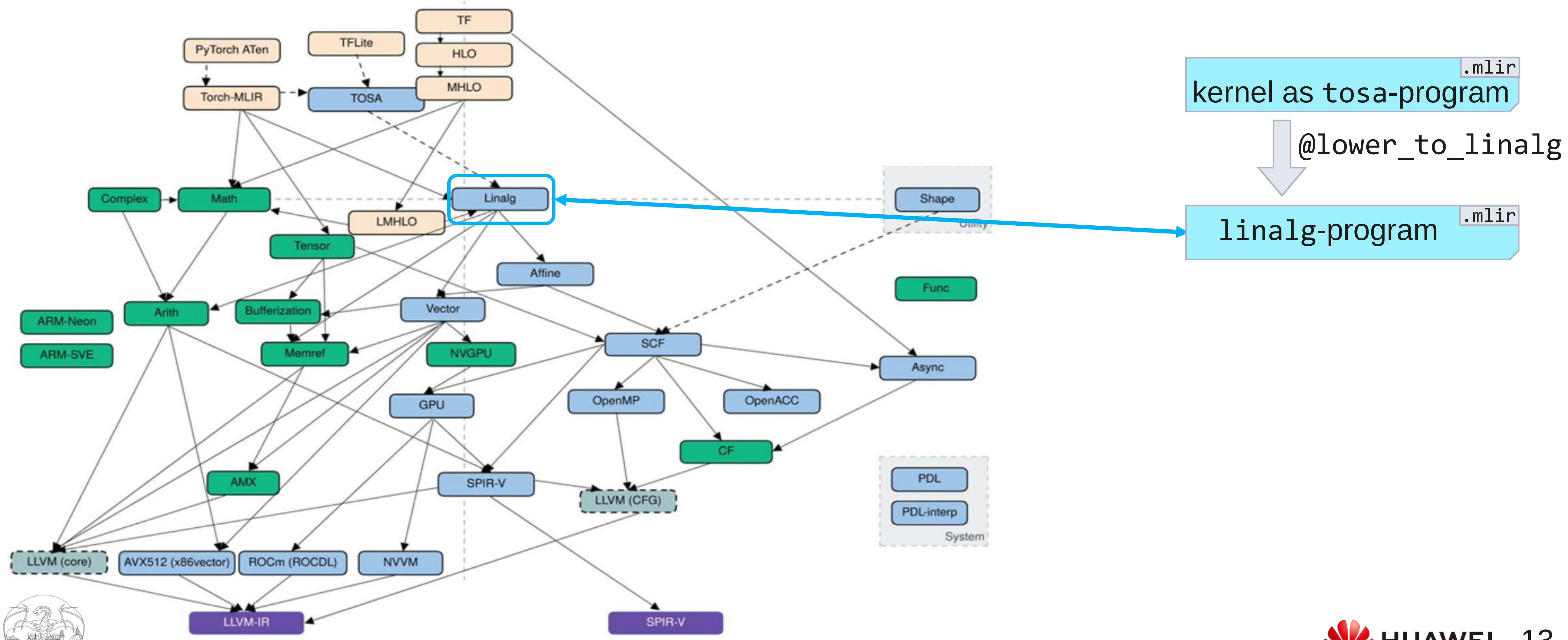
How about combining schedules and multi-level rewriting?

