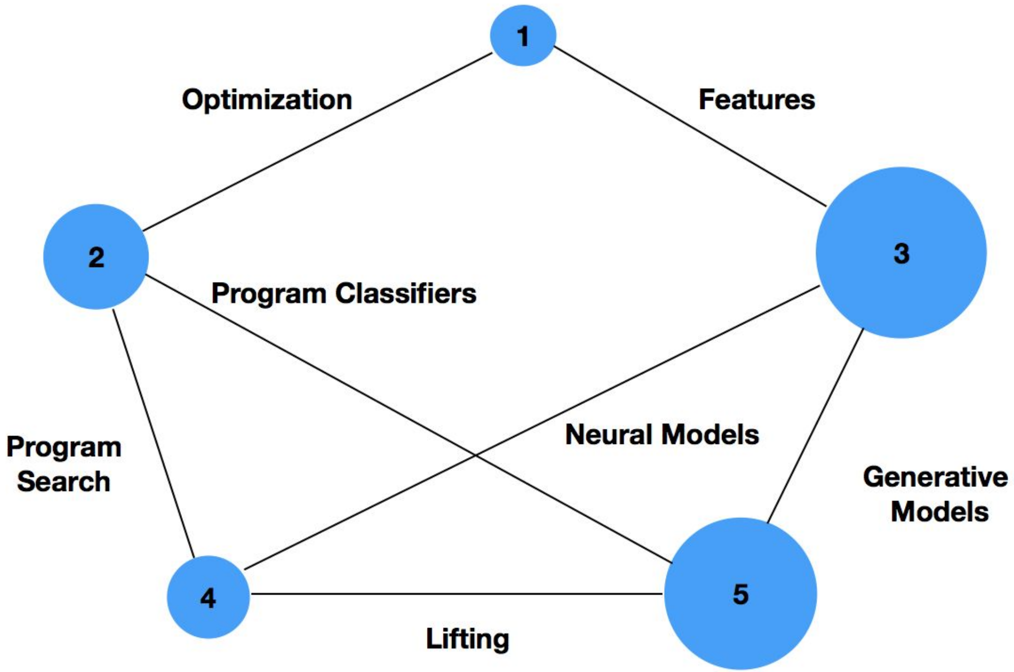
Rethinking Compilation: L2

Overview

- L1: Motivation and brief survey of auto-tuning/machine learning for compilers
- This Lecture: Program rewriting schemes - e-graphs and equality saturation
- L3: Program embeddings and Graph Neural Networks
- L4: Program synthesis and neural synthesis
- L5: Neural Machine Translation,Transformers and Large language models


1
Optimization
Features
2
3
Program Classifiers
Program
Search
Neural Models
Generative
Models
4
5
Lifting

Lecture Structure

1. Why Rewrite?
2. Rewrite Rules in LLVM and GCC
3. Scheduled Rewrites: LIFT and Halide
4. Exploration-Based Rewrites: EGraphs
5. Large-Scale Rewrites: IDL and FACC

Why Rewrite?

- Cannonicalize
  - Easier to write other optimizations
  - Remove challenging constructs
- Better Performance
- Explore programs equivalent by construction

Why Rewrite?

- Simple to Conceptualize
  - Close to code
- Simple to write
  - <Pattern> → <Replacement>
- Simple to compartmentalize

Rewrite Rule Examples

- X * -1 → -X
-

```
For (int I = 0, I < 10; I ++) {
    a[i] ++;
}
```

→

```
For (int I = 0, I < 10; I +=2) {
    a[i] ++;
    a[i + 1] ++;
}
```

- (Unroll)
- <FFT> → <Accelerator>

Rewrite Rules in GCC

- See `match.pd`
  - Lisp-based DSL

```
/* X / -X is -1.  */
(simplify
  (div:C @0 (negate @0))
  (if ((INTEGRAL_TYPE_P (type) || VECTOR_INTEGER_TYPE_P (type))
   && TYPE_OVERFLOW_UNDEFINED (type)
   && !integer_zerop (@0)
   && (!flag_non_call_exceptions || tree_expr_nonzero_p (@0)))
   { build_minus_one_cst (type); }))
```

Conditions

Rewrite Rules in MLIR

- Rewriter uses Tablegen
- Pattern class:

```
class Pattern<dag sourcePattern, list<dag>
resultPatterns,
list<dag> additionalConstraints = [],
dag benefitsAdded = (addBenefit 0)>
```

[1]

MLIR Rule Example [1]

```
def AOp : Op<"a_op"> {
   let arguments = (ins
      AnyType:$a_input,
      AnyAttr:$a_attr
   );
   let results = (outs
      AnyType:$a_output
   );
}
```

```
def COp : Op<"c_op"> {
  let arguments = (ins
    AnyType:$c_input,
    AnyAttr:$c_attr
  );
  let results = (outs
    AnyType:$c_output
  );
}
```

```
def : Pat<(AOp $input, $attr), (COp $input,
$attr)>;
```

Limits of Traditional Approaches

- Scale – How to Handle Massive Patterns?
- Phase-ordering – When to apply what?
- Rule selection – How to deal with lots of rules?

LIFT [3]

