A Term Rewriting Path to High-Performance

Thomas Kœhler Phil Trinder Michel Steuwer

University of Glasgow THE UNIVERSITY of EDINBURGH

INRIA CAMUS, Strasbourg — August 2022

Optimizing Low-Level Code is Hard

- hand optimization is time-consuming and error-prone

  e.g. in C, OpenCL, CUDA
- critical in performance-demanding domains

  e.g. image processing, numeric simulation, machine learning
- typically leads to orders of magnitude performance improvements

Optimizing Matrix Multiplication in C

```c
for (int im = 0; im < m; im++) {
  for (int in = 0; in < n; in++) {
    float acc = 0.0f;
    for (int ik = 0; ik < k; ik++) {
      acc += a[ik + (k * im)] * b[in + (n * ik)];
    }

    output[in + (n * im)] = acc;
  }
}
```

Optimized program on the right:

- **+** 110× faster runtime

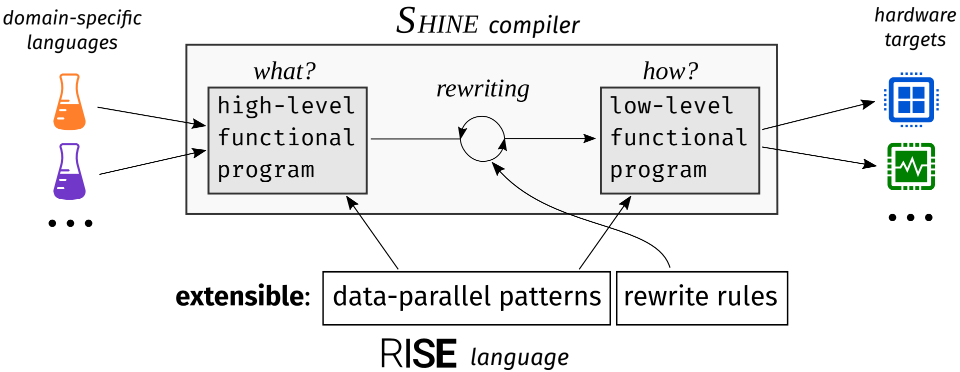
  Intel i5-4670K CPU

- **-** 6× more lines, more complex code

  threads, SIMD, indexing

```c
float aT[n * k];
#pragma omp parallel for
for (int in = 0; in < (n / 32); in = 1 + in) {
  for (int ik = 0; ik < k; ik = 1 + ik) {
    #pragma omp simd
    for (int jn = 0; jn < 32; jn = 1 + jn) {
      aT[(ik + ((32 * in) * k)) + (jn * k)] = a[(jn + (32 * in)) + (ik * n)];
    }
  }
}
#pragma omp parallel for
for (int im = 0; im < (m / 32); im = 1 + im) {
  for (int in = 0; in < (n / 32); in = 1 + in) {
    float tmp1[1024];
    for (int jm = 0; jm < 32; jm = 1 + jm) {
      for (int jn = 0; jn < 32; jn = 1 + jn) {
        tmp1[jn + (32 * jm)] = 0.0f;
      }
    }
    for (int ik = 0; ik < (k / 4); ik = 1 + ik) {
      for (int jm = 0; jm < 32; jm = 1 + jm) {
        float tmp2[32];
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] = tmp1[jn + (32 * jm)];
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((4 * ik) + ((32 * im) * k)) + (jm * k)] * aT[((4 * ik) + ((32 * in) * k)) + (jn * k)]);
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((1 + (4 * ik)) + ((32 * im) * k)) + (jm * k)] *
            aT[((1 + (4 * ik)) + ((32 * in) * k)) + (jn * k)]);
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((2 + (4 * ik)) + ((32 * im) * k)) + (jm * k)] *
            aT[((2 + (4 * ik)) + ((32 * in) * k)) + (jn * k)]);
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((3 + (4 * ik)) + ((32 * im) * k)) + (jm * k)] *
            aT[((3 + (4 * ik)) + ((32 * in) * k)) + (jn * k)]);
        }
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp1[jn + (32 * jm)] = tmp2[jn];
        }
      }
    }
    for (int jm = 0; jm < 32; jm = 1 + jm) {
      for (int jn = 0; jn < 32; jn = 1 + jn) {
        output[((jn + ((32 * im) * n)) + (32 * in)) + (jm * n)] = tmp1[jn + (32 * jm)];
      }
    }
  }
}
```

