TVM: ALGORITHM & FRONT-END IMPLEMENTATION

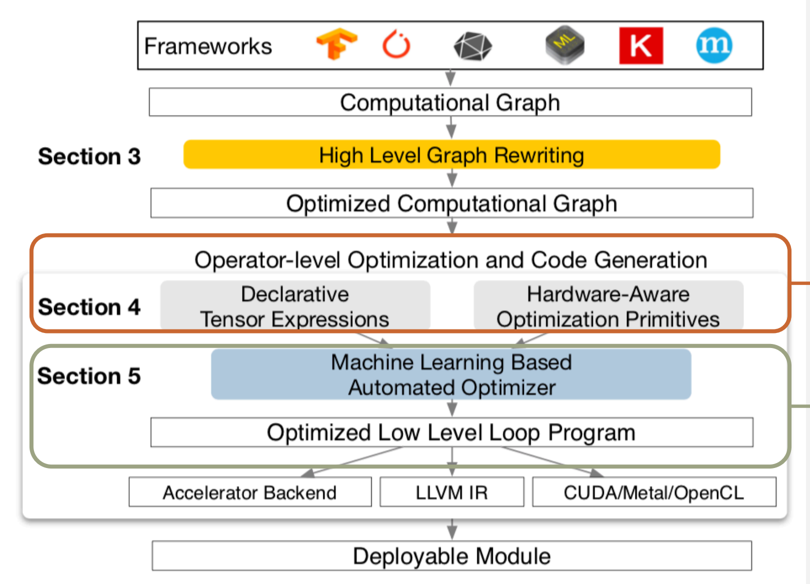
Rundong Li

14th May, 2019

OUTLINE

- Decouple Algorithm and Schedule:
  - (Auto)TVM: T. Chen, et al., OSDI'18 & NIPS'18
  - (*) Halide Language: J. Ragan-Kelley, et al., PLDI'13
- Systematic Optimization:
  - AutoTVM: T. Chen, L. Zheng, et al., NIPS'18
  - (*) VTA: T. Moreau, T. Chen, et al., Tech report
- TVM Code Reading (`master: 4332b0aa`)

TVM STACK



Similar to Halide: for each operator, its algorithm and schedule are decoupled:

- Algorithm: description ("how");
- Schedule: rules for execution (parallel, vectorization, etc.);

Given algorithm e, compiler g and real-world evaluator f, search for the optimal schedule s such that minimize:

$$\arg\min_{s \in S_e} f(g(e, s))$$

OPERATOR OPTIMIZATION

ALGORITHM: e

```
m, n, h = t.var('m'), t.var('n'), t.var('h')
A = t.placeholder((m, h), name='A')
B = t.placeholder((n, h), name='B')
k = t.reduce_axis((0, h), name='k')
C = t.compute((m, n), lambda y, x:
        t.sum(A[k, y] * B[k, x], axis=k))
```

computing rule

result shape

Optimization logic:

- Given algorithm e;
- **Generate** schedule space $\mathcal{S}_e$;
- **Search for** best $s^* \in \mathcal{S}_e$;
- Compile bytecode $x = g(s^*, e)$;
- Such that real-world latency $f(x^*)$ is minimized;

** These are corresponding lower-level codes of schedules.*

*SCHEDULE "SPACE" OF e: $\mathcal{S}_e$

```
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]
```

```
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]
```

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

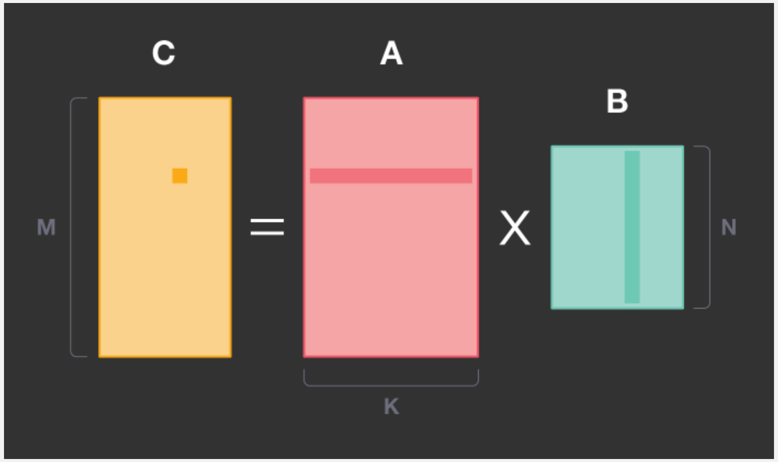
SPECIFY / GENERATE $\mathcal{S}_e$ & LOWERING