- Aim: Rewrite Exploration for code optimization

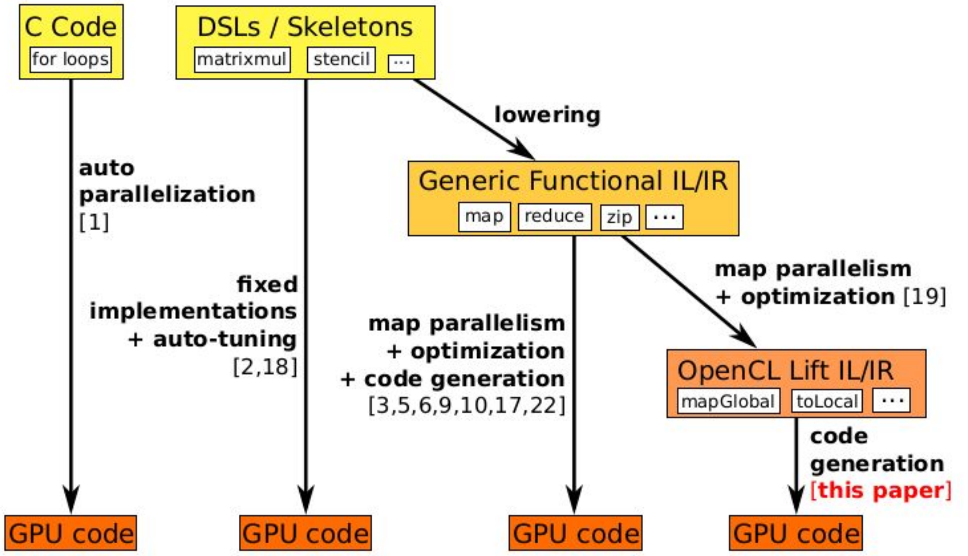
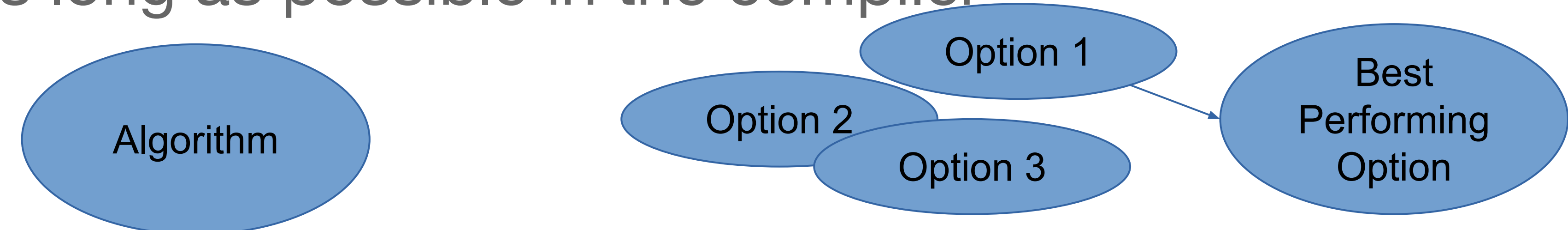


Figure 1. GPU code generation landscape.

What does LIFT Address?

- Which rules to apply?
- When to apply them?

- Key design: “to preserve algorithmic information for as long as possible in the compiler”



LIFT Rewrite Example: Matrix Multiplication Lowering [15]

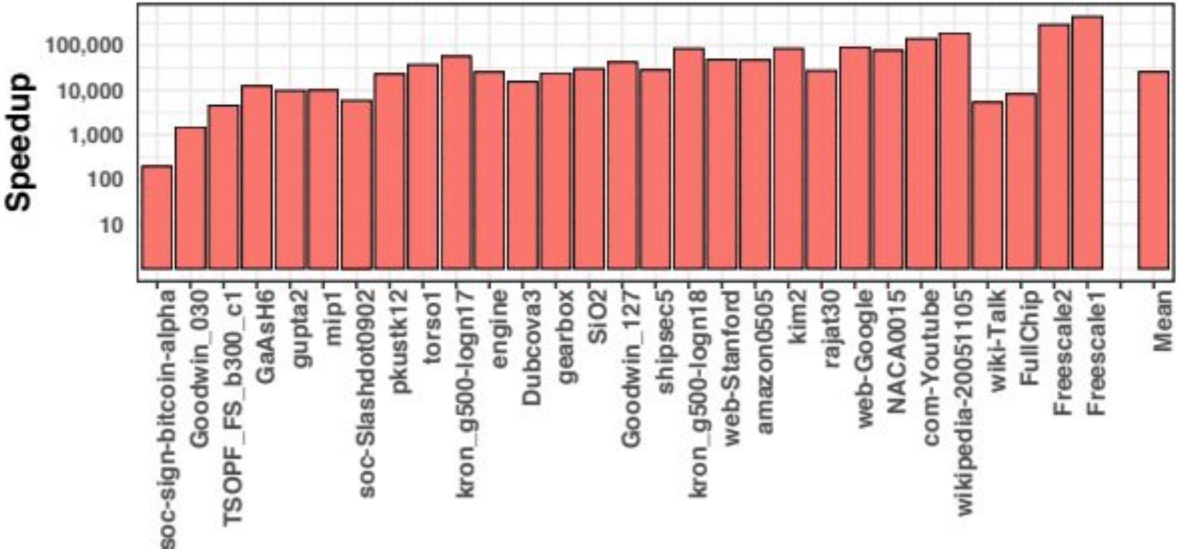
```
nFun(n=>
fun(offsets:[int]n+1 =>
fun(values:[i ↦[float]toNat(offsets@(i+1)-offsets@i)]n =>
 values :>> map(reduce(+, 0.0f)) ) ))
```

Listing 6. Sum of rows for CSR matrix

```
for i in 0...n
 float accum = 0.0f
 for j in 0...toNat(offset@(i+1)-offset@i)
   accum += (matrix@i)@j
 output@i = accum;
```

We replace the ***map*** and ***reduce*** patterns with a **for** loop and explicit array accesses. Additional memory buffers implied in the functional expressions are generated. Multiple

LIFT Rewrite Example: Speedups (Optimized vs Unoptimized) [15]



Limits of LIFT

- Slow (big search space)
- Large-Scale pattern matching still a challenge

Halide [4]

- Separate Algorithm from Schedule
  - Algorithm:

```
bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;
```

- Schedule:

```
bv.tile(x, y, xi, yi, 256, 32)
.vectorize(xi, 8).parallel(y);
bh.compute_at(bv, x).vectorize(x, 8);
```

Optimized
Code

What does Halide Address?

