Google

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 → x_o, x_i
vectorize: x_i
store at out.x_o
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

Halide (Ragan-Kelley et.al. 2013)

+ 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]
```

+ Cache Data on Accelerator Special Buffer

```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ Map to Accelerator Tensor Instructions

```
s[CL].tensorize(yi, vdla.gemm8x8)
```

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)

```
mm = MatMul(M,N,K)(GL,GL,GL)(Kernel)
mm                  // resulting intermediate specs below
.tile(128,128)      // MatMul(128,128,K)(GL,GL,GL)(Kernel)
  .to(Block)        // MatMul(128,128,K)(GL,GL,GL)(Block )
.load(A, SH, _)     // MatMul(128,128,K)(SH,GL,GL)(Block )
.load(A, SH, _)     // MatMul(128,128,K)(SH,SH,GL)(Block )
.tile(64,32)        // MatMul(64, 32, K)(SH,SH,GL)(Block )
  .to(Warp)         // MatMul(64, 32, K)(SH,SH,GL)(Warp  )
.tile(8,8)          // MatMul(8,  8,  K)(SH,SH,GL)(Warp  )
  .to(Thread)       // MatMul(8,  8,  K)(SH,SH,GL)(Thread)
.load(A, RF, _)     // MatMul(8,  8,  K)(RF,SH,GL)(Thread)
.load(B, RF, _)     // MatMul(8,  8,  K)(RF,RF,GL)(Thread)
.tile(1,1)          // MatMul(1,  1,  K)(RF,RF,GL)(Thread)
.done(dot.cu)       // invoke codegen, emit dot micro-kernel
```

Fireiron (Hagedorn et.al. 2020)

Motivation

```
# Avoid spurious versioning
addContext(C1L1,'ITMAX>=9')
addContext(C1L1,'doloop_ub>=ITMAX')
addContext(C1L1,'doloop_ub<=ITMAX')
addContext(C1L1,'N>=500')
addContext(C1L1,'M>=500')
addContext(C1L1,'MNMIN>=500')
addContext(C1L1,'MNMIN<=M')
addContext(C1L1,'MNMIN<=N')
addContext(C1L1,'M<=N')
addContext(C1L1,'M>=N')

# Move and shift calc3 backwards
shift(enclose(C3L1),{'1','0','0'})
shift(enclose(C3L10),{'1','0'})
shift(enclose(C3L11),{'1','0'})
shift(C3L12,{'1'})
shift(C3L13,{'1'})
shift(C3L14,{'1'})
shift(C3L15,{'1'})
shift(C3L16,{'1'})
shift(C3L17,{'1'})
motion(enclose(C3L1),BLOOP)
motion(enclose(C3L10),BLOOP)
motion(enclose(C3L11),BLOOP)
motion(C3L12,BLOOP)
motion(C3L13,BLOOP)
motion(C3L14,BLOOP)
motion(C3L15,BLOOP)
motion(C3L16,BLOOP)
motion(C3L17,BLOOP)
```

```
# Peel and shift to enable fusion
peel(enclose(C3L1,2),'3')
peel(enclose(C3L1_2,2),'N-3')
peel(enclose(C3L1_2_1,1),'3')
peel(enclose(C3L1_2_1_2,1),'M-3')
peel(enclose(C1L1,2),'2')
peel(enclose(C1L1_2,2),'N-2')
peel(enclose(C1L1_2_1,1),'2')
peel(enclose(C1L1_2_1_2,1),'M-2')
peel(enclose(C2L1,2),'1')
peel(enclose(C2L1_2,2),'N-1')
peel(enclose(C2L1_2_1,1),'3')
peel(enclose(C2L1_2_1_2,1),'M-3')
shift(enclose(C1L1_2_1_2_1),{'0','1','1'})
shift(enclose(C2L1_2_1_2_1),{'0','2','2'})

# Double fusion of the three nests
motion(enclose(C2L1_2_1_2_1),TARGET_2_1_2_1)
motion(enclose(C1L1_2_1_2_1),C2L1_2_1_2_1)
motion(enclose(C3L1_2_1_2_1),C1L1_2_1_2_1)

# Register blocking and unrolling (factor 2)
time_stripmine(enclose(C3L1_2_1_2_1,2),2,2)
time_stripmine(enclose(C3L1_2_1_2_1,1),4,2)
interchange(enclose(C3L1_2_1_2_1,2))
time_peel(enclose(C3L1_2_1_2_1,3),4,'2')
time_peel(enclose(C3L1_2_1_2_1_2,3),4,'N-2')
time_peel(enclose(C3L1_2_1_2_1_2_1,1),5,'2')
time_peel(enclose(C3L1_2_1_2_1_2_1_2,1),5,'M-2')
fullunroll(enclose(C3L1_2_1_2_1_2_1_2,2))
fullunroll(enclose(C3L1_2_1_2_1_2_1_2,1))
```

URUK (Girbal et.al. 2006)

Distribution Distribute loop at depth L over the statements D, with statement s_p going into ${r_p}^{th}$ loop.

Requirements: $\forall s_p, s_q \;\; s_p \in D \land s_q \in D \Rightarrow \text{loop}(f_p^L) \land L \leq \text{csl}(s_p, s_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, \text{syntactic}(r_p), f_p^L, \ldots, f_p^n]$

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

Requirements: $\forall s_p, s_q \;\; s_p \in D \land s_q \in D \Rightarrow \text{syntactic}(f_p^L) \land L \leq csl(s_p, s_q) + 1 \land$
$(L \leq csl(s_p, s_q) \Leftrightarrow r_p = r_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, \text{syntactic}(r_p), f_p^{(L+1)}, \ldots, f_p^n]$

Fusion Fuse the loops at level L for the statements D with statement s_p going into the ${r_p}^{th}$ loop.

Requirements: $\forall s_p, s_q \; s_p \in D \land s_q \in D \Rightarrow \text{syntactic}(f_p^{(L-1)}) \land \text{loop}(f_p^L) \land L - 2 \leq csl(s_p, s_q) + 2 \land$
$(L - 2 < csl(s_p, s_q) + 2 \Rightarrow r_p = r_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-2)}, \text{syntactic}(r_p), f_p^{(L)}, f_p^{(L-1)}, f_p^{(L+1)}, \ldots, f_p^n]$

Unimodular Transformation Apply a $k \times k$ unimodular transformation U to a perfectly nested loop containing statements D at depth $L \ldots L + k$. Note: Unimodular transformations include loop interchange, skewing and reversal [Ban90, WL91b].

Requirements: $\forall i, s_p, s_q \;\; s_p \in D \land s_q \in D \land L \leq i \leq L + k \Rightarrow \text{loop}(f_p^i) \land L + k \leq csl(s_p, s_q))$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, U[f_p^{(L)}, \ldots f_p^{(L+k)}]^\top, f_p^{(L+k+1)}, \ldots, f_p^n]$

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

Requirements: $\forall s_p, s_q \;\; s_p \in D \land s_q \in D \Rightarrow \text{loop}(f_p^L) \land L \leq csl(s_p, s_q)) \land B$ is a known integer constant

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, B(f_p^{(L)} \text{ div } B), f_p^{(L)}, \ldots, f_p^n]$

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

Requirements: C is affine expression of symbolic constants and indexes common to statements D.

Transformation: $\forall s_p \in D$, replace T_p by $(T_p \mid C) \cup (T_p \mid \neg C)$

Omega (Pugh, 1991)

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

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]}
```

Specific transform

Target

Transform properties

Handle to matched operations

"Main" transformation result

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.

```
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
}
```

Google

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

```
%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 `TranfsormOpInterface` 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?