Transform Interfaces RFC

Alex Zinenko, <zinenko@google.com>
2022-03-17
Motivation

Input: 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)

Input: Schedule
blurx: split x by 4 to x₀, x₁
vectorize: x₁
store at out x₀
compute at out y₁
out: split x by 4 to x₀, x₁
split y by 4 to y₀, y₁
reorder: y₀, x₀, y₁, x₁
parallelize: y₀
vectorize: x₁

TVM (Chen et.al. 2018)

tc::IslKernelOptions::makeDefaultM
.scheduleSpecialize(false)
.tile({4, 32})
.mapToThreads({1, 32})
.mapToBlocks({64, 128})
.useSharedMemory(true)
.usePrivateMemory(true)
.unrollCopyShared(false)
.unroll(4);

TC (Vasilache et.al. 2018)

Halide (Ragan-Kelley et.al. 2013)

Fireiron (Hagedorn et.al. 2020)
Motivation

- URUK (Girbal et al. 2006)
- Omega (Pugh, 1991)

Distribution Distribute loop at depth $L$ over the statements $D$, with statement $s_j$ going into $r_p$'th loop.

Requirements: $V_{s_j}, s_j \in B \land s_j \in D \Rightarrow \text{loop}(f_p) \land L \leq \text{col}(s_j)$

Transformation: $V_{s_j} \in D$, replace $T_p$ by $[f_1, \ldots, f_p, \text{syntactic}(r_p), f_{p+1}, \ldots, f_n]$

Statement Reordering Reorder statements $D$ at level $L$ so that new position of statement $s_j$ is $r_p$.

Requirements: $V_{s_j}, s_j \in B \land s_j \in D \Rightarrow \text{syntactic}(f_p) \land L \leq \text{col}(s_j)$

Transformation: $V_{s_j} \in D$, replace $T_p$ by $[f_1, \ldots, f_{p-1}, f_p, \text{syntactic}(r_p), f_{p+1}, \ldots, f_n]$

Fusion Fuse the loops at level $L$ for the statements $D$ with statement $s_j$ going into the $r_p$'th loop.

Requirements: $V_{s_j}, s_j \in B \land s_j \in D \Rightarrow \text{syntactic}(f_{L-1}) \land \text{loop}(f_p) \land L \leq \text{col}(s_j)$

Transformation: $V_{s_j} \in D$, replace $T_p$ by $[f_1, \ldots, f_{L-1}, f_p, \text{syntactic}(r_p), f_{L+1}, \ldots, f_n]$

Unimodular Transformation Apply a $k \times k$ unimodular transformation $T$ to a perfectly nested loop containing statements $D$ at depth $L$. $L \leq k$. Note: Unimodular transformations include loop interchange, skewing and reversal [Bar90, W199b].

Requirements: $\forall i, j, s_j \in D \land s_j \in D \Rightarrow \text{col}(f_p) \land L \leq k$ $\Rightarrow \text{col}(s_j)$

Transformation: $V_{s_j} \in D$, replace $T_p$ by $[f_1, \ldots, f_{L-1}, f_p, f_{L+1}, \ldots, f_n]$

Strip-mining Strip-mine the loop at level $L$ for statements $D$ with block size $B$.

Requirements: $V_{s_j}, s_j \in D \land s_j \in D \Rightarrow \text{loop}(f_p) \land \text{col}(s_j)$

Transformation: $V_{s_j} \in D$, replace $T_p$ by $[f_1, \ldots, f_{L-1}, f_p, f_{L+1}, \ldots, f_n]$

Index Set Splitting Split the index set of statements $D$ using condition $C$.

Requirements: $C$ affinely express a symbolic constant and $C$ common to statements $D$.

Transformation: $V_{s_j} \in D$, replace $T_p$ by $(T_p \mid C)$ or $(T_p \mid \neg C)$
Motivation

- Multiple projects have consistently demonstrated state-of-the-art performance results by using *schedule* representations separated from computation.

- Schedules allow for precise targeting of transformation: transform specific operations or operations with certain properties, e.g., loops with known large trip-count.

- Schedules reified as code or other exchange format allow for *externalization of heuristics*.

- Reified schedules can easily be generated via *autotuning* or composed by *experts*.
Goal of the proposal

- Provide a mechanism similar to schedules in MLIR to reap the same benefits as previous work at a larger scale.
  - Must be extensible in presence of custom (out-of-tree) dialects.
  - Reuse existing MLIR concepts.
  - Minimally intrusive in the infrastructure.
Challenges

- Chaining transformations / communicating between them, e.g., “tile the loop then unroll the resulting loops”.

- Possibility to analyze and simplify transformation “recipes”.

- Open set of transformations and rules (as opposed to parameterized heuristics).
Transformations as IR

PDL pattern matching operations

%initial = xform.match @pattern
%one:2 = xform.affine.tile %initial {sizes = [32,32]}
%two:2 = xform.affine.tile %one {sizes = [4,8]}

pdl.pattern @pattern : benefit(1) {
  %0 = // arbitrary matching of affine.for rewrite %0 with @xform
}

Handle to matched operations

Specific transform

Target

"Main" transformation result

Transform properties
Benefits of Transformations as IR

- Reproducible: can be stored, pre-computed outside the compilation flow, replayed, ...
- Verifiable: op verifiers just work, e.g., chained tile sizes are always decreasing.
- Simplifiable: canonicalizers just work, e.g., tile by 0/1 or unroll by 1 are no-ops.
- Better error reporting: pinpoint a transformation step that failed.

```plaintext
module @transform {
  xform.sequence {
    %0 = xform.match ...
    xform.transform %0 {options ...}
    // error: could not apply transformation
    // note: "attribute" prevents the transformation
  }
}
module @payload {
  "payload.op"() : () -> ()
  "payload.op"() {attribute} : () -> ()
  // note: targeted at this op
}
```
Structure of Transformations as IR

module @payload {
    affine.for %i = ... { }
    affine.for %j = ... { }
}

module @transform {
    %initial = xform.match @pattern
    %one:2 = xform.affine.tile %initial {sizes = [32,32]}
    %two:2 = xform.affine.tile %one {sizes = [4,8]}
}

- Nested in the same top-level module.
- Or two separate modules consumed by a tool.
Some possibilities: combinators

Similar to LIFT/RISE

```plaintext
%0 = xform.match @pattern
%1 = xform.try {
  %2 = xform.one.transformation %0
  xform.yield %2
  // If any previous step failed, fallthrough.
} else {
  %2 = xform.another.transformation %0
  xform.yield %2
  // If any previous step failed, fallthrough.
} else {
  xform.yet.another %0
  xform.yield %2
}
```
Some possibilities: typed handles

\%
\%0 = xform.match @pattern : !xform.loop
\%1 = xform.loop.tile \%0 // okay
\%2 = xform.func.outline \%0 // error
Organization

- A dialect, “xform” or “transform” containing common ops and utilities.
- An interface `TransformOpInterface` in this dialect with:

  ```
  virtual LogicalResult apply(TransformState &state) = 0;
  ```

  ```
  struct TransformState {
    void setPayload(OpResult handle, ArrayRef<Operation *> payloadIROps);
    ArrayRef<Operation *> getPayload(Value operand) const;

    // Listeners?
  };
  ```

- Traits to handle common cases such as single-operand single-result ops.
Layering: help wanted!

- Having “xform” dialect depend on all possible dialects / their transforms creates unnecessary coupling.
- So does having every dialect depend on “xform”.

- Create new dialects to contain transformations?
- Somehow inject operations into the “xform” dialect without build-time dependency?