- What Rules to Apply?
- When to Apply them?
- Handles large blocks of code

Limits of Halide

- Hard to write: basically exposing a raw compiler API

Making Halide Easier to Write: Autoscheduling [5]

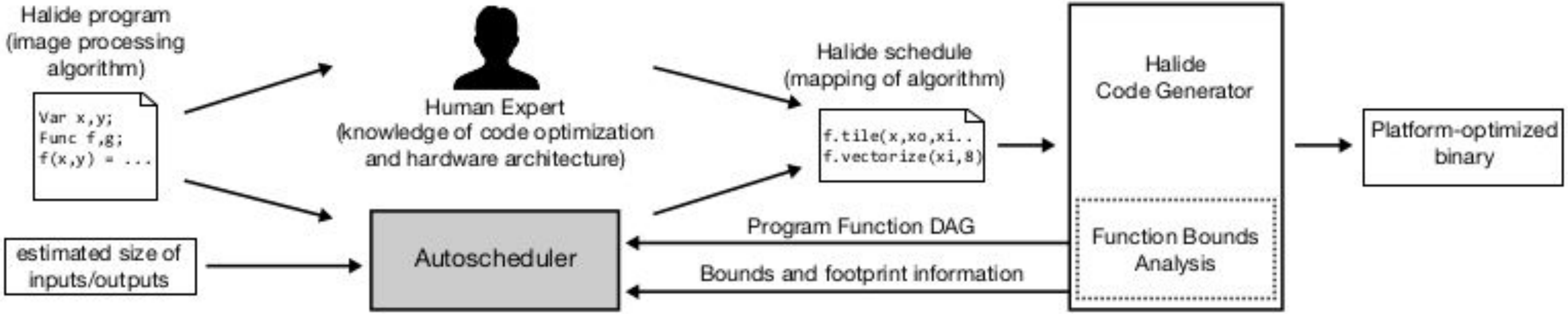


Figure 1: *Our system automatically generates schedules for Halide programs, a task currently performed by expert Halide programmers.*

General Approaches to Autoscheduling: Sketch-based [13]

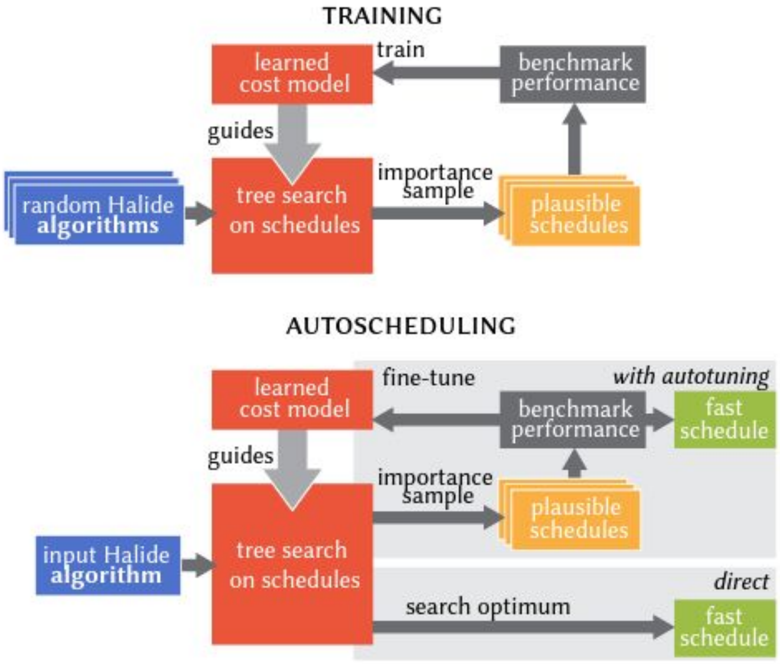
	AutoTVM Workflow	Auto-scheduler Workflow

Step 1:
Write a compute definition
 	
(relatively easy part)
	`# Matrix multiply`
	
	`C = te.compute((M, N), lambda x, y:`
	`te.sum(A[x, k] * B[k, y], axis=k))`
		`# The same`

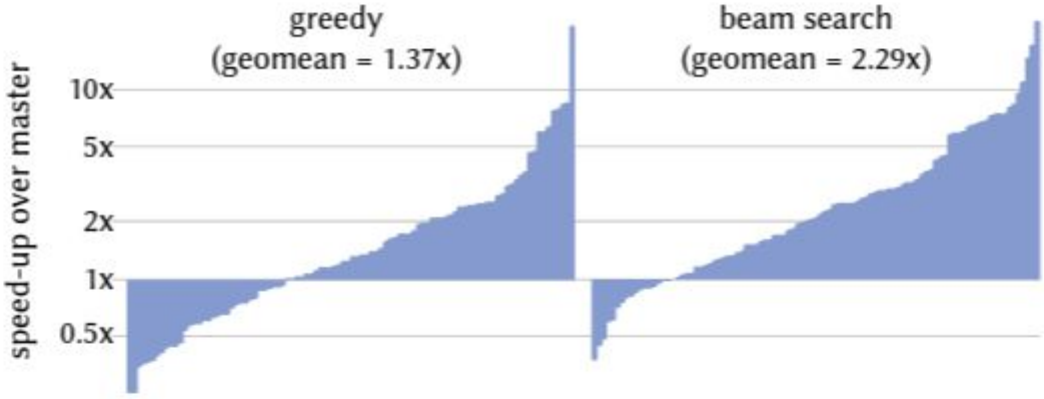
Step 2:
Write a schedule template
 	
(difficult part)
	`# 20-100 lines of tricky DSL code`
	
	`# Define search space`
	`cfg.define_split("tile_x", batch, num_outputs=4)`
	`cfg.define_split("tile_y", out_dim, num_outputs=4)`
	`…`
	
	`# Apply config into the template`
	`bx, txz, tx, xi = cfg["tile_x"].apply(s, C, C.op.axis[0])`
	`by, tyz, ty, yi = cfg["tile_y"].apply(s, C, C.op.axis[1])`
	`s[C].reorder(by, bx, tyz, txz, ty, tx, yi, xi)`
	`s[CC].compute_at(s[C], tx)`
	`…`
		`# Not required`

Step 3: Run auto-tuning (automatic search)	`tuner.tune(…)`	`task.tune(…)`

General Approaches to Autoscheduling: Exploration-Based [14]