Automating Optimization via Term Rewriting



+ convenient, hardware agnostic programming
+ high-performance code generation
+ extensible set of abstractions and optimizations

RISE is a Functional Array Language

High-level matrix multiplication in RISE:

```
def mm a b =
  map (λaRow.                     | for aRow in a:
    map (λbCol.                   |   for bCol in transpose(b):
      dot aRow bCol)              |     ... = dot(aRow, bCol)
      (transpose b)) a

def dot xs ys =
  reduce + 0                      | for (x, y) in zip(xs, ys):
    (map (λ(x, y). x × y)         |   acc += x × y
      (zip xs ys))
```

Rewrite Rules Encode Valid RISE Transformations

split-join:

```
   map f x                  | for m:
                            |  ... = f(...)
     ↦
join
  (map                      | for m / n:
    (map f)                 |  for n:
    (split n x))            |   ... = f(...)
```

transpose-around-map-map:

```
   map                      | for m:
     (map f) x              |  for n:
                            |   ... = f(...)
       ↦
transpose
  (map                      | for n:
    (map f)                 |  for m:
    (transpose x))          |   ... = f(...)
```

Complex Optimizations Emerge from Simple Rules

Matrix multiplication blocking in RISE:

```
map (λaRow.                 | for m:
  map (λbCol.               |  for n:
    dot aRow bCol)          |   for k:
    (transpose b)) a        |    ...
```

$\mapsto^*$

```
join (map (map join) (map transpose
  map                                   | for m / 32:
    (map λx2.                           |  for n / 32:
      reduceSeq (λx3. λx4.              |   for k / 4:
        reduceSeq λx5. λx6.             |    for 4:
          map                           |     for 32:
            (map (λx7.                  |      for 32:
             (fst x7) + (fst (snd x7)) ×
                (snd (snd x7)))
              (map (λx7. zip (fst x7) (snd x7))
                (zip x5 x6)))
         (transpose (map transpose
         (snd (unzip (map unzip map (λx5.
           zip (fst x5) (snd x5))
           (zip x3 x4)))))))
        (generate (λx3. generate (λx4. 0)))
        transpose (map transpose x2))
    (map (map (map (map (split 4))))
     (map transpose
       (map (map (λx2. map (map (zip x2)
         (split 32 (transpose b)))))
           split 32 a))))))
```

Deciding which Rewrites to Apply is Hard

automation ⟷ **control**

Equality Saturation	Sketch-Guided Equality Saturation	Rewriting Strategies

ELEVATE **Rewriting Strategies**

- programmers describe optimizations as compositions of rewrite rules
- MM blocking:

```
def blocking = ( baseline `;`
  tile(32,32)        `@` outermost(mapNest(2))   `;;`
  fissionReduceMap `@` outermost(appliedReduce)`;;`
  split(4)           `@` innermost(appliedReduce)`;;`
  reorder(List(1,2,5,6,3,4)))
```

\+ empowers programmers to manually control the rewrite process

\+ to define their own abstractions: `tile`, `split`, `reorder` are not built-in

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies". In: ICFP (2020)

ELEVATE **Rewriting Strategies**

- programmers describe optimizations as compositions of rewrite rules
- MM blocking:

```
def blocking = ( baseline `;`
  tile(32,32)        `@` outermost(mapNest(2))   `;;`
  fissionReduceMap `@` outermost(appliedReduce)`;;`
  split(4)           `@` innermost(appliedReduce)`;;`
  reorder(List(1,2,5,6,3,4)))
```

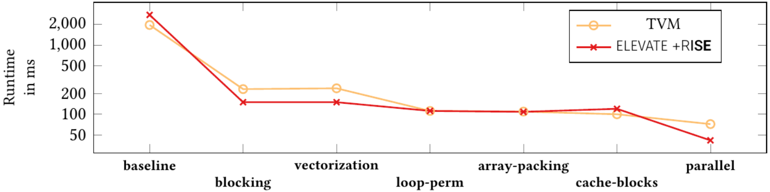
  - requires programmers to order all rewrite steps
  - strategies are often program-specific and tedious to implement