Progressive lowering/optimization
through the dialects



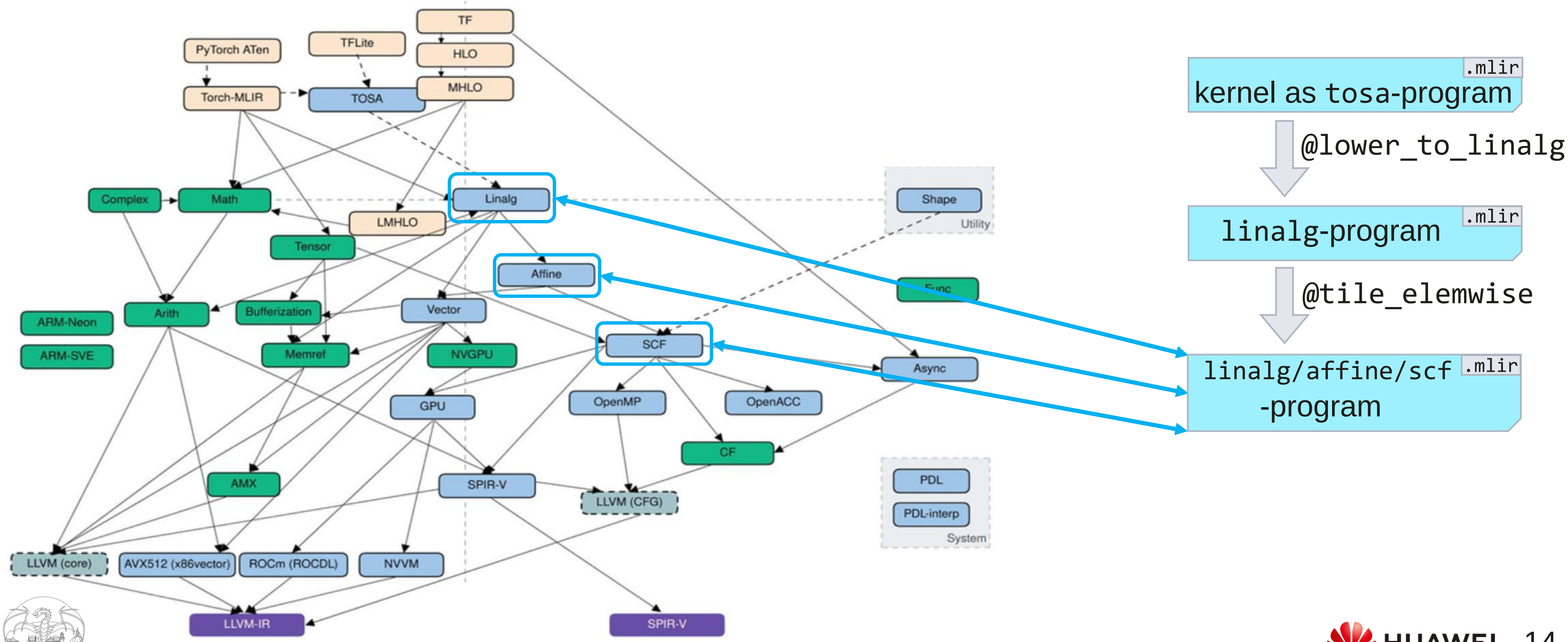
(Quinn Dawkins, 2022)