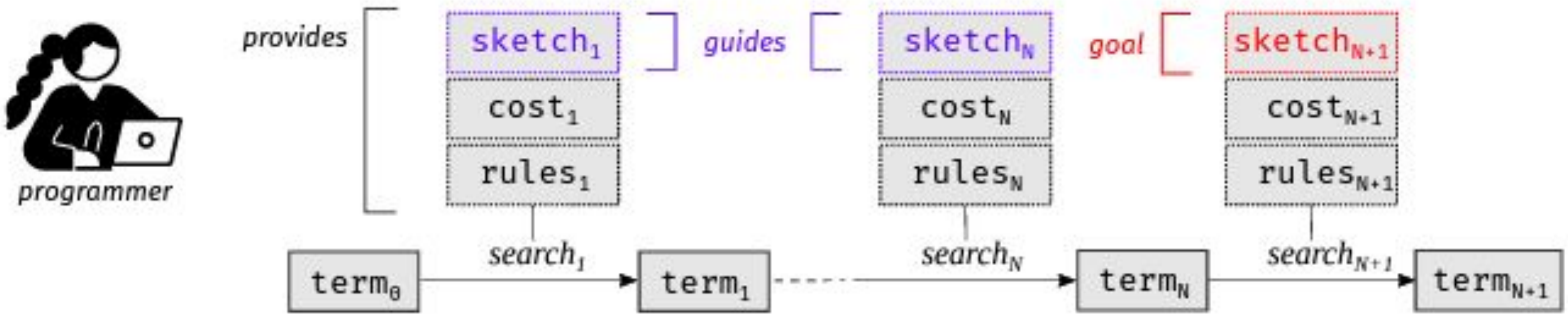


General Approaches to Autoscheduling: Exploration-Based [14]



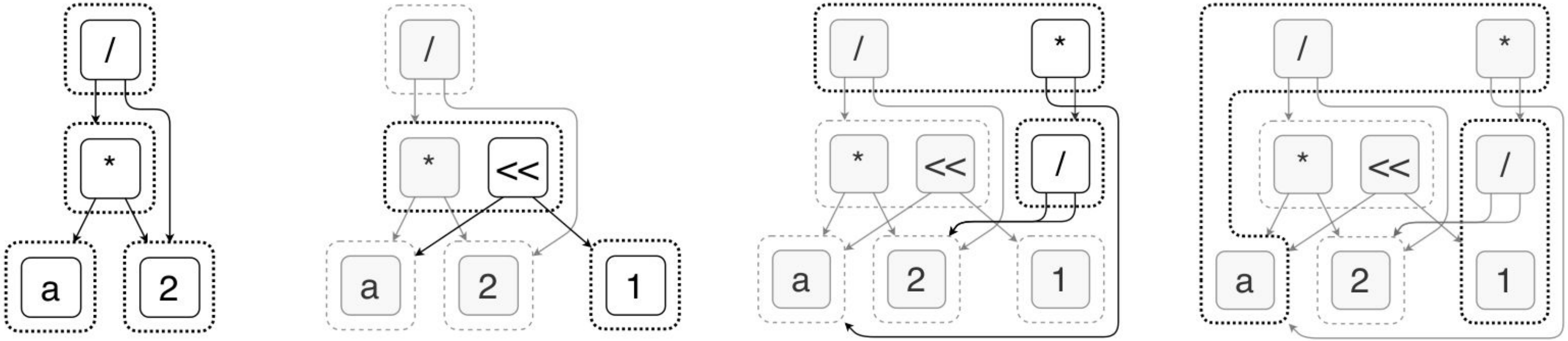
Performance relative to exiting autoscheduler

Making Halide Easier to Write, Schedule Synthesis [6]



Egraphs [7]

- Intuition: Apply Every Rule at Once



(a) Initial e-graph contains $(a \times 2)/2$.

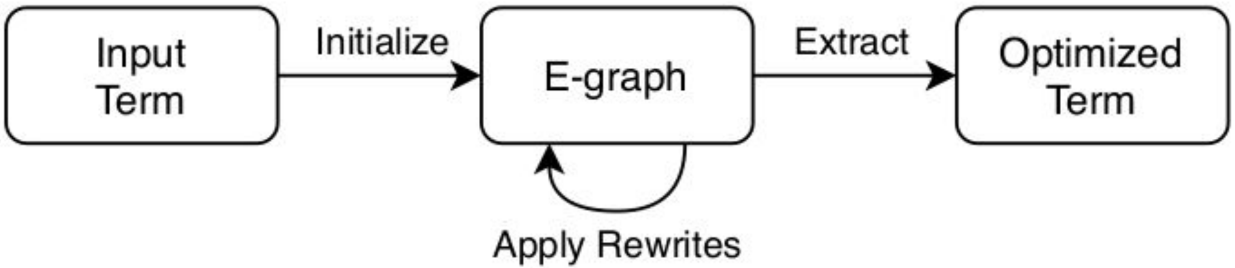
(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.

(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

(d) After applying rewrites $x/x \rightarrow 1$ and $1 \times x \rightarrow x$.

Egraphs: Strategy

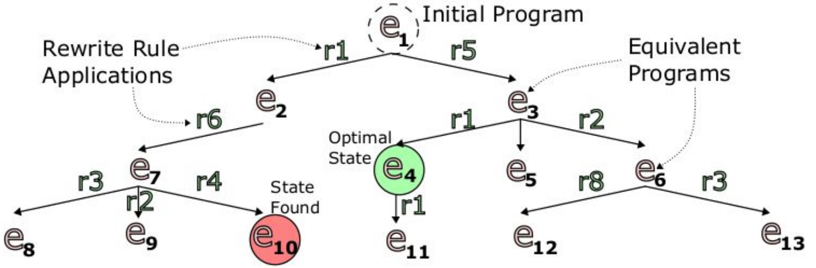
1. Generate Equivalence Classes using Rewrite Rules (equality saturation [Equality Saturation]
2. Solve for best graph (e.g. with ILP) [EGraphs]



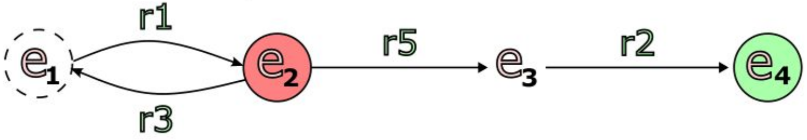
What Do EGraphs Solve?