- Schedules can be assigned *manually* via high-level API, or be *optimized* by `autotvm` module;
- TVM performs optimized "lowering" (generate lower-level code, the `stmt` class), where schedules are preserved;
- Optimizations in lowering:
  - Nested Parallelism with Cooperation (blocking or tiling);
  - Tensorization;
  - Explicit Memory Latency Hiding;

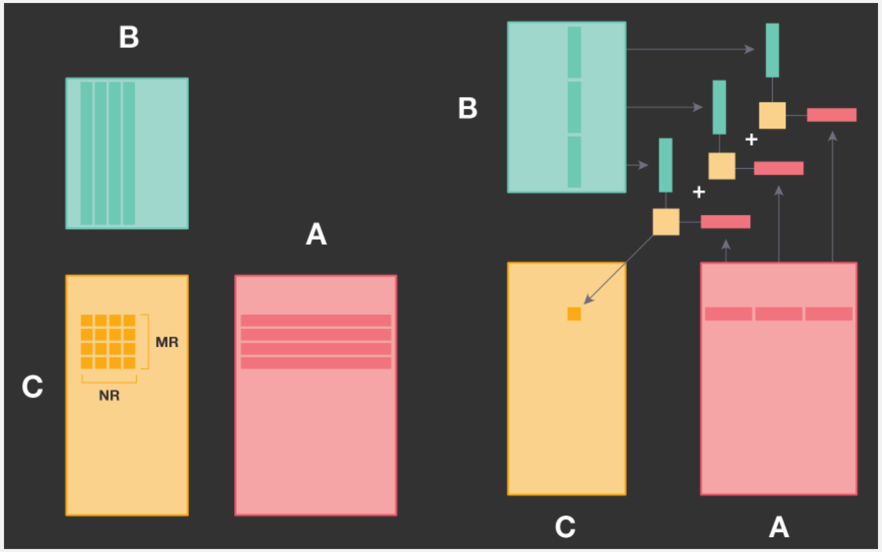
```python
# Phase 0
if isinstance(sch, schedule.Schedule):
    stmt = form_body(sch)

for f in lower_phase0:
    stmt = f(stmt)
# Phase 1
stmt = ir_pass.StorageFlatten(stmt, binds, 64, cf
stmt = ir_pass.CanonicalSimplify(stmt)
for f in lower_phase1:
    stmt = f(stmt)
# Phase 2
if not simple_mode:
    stmt = ir_pass.LoopPartition(stmt, cfg.partit
stmt = ir_pass.VectorizeLoop(stmt)
stmt = ir_pass.InjectVirtualThread(stmt)
stmt = ir_pass.InjectDoubleBuffer(stmt, cfg.doubl
stmt = ir_pass.StorageRewrite(stmt)
stmt = ir_pass.UnrollLoop(
    stmt,
    cfg.auto_unroll_max_step,
    cfg.auto_unroll_max_depth,
    cfg.auto_unroll_max_extent,
    cfg.unroll_explicit)
for f in lower_phase2:
    stmt = f(stmt)
# Phase 3
stmt = ir_pass.Simplify(stmt)
stmt = ir_pass.LowerStorageAccessInfo(stmt)
stmt = ir_pass.RemoveNoOp(stmt)
if not cfg.disable_select_rewriting:
    stmt = ir_pass.RewriteUnsafeSelect(stmt)
for f in lower_phase3:
    stmt = f(stmt)
# Instrument BoundCheckers
if cfg.instrument_bound_checkers:
    stmt = ir_pass.InstrumentBoundCheckers(stmt)
if simple_mode:
    return stmt
```