Progressive lowering/optimization through the dialects



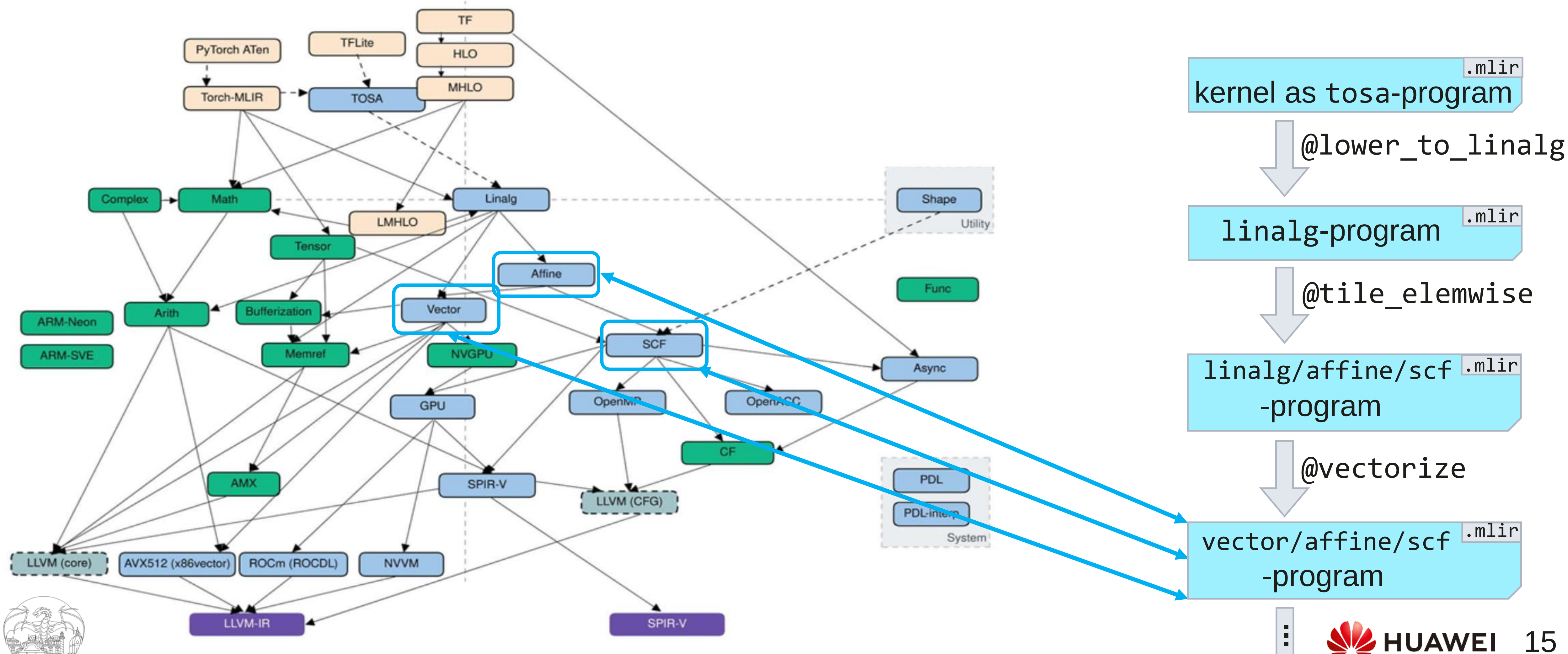
(Quinn Dawkins, 2022)

Progressive lowering/optimization
through the dialects



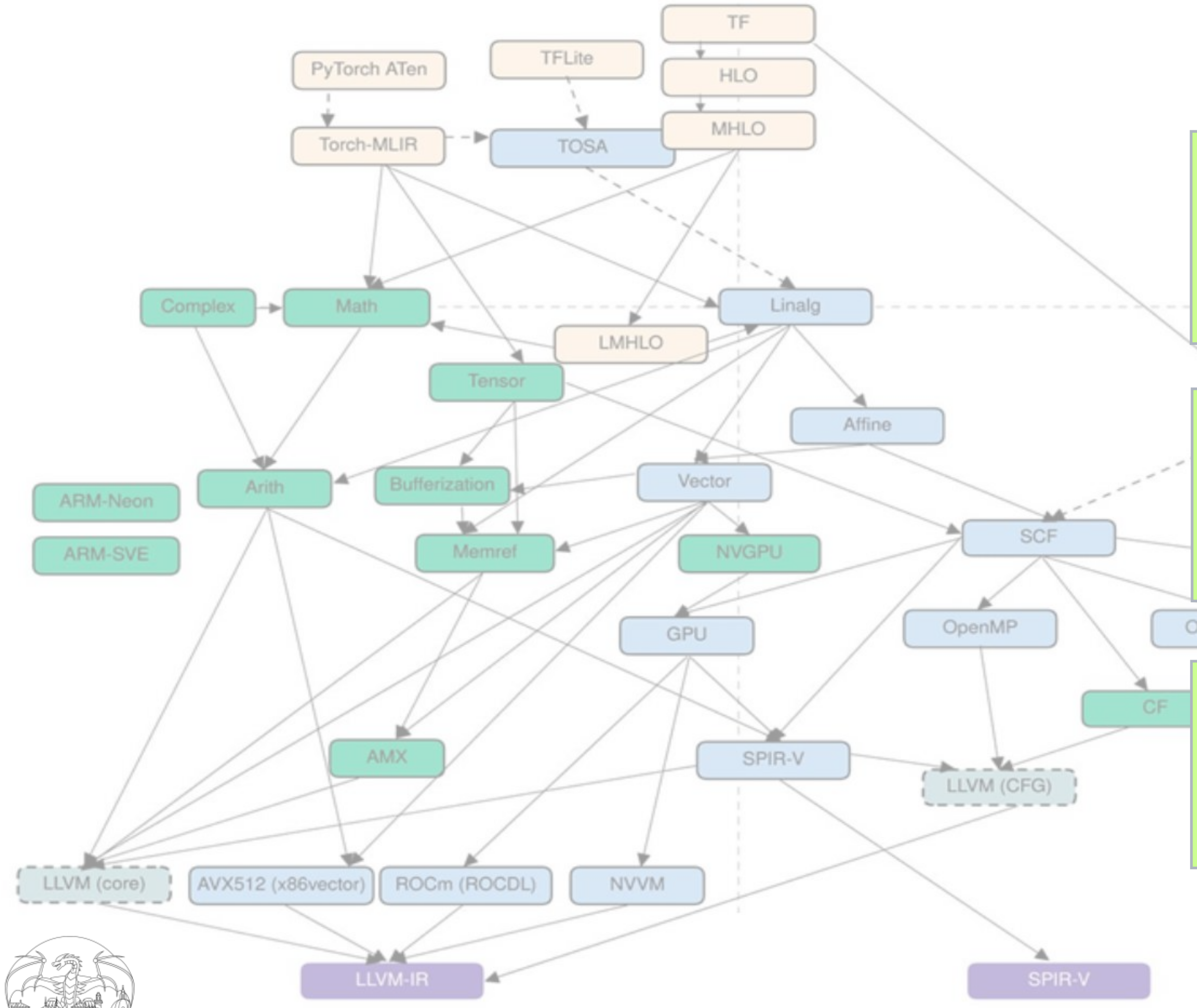
(Quinn Dawkins, 2022)