- What Rules to Apply?
- When to Apply Them?

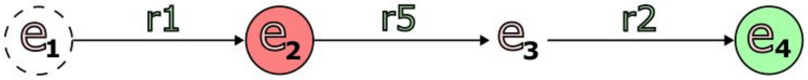
Egraphs: Intuition [FlexC]



(a) In the *exploration problem*, the expression in green is the optimal choice for this CGRA, but may never be reached in a greedy application of rewrite rules, which will reach the red state instead.



(b) In the *cycle problem*, A greedy rewriter may get stuck in a cycle due to cyclical groups of rules, preventing it from finding the optimal state.

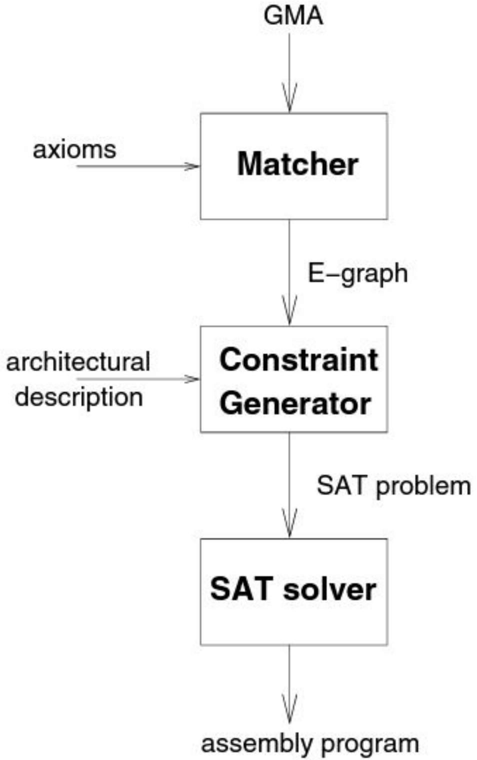


(c) In the *cost-trap problem*, A greedy rewriter can get stuck in state e_2 as e_3 is a less valuable state.

Challenges of EGraphs

- Computation Time
- Handle Large Rules?

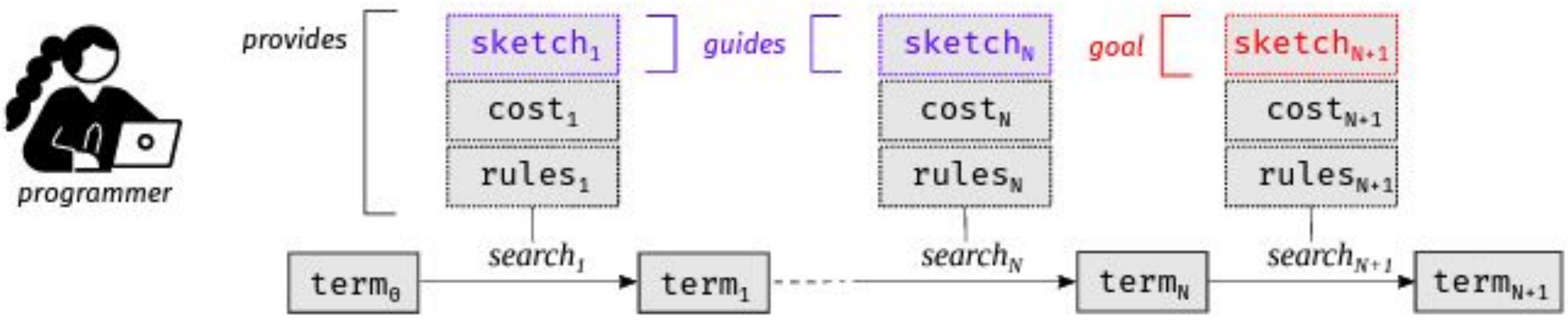
Case Study: Denali [2] (2002)



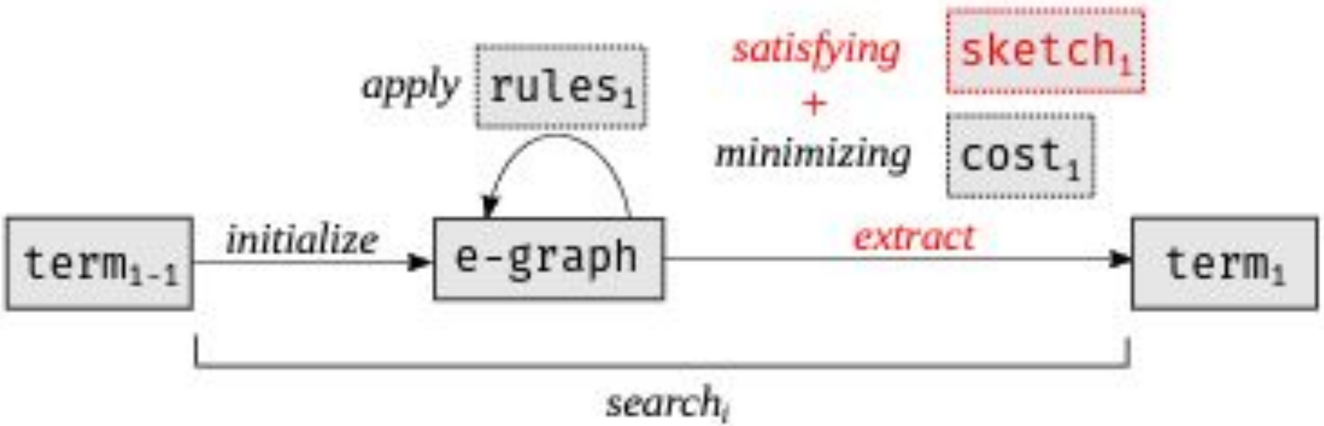
Case Study: Why didn't Denali take off? [2]

- Lack of Computer Power in 2002 (SAT Solver Costly)
- No plug-and-play library
- Straight-line code only

Egraphs: Memory Usage [6]