Achieving High-Performance with ELEVATE

Case Study 1) Matrix Multiplication Optimizations for Intel CPU

- Transform loops *blocking, permutation, unrolling*
- Change data layout *array packing*
- Add parallelism *vectorization, multi-threading*

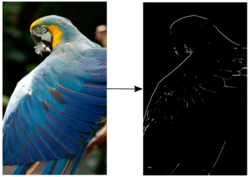
- Performance is on par with reference schedules from TVM

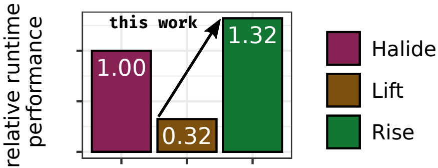


Achieving High-Performance with ELEVATE

Case Study 2) Harris Corner Detection Optimizations for ARM CPU

- 4 optimizations from reference Halide schedule
  circular buffering, operator fusion, multi-threading, vectorization
- 2 optimizations not supported by Halide *convolution separation, register rotation*

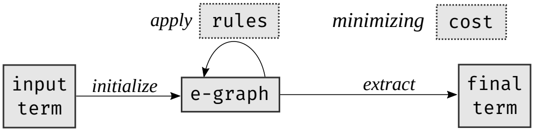




Thomas Koehler and Michel Steuwer. "Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs". In: *CGO*. 2021

Rewriting strategies achieve high-performance, but require **tedious manual rewrite ordering**

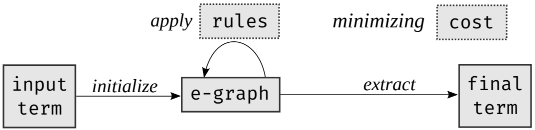
Equality Saturation



- Optimize programs by efficiently exploring many possible rewrites
- Many successful applications sparked from the recent `egg` library

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "egg: fast and extensible equality saturation". In: POPL (2021)

Equality Saturation



- Optimize programs by efficiently exploring many possible rewrites
- Many successful applications sparked from the recent `egg` library

No manual rewrite ordering, but **does not scale to the RISE case studies**

Sketch-Guided Equality Saturation

Observation:

- The *shape* of the optimized program is often used to explain optimizations:

```
for m:
 for n:
  for k:
   ..
```

$\mapsto^*$

```
for m / 32:
 for n / 32:
  for k / 4:
   for 4:
    for 32:
     for 32:
      ..
```

Sketch-Guided Equality Saturation

Observation:

- The *shape* of the optimized program is often used to explain optimizations:

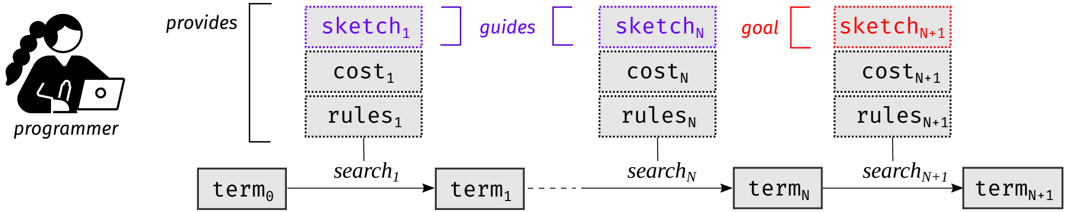
```
for m:
 for n:
  for k:
   ..
```

$\mapsto^*$

```
for m / 32:
 for n / 32:
  for k / 4:
   for 4:
    for 32:
     for 32:
      ..
```

Explanatory shapes can be formalized as sketches and used to guide rewriting

Sketch-Guided Equality Saturation



- Factors an unfeasible search into a sequence of feasible ones:
  1. Break long rewrite sequences
  2. Isolate explosive combinations of rewrite rules

https://arxiv.org/abs/2111.13040

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                  | for m:
 containsMap(n,                 |  for n:
  containsReduceSeq(k,          |   for k:
   containsAddMul)))            |     .. + .. × ..
```

- Abstractions defined in terms of smaller building blocks:

```
def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                          | for m:
 containsMap(n,                         |  for n:
  containsReduceSeq(k,                  |   for k:
   containsAddMul)))                    |     .. + .. × ..
```

- A sketch `s` is satisfied by a set of terms `R(s)`:

```
def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))