Progressive lowering/optimization through the dialects



(Quinn Dawkins, 2022)

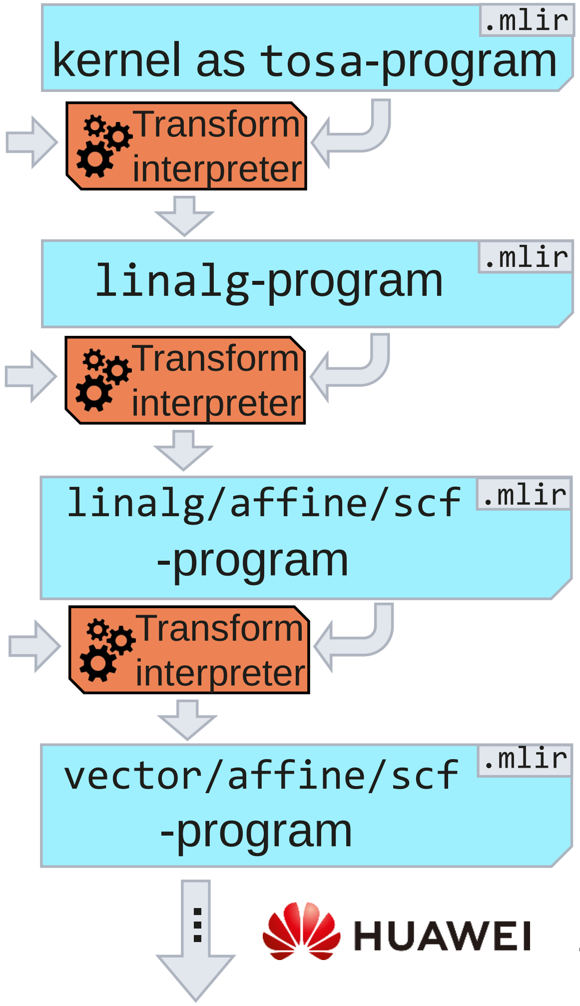
Progressive lowering/optimization
through the dialects, schedule by schedule



.mlir
```
// @lower_to_linalg
...
transform.apply_registered_pass
  "tosa-to-linalg-pipeline" to %0
...
```