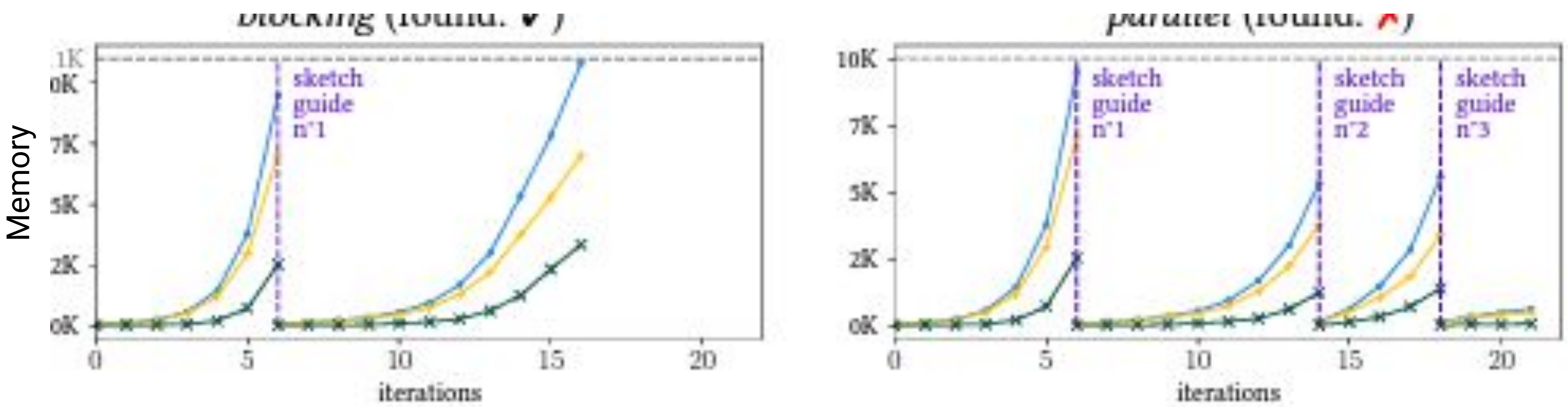


Egraphs: Memory Usage [6]



- Key concept: Reduce Search Space

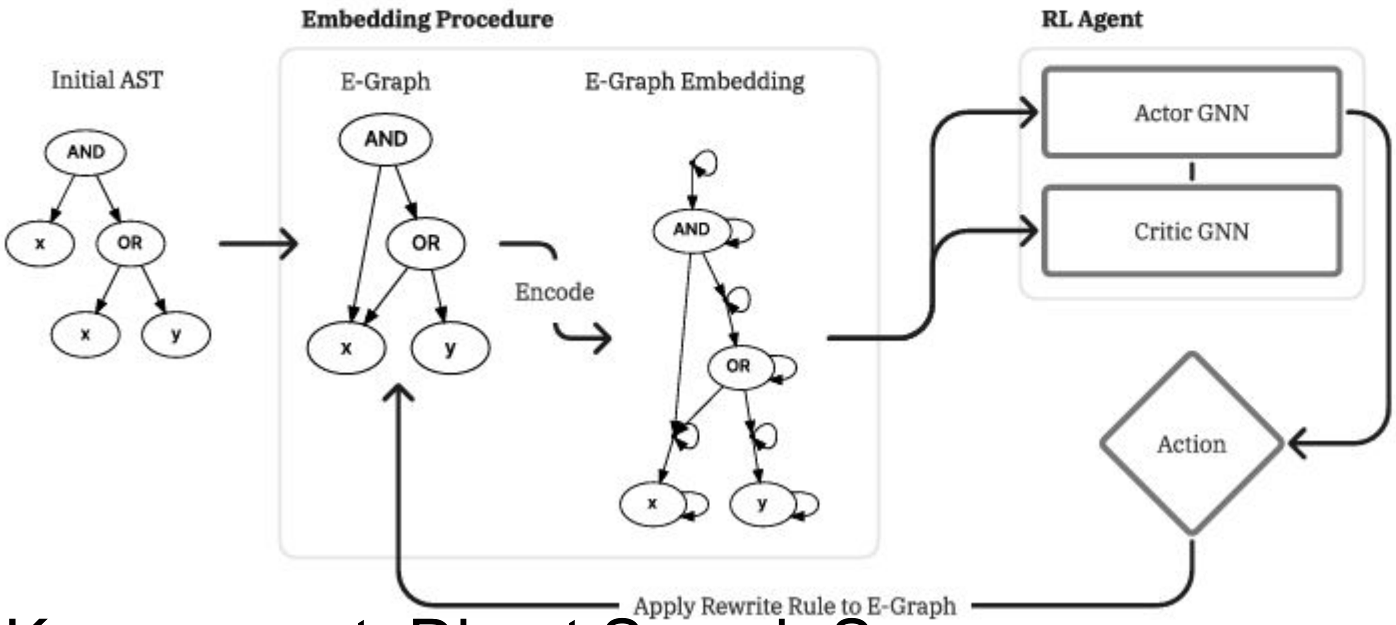
Egraphs: Memory Usage [6]



(c) sketch-guided equality saturation *blocking* (found: ✓)

(d) sketch-guided equality saturation *parallel* (found: ✓)

Egraphs: Computation Time [8]