OPTIMIZE GEMM: BLOCKING

VANILLA GEMM:
MEMORY BOUNDED



BLOCKED GEMM:
MORE CACHE FRIENDLY



SPECIFY s: BLOCKING

MANUALLY SPECIFY SCHEDULE

```
bn = 32
s = tvm.create_schedule(C.op)

# Blocking by loop tiling
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
k, = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)

# Hoist reduction domain outside the blocking loop
s[C].reorder(xo, yo, ko, ki, xi, yi)

func = tvm.build(s, [A, B, C], target=target, name='mmult')
```

CORRESPONDING BYTECODE

```
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]
```

GENERATE $\mathcal{S}_e$: BLOCKING

SPECIFY **TEMPLATE** SCHEDULE…

```python
@autotvm.template  # 1. use a decorator
def matmul_v1(N, L, M, dtype):
    A = tvm.placeholder((N, L), name='A', dtype=dtype)
    B = tvm.placeholder((L, M), name='B', dtype=dtype)

    k = tvm.reduce_axis((0, L), name='k')
    C = tvm.compute((N, M), lambda i, j: tvm.sum(A[i, k]
    s = tvm.create_schedule(C.op)

    # schedule
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    # 2. get the config object
    cfg = autotvm.get_config()

    # 3. define search space
    cfg.define_knob("tile_y", [1, 2, 4, 8, 16])
    cfg.define_knob("tile_x", [1, 2, 4, 8, 16])

    # 4. schedule according to config
    yo, yi = s[C].split(y, cfg['tile_y'].val)
    xo, xi = s[C].split(x, cfg['tile_x'].val)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
```

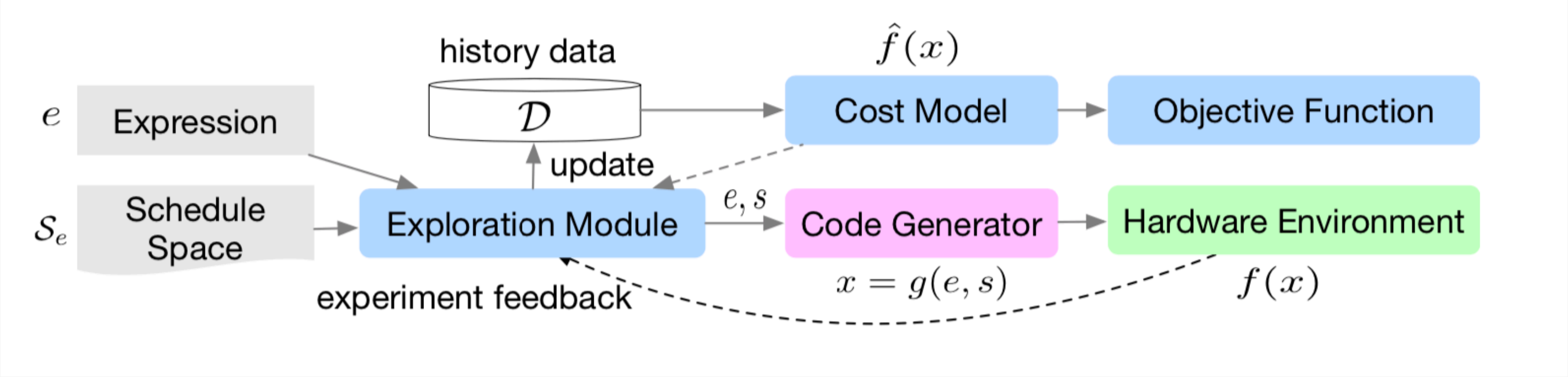
… THEN TUNED BY **AUTOTVM**

```python
N, L, M = 512, 512, 512
task = autotvm.task.create(matmul, args=(N, L, M, 'float32'), target=
measure_option = autotvm.measure_option(
    builder='local',
    runner=autotvm.LocalRunner(number=5))

# begin tuning, log records to file `matmul.log`
tuner = autotvm.tuner.RandomTuner(task)
tuner.tune(n_trial=10,
           measure_option=measure_option,
           callbacks=[autotvm.callback.log_to_file('matmul.log')])
```

The template is "search space" $\mathcal{S}_e$

- Define "knob"s manually, i.e. specify all candidates;
- Or let `autotvm` extract all tunable "knob"s from template!