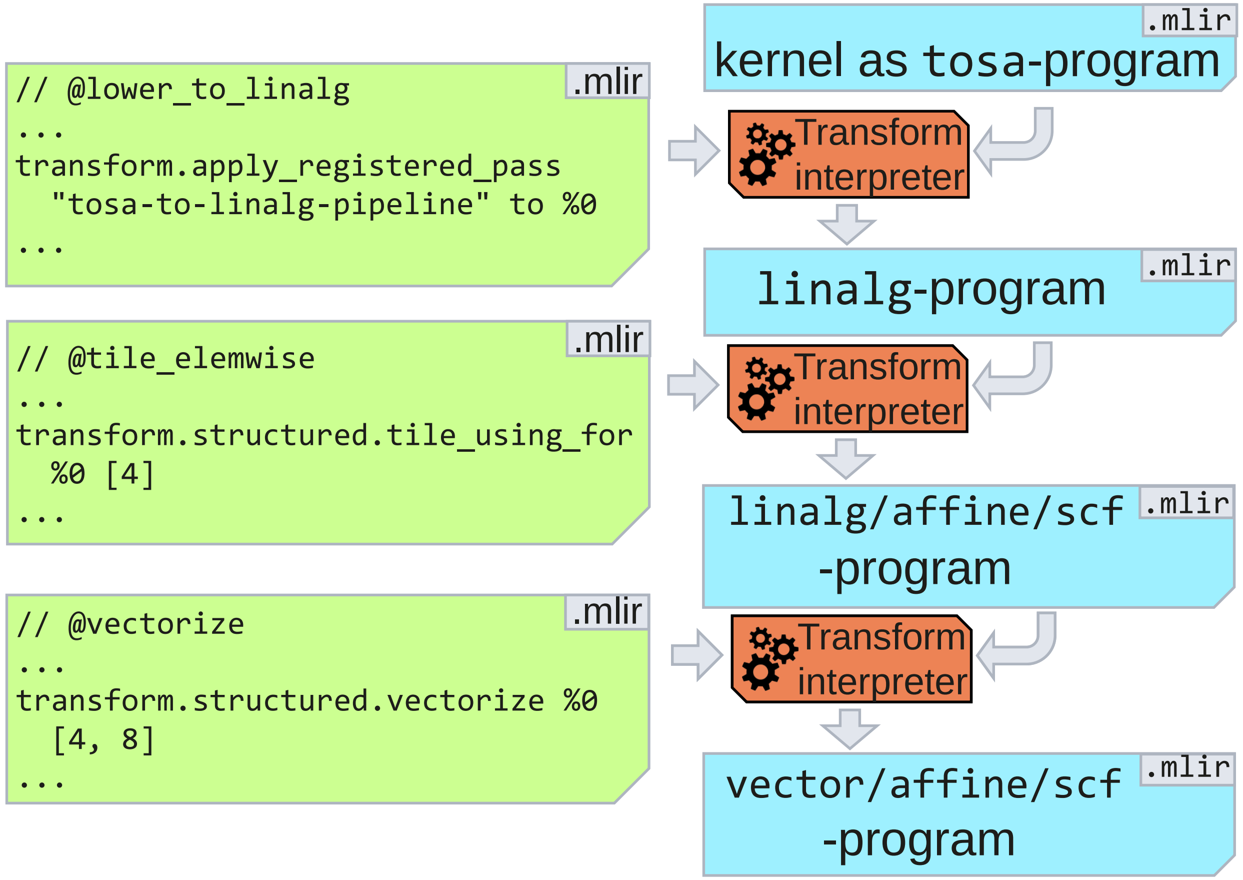
.mlir
```
// @tile_elemwise
...
transform.structured.tile_using_for
  %0 [4]
...
```