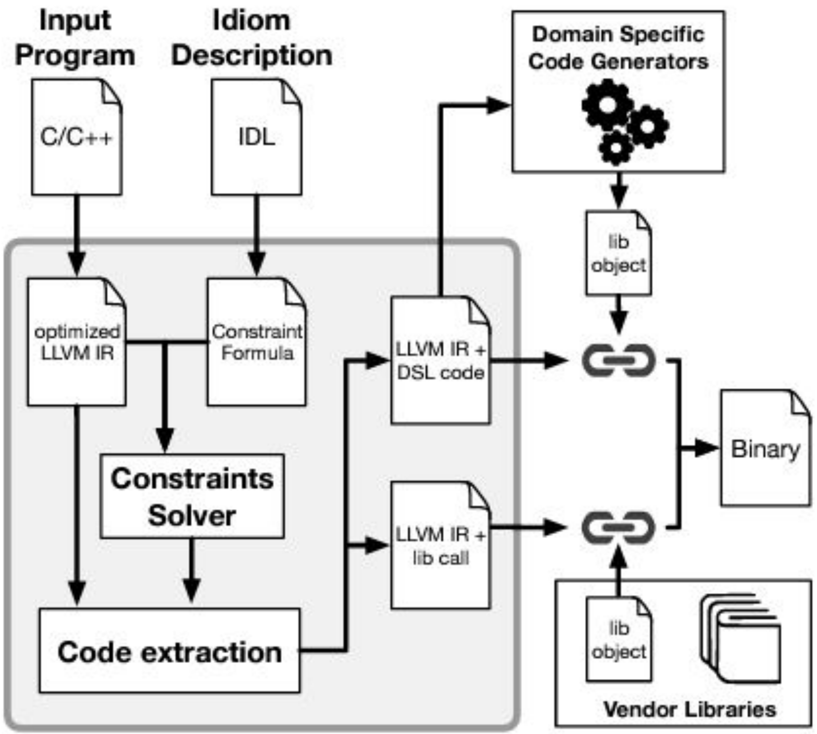


- Key concept: Direct Search Space

Large-Scale Rewrites [10]

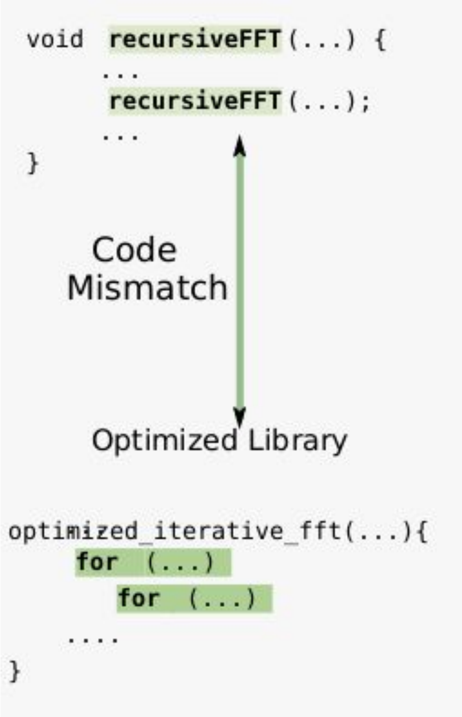
- Why?
  - Optimize large chunks of code (e.g. matmul)
- Challenge:
  - Cannonicalization is hard at scale

Large-Scale Rewrites [10]



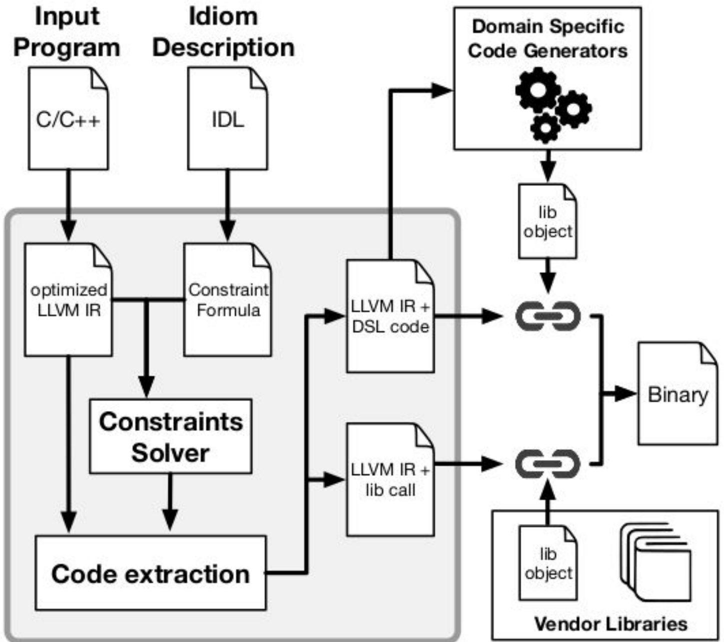
Cannonicalization at Scale [11]

Two methods of implementing FFTs



IDL: Large Patterns [10]

- Use constraint solving and a DSL for pattern matching:



IDL Example [10]

- Hard to read
- Hard to write

```
Constraint FactorizationOpportunity
( {sum} is add instruction and
  {left_addend} is first argument of {sum} and
  {left_addend} is mul instruction and
  {right_addend} is second augment of {sum} and
  {right_addend} is mul instruction and
  ( {factor} is first argument of {left_addend} or
    {factor} is second argument of {left_addend}) and
  ( {factor} is first argument of {right_addend} or
    {factor} is second argument of {right_addend}))
End
```

Figure 2. IDL formulation of (x*y)+(x*z) pattern

Challenges of IDL: Pattern Size [10]

