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)
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

\[
\text{blurx}(x,y) = \text{in}(x-1,y) + \text{in}(x,y) + \text{in}(x+1,y)
\]
\[
\text{out}(x,y) = \text{blurx}(x,y-1) + \text{blurx}(x,y) + \text{blurx}(x,y+1)
\]

program

\[
\text{par for} \quad \text{out.y}_o \text{ in } 0..\text{out.y.extent}/4
\]
\[
\text{for} \quad \text{out.x}_o \text{ in } 0..\text{out.x.extent}/4
\]
\[
\text{alloc} \quad \text{blurx}[[\text{blurx.y.extent}][\text{blurx.x.extent}]
\]
\[
\text{for} \quad \text{out.y}_i \text{ in } 0..4
\]
\[
\quad \text{let} \quad \text{blurx.y.min} = 4*\text{out.y}_o\text{.min} + \text{out.y}_i\text{.min} - 1
\]
\[
\quad \text{for} \quad \text{blurx.y} \text{ in } \text{blurx.y.min..\text{blurx.y.max}}
\]
\[
\quad \quad \text{for} \quad \text{blurx.x}_o \text{ in } \text{blurx.x.min/4..\text{blurx.x.max/4}}
\]
\[
\quad \quad \text{vec} \quad \text{for} \quad \text{blurx.x}_i \text{ in } 0..4
\]
\[
\quad \quad \quad \text{blurx}[[\text{blurx.y.stride*blurx.y+...}]] =
\]
\[
\quad \quad \quad \quad \text{in}[[\text{in.y.stride*(blurx.y.min+blurx.y)}
\]
\[
\quad \quad \quad \quad \quad +4*\text{blurx.x}_o\text{+ramp}(4)] + ...
\]
\[
\quad \quad \text{vec} \quad \text{for} \quad \text{out.x}_i \text{ in } 0..4
\]
\[
\quad \quad \quad \text{out}[[\text{out.y.stride*(4*(\text{out.y}_o-\text{out.y}.min)+\text{out.y}_i)+...}]
\]
\[
\quad \quad \quad \quad \text{=} \quad \text{blurx}[[\text{blurx.y.stride*(out.y}_i\text{.1-\text{blurx.y.min})}
\]
\[
\quad \quad \quad \quad \quad + \text{out.x}_i - \text{blurx.x.min}] + ...
\]

Halide (Ragan-Kelley et al, 2013)
**Image processing à la Halide**

**Algorithm** (high-level code)

\[
\text{blur}(x, y) = \text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y)
\]

\[
\text{out}(x, y) = \text{blur}(x, y-1) + \text{blur}(x, y) + \text{blur}(x, y+1)
\]

**Schedule**

- **blur**x: split x by 4 \(\rightarrow x_o, x_i\)
- vectorize: \(x_i\)
- store at \(\text{out}.x_0\)
- compute at \(\text{out}.y_i\)

- **out**: split x by 4 \(\rightarrow x_o, x_i\)
- split y by 4 \(\rightarrow y_o, y_i\)
- reorder: \(y_o, x_o, y_i, x_i\)
- parallelize: \(y_o\)
- vectorize: \(x_i\)

**Program** (lowered & optimized code)

- **par for** \(\text{out}.y_o\) in 0..\(\text{out}.y\).extent/4
- for \(\text{out}.x_o\) in 0..\(\text{out}.x\).extent/4
- alloc \(\text{blur}x[\text{blur}x.y.extent][\text{blur}x.x.extent]\)
- for \(\text{out}.y_i\) in 0..4
  - let \(\text{blur}x.y\).min = 4*\(\text{out}.y_o\).min + \(\text{out}.y_i\).min - 1
  - for \(\text{blur}x.y\) in \(\text{blur}x.y\).min..\(\text{blur}x.y\).max
    - for \(\text{blur}x.x_o\) in \(\text{blur}x.x\).min/4..\(\text{blur}x.x\).max/4
      - vec for \(\text{blur}x.x_i\) in 0..4
        - \(\text{blur}x[\text{blur}x.y\text{.stride}*\text{blur}x.y+...]\) =
          - \(\text{in}[\text{in}.y\text{.stride}*(\text{blur}x.y\text{.min}+\text{blur}x.y)]\) + ...
      - vec for \(\text{out}.x_i\) in 0..4
        - \(\text{out}[\text{out}.y\text{.stride}*(4*(\text{out}.y_o-\text{out}.y_o\text{.min})+\text{out}.y_i)+...]\) =
          - \(\text{blur}x[\text{blur}x.y\text{.stride}*(\text{out}.y_i-1-\text{blur}x.y\text{.min})\) + \(\text{out}.x_i - \text{blur}x.x\text{.min}\) + ...

Halide (Ragan-Kelley et al, 2013)
Many scheduling DSLs

... in support of optimizing BLAS / tensor programs:

Tensor Comprehensions (Vasilache et al, 2018)

Fireiron (Hagedorn et al, 2020)

TVM (Chen et al, 2018)

TTile (Tollenaere et al, 2021)

Tiramisu (Baghdadi et al, 2019)

+ Loop Tiling

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:**

```mlir
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>
}
```

**transform IR:**

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

```

**transformed IR:**

```mlir
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_slice_elem, %y_slice_elem linalg.yield %βy_slice_elem : f32 } -> tensor<?xf32> %in_slice = tensor.insert_slice ... scf.yield %in_slice } ...

```

**transform interpreter**

The transform interpreter processes the *transform IR* to produce the *transformed IR*.
payload IR and transform IR:

```
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
  ...
}
```

transformed IR:

```
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
    }
    ...
  }
}
```
How about combining schedules and multi-level rewriting?
Progressive lowering/optimization through the dialects

(kernel as tosa-program)

(Quinn Dawkins, 2022)
Progressive lowering/optimization through the dialects

kernel as tosa-program
@lower_to_linalg
linalg-program

(Quinn Dawkins, 2022)
Progressive lowering/optimization through the dialects

(kernel as tosa-program)

@lower_to_linalg

(linalg-program)

@tile_elemwise

(linalg/affine/scf-program)

(Quinn Dawkins, 2022)
Progressive lowering/optimization through the dialects

(kernel as tosa-program)
@lower_to_linalg

(linalg-program)
@tile_elemwise

(linalg/affine/scf-program)
@vectorize

(vector/affine/scf-program)

(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 ...

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

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

kernel as tosa-program

linalg-program

linalg/affine/scf-program

vector/affine/scf-program
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

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

```
// 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 ...
}
```

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

```
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