.mlir
```
// @vectorize
...
transform.structured.vectorize %0
  [4, 8]
...
```



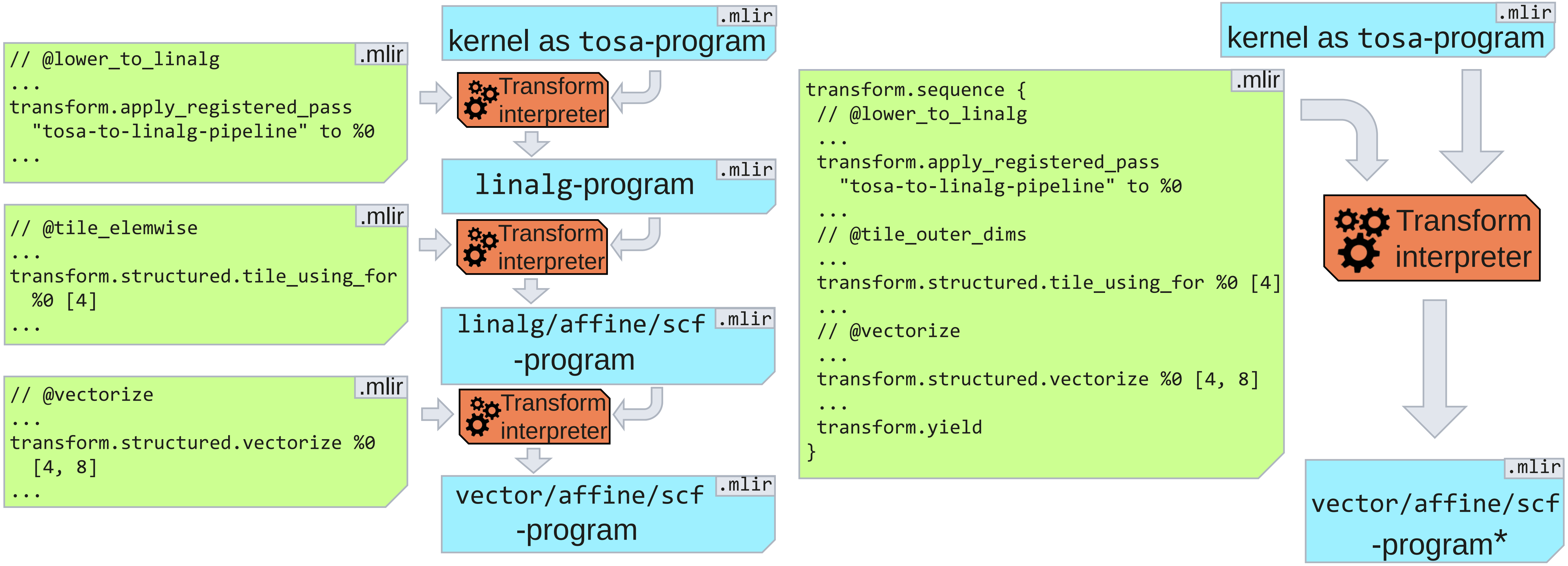
What about composing (Transform-dialect) schedules?

Monolithic Transform-dialect schedules



*: might be a different program due to Transform Dialect's backtracking semantics

Monolithic Transform-dialect schedules



*: might be a different program due to Transform Dialect’s backtracking semantics

Composable schedules over monolithic schedules

Monolithic sequences are not ideal:

- Need to be programmatically generated
- Structure of the lowering pipeline is lost
- Harder to debug, maintain & reuse

Composing schedules:
Transform Dialect's **`named_sequences`** and **`include`**

.mlir

```
transform.named_sequence @tile_elemwise (%arg0)  {
  %elemwise = transform.structured.match attrs { iter_types = ["par"] } %arg0
  %tiled, %loop = transform.structured.tile_using_for %elemwise [4]
  transfrom.yield %loop
}

...
%matched = transform.structured.match %payload
%tiled_loop = transform.include @tile_elemwise failures(propagate) (%matched)
...
```

Composing schedules: main sequence calling other sequences

.mlir

```
module attributes {transform.with_named_sequence} {
  transform.named_sequence @lower_to_linalg(%mod) -> !transform.any_op {
    ...
    %transformed_mod = ...
    transform.yield %transformed_mod
  }
  ...
  transform.named_sequence @tile_elemwise(%mod) -> !transform.any_op { ... }
  transform.named_sequence @vectorize(%mod) -> !transform.any_op { ... }
  ...
  transform.named_sequence @__transform_main(%payload) {
    %mod1 = transform.include @lower_to_linalg failures(propagate) (%payload)
    %mod2 = transform.include @tile_elemwise failures(propagate) (%mod1)
    %mod3 = transform.include @vectorize failures(propagate) (%mod2)
    ...
  }
}
```