SEARCH $\mathcal{S}_e$: COST MODEL + SIMULATED ANNEALING



- Cost module: Gradient Boost Trees (GBT, based on xgboost);
- Features: manually selected,
    - Loop structure information (e.g. memory access count and data reuse ratio);
    - Generic annotations (e.g. vectorization, unrolling, thread binding);
- Object function: rank based (x_i is slower or faster than x_j)

$$\sum_{i,j} \log(1 + e^{-\operatorname{sign}(c_i - c_j)(\hat{f}(x_i) - \hat{f}(x_j))})$$

SEARCH $\mathcal{S}_e$: COST MODEL + SIMULATED ANNEALING

Algorithm 1: Learning to Optimize Tensor Programs

Input : Transformation space $\mathcal{S}_e$
Output : Selected schedule configuration s^*
$\mathcal{D} \leftarrow \emptyset$
while *n_trials* $<$ *max_n_trials* **do**

```
// Pick the next promising batch
```

$Q \leftarrow$ run parallel simulated annealing to collect candidates in $\mathcal{S}_e$ using energy function $\hat{f}$
$S \leftarrow$ run greedy submodular optimization to pick $(1-\epsilon)b$-subset from Q by maximizing Equation 3
$S \leftarrow S \cup \{$ Randomly sample ϵb candidates. $\}$

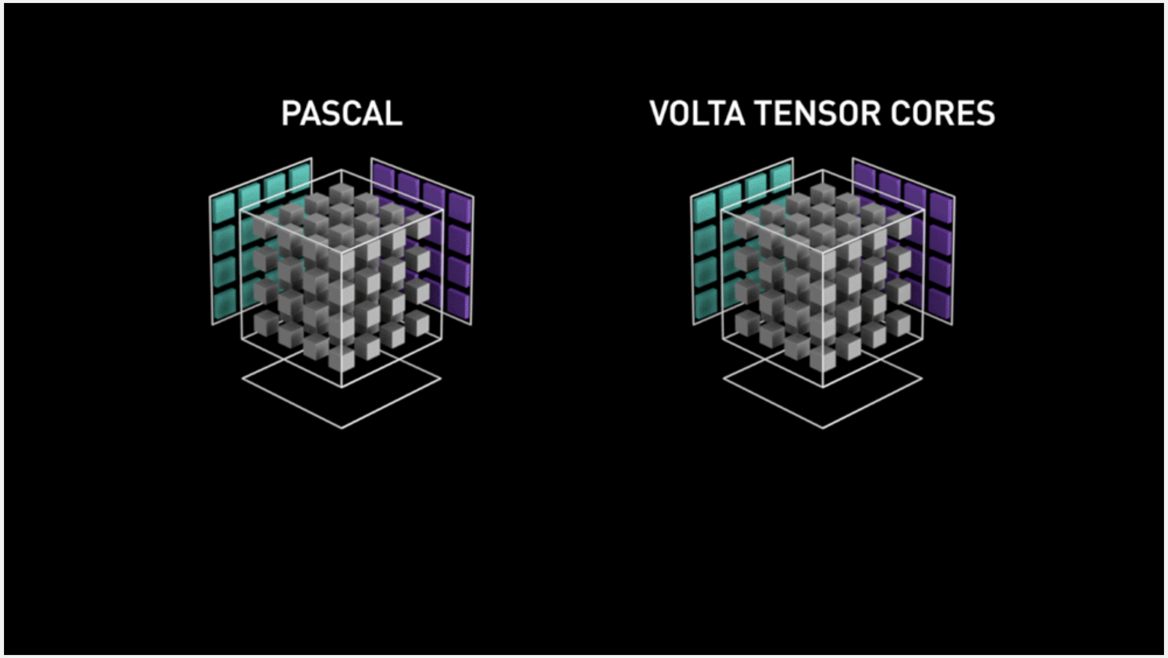
```
// Run measurement on hardware environment
```