R(containsAddMul) = { R(app(app(+, ?), contains(×))) } ∪
  { F(t_1, .., t_n) | ∃t_i ∈ R(containsAddMul) }
R(app(app(+, ?), contains(×))) = { app(app(+, t_1), t_2) | t_2 ∈ R(contains(×)) }
R(contains(×)) = { × } ∪ { F(t_1, .., t_n) | ∃t_i ∈ R(contains(×)) }
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                  | for m:
 containsMap(n,                 |  for n:
  containsReduceSeq(k,          |   for k:
   containsAddMul)))            |     .. + .. × ..
```

blocking sketch:

```
containsMap(m / 32,             | for m / 32:
 containsMap(n / 32,            |  for n / 32:
  containsReduceSeq(k / 4,      |   for k / 4:
   containsReduceSeq(4,         |    for 4:
    containsMap(32,             |     for 32:
     containsMap(32,            |      for 32:
      containsAddMul))))))      |       .. + .. × ..
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                   | for m:
 containsMap(n,                  |  for n:
  containsReduceSeq(k,           |   for k:
   containsAddMul)))             |     .. + .. × ..
```

sketch guide:

how to split the loops before reordering them?

```
containsMap(m / 32,              | for m / 32:
 containsMap(32,                 |  for 32:
  containsMap(n / 32,            |   for n / 32:
   containsMap(32,               |    for 32:
    containsReduceSeq(k / 4,     |     for k / 4:
     containsReduceSeq(4,        |      for 4:
      containsAddMul))))))       |       .. + .. × ..
```

blocking sketch:

```
containsMap(m / 32,              | for m / 32:
 containsMap(n / 32,             |  for n / 32:
  containsReduceSeq(k / 4,       |   for k / 4:
   containsReduceSeq(4,          |    for 4:
    containsMap(32,              |     for 32:
     containsMap(32,             |      for 32:
      containsAddMul))))))       |       .. + .. × ..
```

Impact of Sketch-Guidance on MM Case Study

- Equality Saturation without Sketch Guides[2]:

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

- Sketch-Guided Equality Saturation[3]:

goal	sketch guides	found?	runtime	RAM
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

[2] Intel Xeon E5-2640 v2

[3] AMD Ryzen 5 PRO 2500U

Impact of Sketch-Guidance on MM Case Study

- Equality Saturation without Sketch Guides[2]:

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

- Sketch-Guided Equality Saturation[3]:

goal	sketch guides	found?	runtime	RAM
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

Sketch-guidance enables to find all 7 optimization goals

[2] Intel Xeon E5-2640 v2

[3] AMD Ryzen 5 PRO 2500U

Impact of Sketch-Guidance on MM Case Study

- Equality Saturation without Sketch Guides[2]:

goal	**found?**	**runtime**	**RAM**
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

582x

116x

- Sketch-Guided Equality Saturation[3]:

goal	**sketch guides**	**found?**	**runtime**	**RAM**
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

[2]Intel Xeon E5-2640 v2

[3]AMD Ryzen 5 PRO 2500U

Impact of Sketch-Guidance on MM Case Study

Sketches vs Full Program

all goals except *baseline*:

sketch guides	sketch goal	sketch sizes	program size
1-3	1	7-12	90-124

- sketches elide around 90% of the program
- sketches elide intricate details such as array reshaping patterns (e.g. `split`, `join`, `transpose`)

Conclusion

We talked about:

- The RISE language & SHINE compiler automating optimization via term rewriting
- ELEVATE *rewriting strategies* achieving high-performance by controlling rewriting
- *Sketch-guided equality saturation*, a novel, semi-automatic optimization technique

Conclusion

We talked about:

- The RISE language & SHINE compiler automating optimization via term rewriting
- ELEVATE *rewriting strategies* achieving high-performance by controlling rewriting
- *Sketch-guided equality saturation*, a novel, semi-automatic optimization technique

---

thomas.koehler@thok.eu
thok.eu

Thanks!

rise-lang.org
elevate-lang.org

Future Work

- Combine sketch-guided with rewriting strategies
- Apply sketch-guiding to more diverse applications
- Improve automated search (rewrite rule scheduling, heuristics, pruning, leverage hardware knowledge)
- Can we synthesize sketch guides from a sketch goal?
- Use in an interactive optimization assistant

Sketch Definition

$$S \;::=\; ? \;\;|\;\; F(S, .., S) \;\;|\;\; contains(S)$$

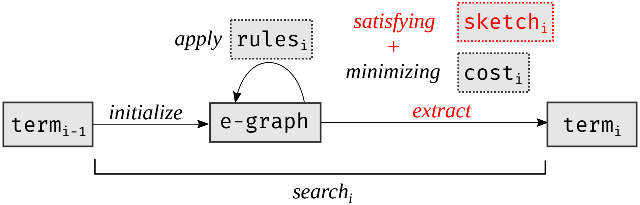
$$R(?) = T = \{F(t_1, .., t_n)\}$$

$$R(F(s_1, .., s_n)) = \{F(t_1, .., t_n) \mid t_i \in R(s_i)\}$$

$$R(contains(s)) = R(s) \cup \{F(t_1, .., t_n) \mid \exists t_i \in R(contains(s))\}$$

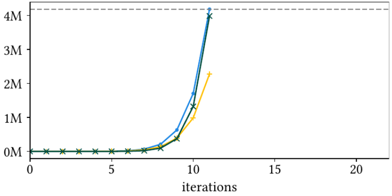
```
def containsMap(n: NatSketch, f: Sketch): Sketch =
  contains(app(map :: ?t → n.?dt → ?y, f))