Entire pipelines with reused schedules

.mlir
```
// kernel1 pipeline
transform.named_sequence
  @__transform_main(%mod) {
 transform.include @lower_to_linalg …
 transform.include @tile_elemwise …
 ...
 transform.include @vectorize …
 ...
 transform.include @lower_to_llvm …
}
```

LIB.mlir
```
transform.named_sequence
  @lower_to_linalg(%mod) -> !any_op {
 ...
 %transformed_mod = ...
 transform.yield %transformed_mod
}
transform.named_sequence
  @fuse_reductions(%mod) -> !any_op { ... }
transform.named_sequence
  @tile_elemwise(%mod) -> !any_op { ... }
transform.named_sequence
  @vectorize(%mod) -> !any_op { ... }
...
transform.named_sequence
  @lower_to_llvm(%mod) -> !any_op { ... }
```

.mlir
```
// kernel2 pipeline
transform.named_sequence
  @__transform_main(%mod) {
 transform.include @lower_to_linalg …
 transform.include @fuse_reductions
 ...
 transform.include @vectorize
 ...
 transform.include @lower_to_llvm …
}
```

kernel1 as tosa-program .mlir

Transform interpreter

kernel1 as llvm-program .mlir

kernel2 as tosa-program .mlir

Transform interpreter

kernel2 as llvm-program .mlir

Composing schedules … with glue: CSE and canonicalization

.mlir

```
transform.named_sequence @__transform_main(%tosa_mod) {
 ...
 %mod2 = transform.include @tile_elemwise failures(propagate) (%mod1)
 %mod2_postcse = transform.apply_cse to %mod2
 %mod2_postcanon = transform.apply_registered_pass "canonicalize" to %mod2_postcse
 %mod3_precse = transform.apply_cse to %mod2_postcanon
 %mod3 = transform.include @vectorize failures(propagate) (%mod3_precse)
 ...
}
```

So, the Transform Dialect's interpreter as a compiler?

Schedule-based MLIR-compiler

- We use MLIR’s Python bindings …

- … to generate schedules for each lowering / optimization step
  - … programmatically so that op attributes, e.g. tile sizes, get set appropriately
  - Mostly upstream Transform ops: less than a dozen are custom

- … programmatically generate a main sequence for each pipeline
  - Each pipeline lowers from high-level dialects all the way to the LLVM dialect
    … using just one schedule … composed of many small schedules

- … to delegate running of pipelines fully to the Transform interpreter
  - By invoking a single pass: `-transform-interpreter`

Schedule reuse in BLAS+-library

We have 19 distinct pipelines (i.e. main sequences) …

… which call out to 26 different stepwise schedules

- 15 stepwise schedules are used in 10+ pipelines

  … 4 of these make up a common suffix of all pipelines

- Around 80% of stepwise schedules have at most 7 Transform ops

- Only a couple main sequences have over 11 `transform.include` ops

  … including the common suffix

In summary

Schedules allow for declarative descriptions of lowering & optimization

- Transform Dialect allows writing schedules *for* MLIR *in* MLIR

By writing small composable schedules we can keep our compiler modular

- Small schedules facilitate reuse and maintainability

We can compose schedules for entire pipelines

… and delegate the actual work to the Transform interpreter

Next steps

- More than just linear pipelines
  - conditional execution: AlternativesOp and backtracking upon a match failure
  - One (DAG-shaped) schedule encompassing all pipelines

- Make the stepwise schedules take (e.g. tile size) parameters
  - Would allows for a static `.mlir` library of schedules vs. programmatically generating them
    - Transform ops mainly use attrs for parameters (which need to be statically known in MLIR)

- Infer properties of composed schedules, e.g.
  - Inferring overall parameter space for autotuning purposes