for s ***in*** S **do**
$c \leftarrow f(g(e, s)); \mathcal{D} \leftarrow \mathcal{D} \cup \{(e, s, c)\}$
end

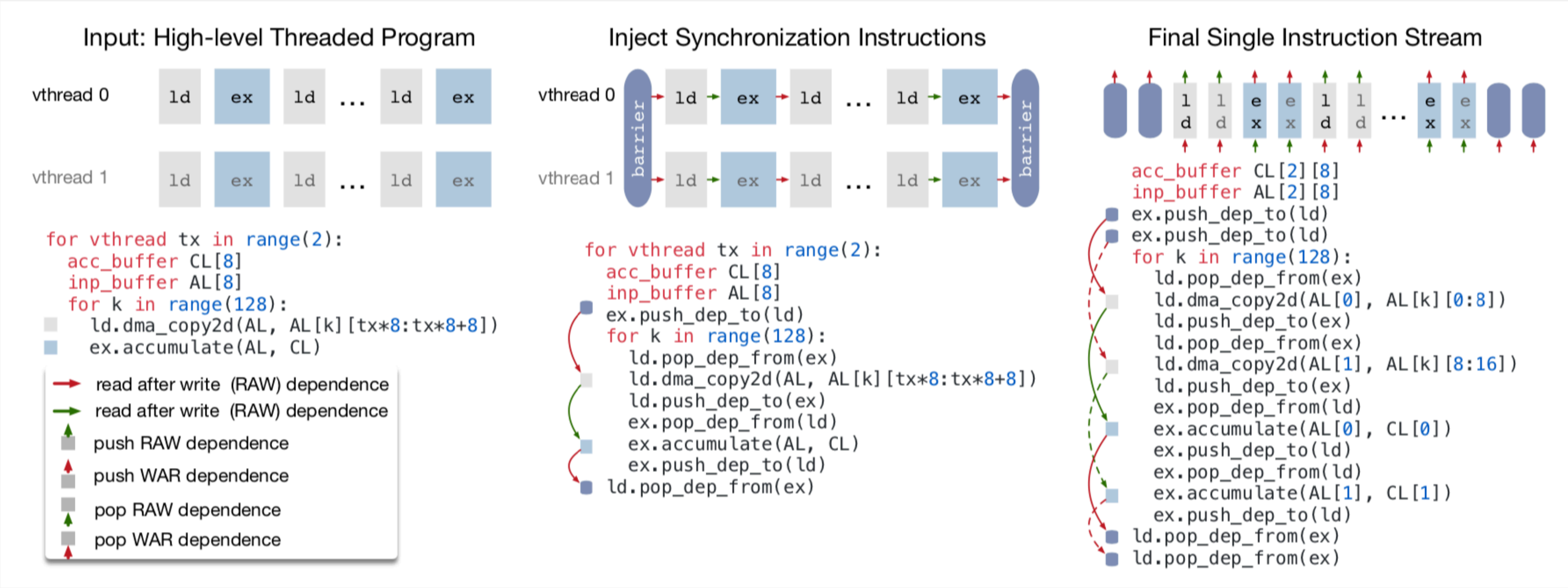
```
// Update cost model
```

update $\hat{f}$ using $\mathcal{D}$
n_trials $\leftarrow$ n_trials $+ b$
end
$s^* \leftarrow$ history best schedule configuration

OTHER OPT.: TENSORIZATION



OTHER OPT.: VIRTUAL THREADS



- User write multi-thread schedules via high-level APIs;
- TVM automatically synchronize these virtual threads and compile to serialized bytecode;