def containsReduceSeq(n: NatSketch, f: Sketch): Sketch =
  contains(app(reduceSeq :: ?t → ?t → n.?dt → ?t, f))

def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))
```

Sketch-Satisfying Equality Saturation

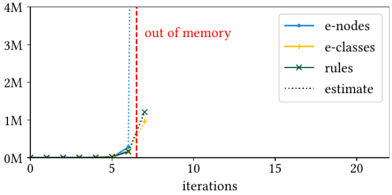


- Terminates as soon as a program satisfying the sketch is found

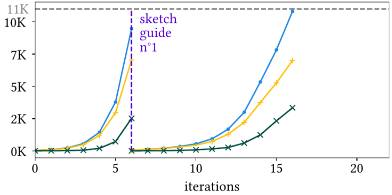
E-Graph Evolution



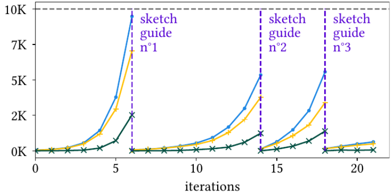
(a) unguided, *blocking*, found: ✓



(b) unguided, *parallel*, found: ✗

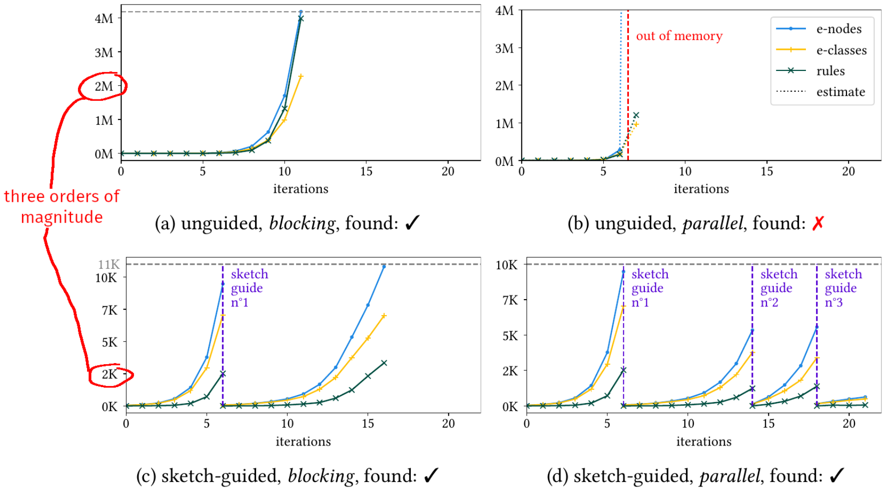


(c) sketch-guided, *blocking*, found: ✓

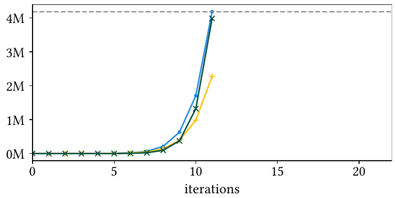


(d) sketch-guided, *parallel*, found: ✓

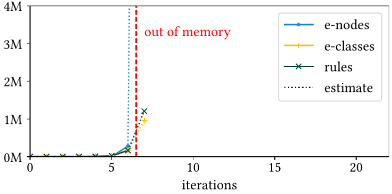
E-Graph Evolution



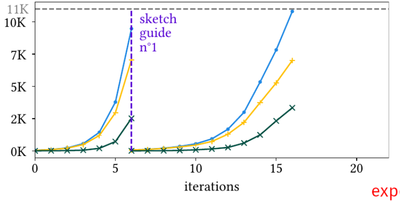
E-Graph Evolution



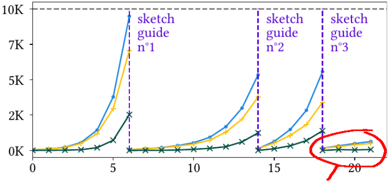
(a) unguided, *blocking*, found: ✓



(b) unguided, *parallel*, found: ✗



(c) sketch-guided, *blocking*, found: ✓



exponential growth, except linear here: no explosive rewrites

(d) sketch-guided, *parallel*, found: ✓

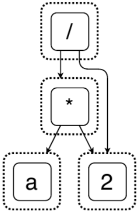
Sketches vs Full Program

goal	sketch guides	sketch goal	sketch sizes	program size
blocking	*split*	$reorder_1$	7	90
vectorization	*split* + $reorder_1$	$lower_1$	7	124
loop-perm	*split* + $reorder_2$	$lower_2$	7	104
array-packing	*split* + $reorder_2$ + *store*	$lower_3$	7-12	121
cache-blocks	*split* + $reorder_2$ + *store*	$lower_4$	7-12	121
parallel	*split* + $reorder_2$ + *store*	$lower_5$	7-12	121

- each sketch corresponds to a logical transformation step
- sketches elide around 90% of the program
- intricate details such as array reshaping patterns are not specified (e.g. `split`, `join`, `transpose`)

E-Graph Example

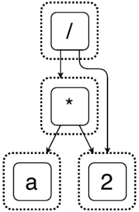
$$(a * 2)/2 \longrightarrow^{*} a$$

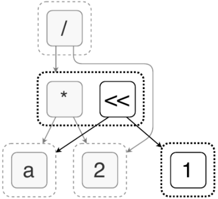


$(a * 2)/2$

E-Graph Example

$$(a * 2)/2 \longrightarrow^* a$$



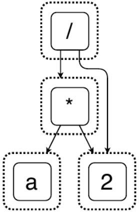


$(a * 2)/2$

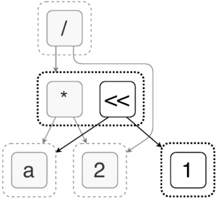
$x * 2 \longrightarrow x \ll 1$

E-Graph Example

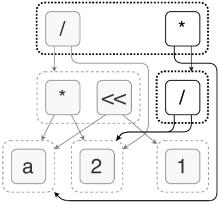
$$(a * 2)/2 \longrightarrow^* a$$



$(a * 2)/2$



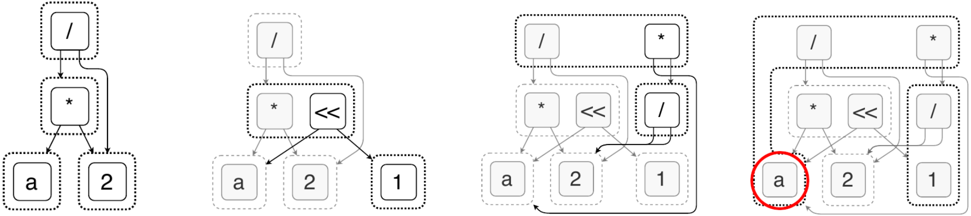
$x * 2 \longrightarrow x \ll 1$



$(x * y)/z \longrightarrow x * (y/z)$

E-Graph Example

$$(a * 2)/2 \longrightarrow^{*} a$$



$(a * 2)/2$

$x * 2 \longrightarrow x \ll 1$

$(x * y)/z \longrightarrow x * (y/z)$

$x/x \longrightarrow 1$

$x * 1 \longrightarrow x$

cost = term size