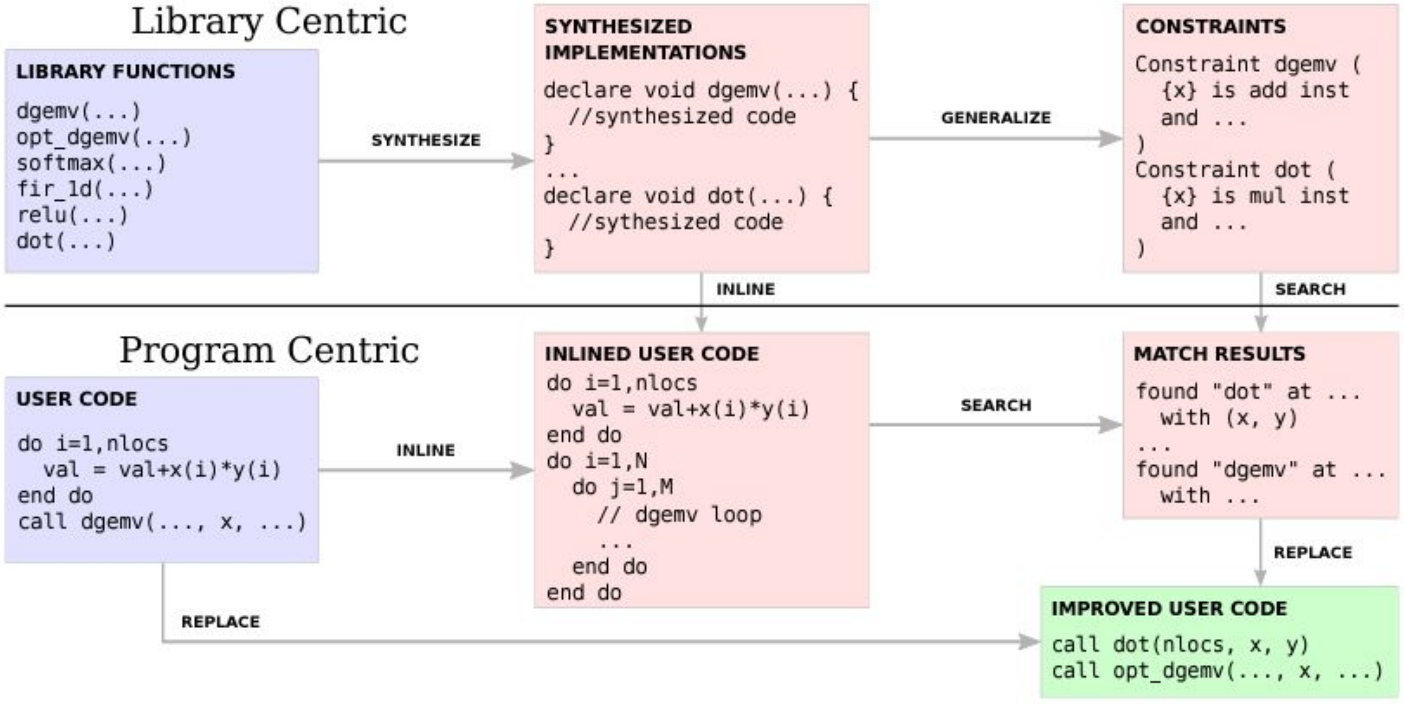
- Very very hard to read/write/compose

```
Constraint SESE
( {precursor} is branch instruction and
{precursor} has control flow to {begin} and
{end} is branch instruction and
{end} has control flow to {successor} and
{begin} control flow dominates {end} and
{end} control flow post dominates {begin} and
{precursor} strictly control flow dominates
{begin} and
{successor} strictly control flow post dominates
{end} and
all control flow from {begin} to {precursor}
passes through {end} and
all control flow from {successor} to {end}
passes through {begin})
End
```

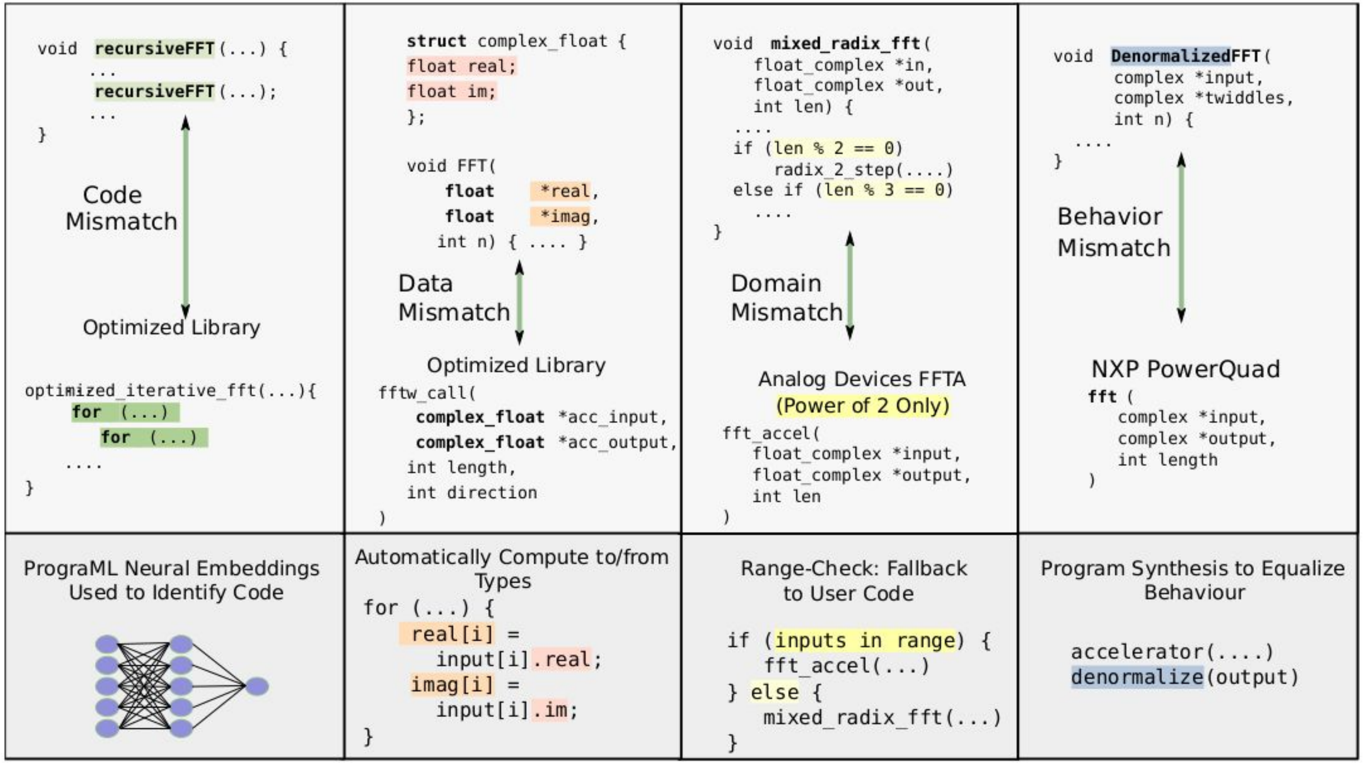
Challenges of IDL: Pattern Size with Synthesis as a Solution

- Automate pattern Writing

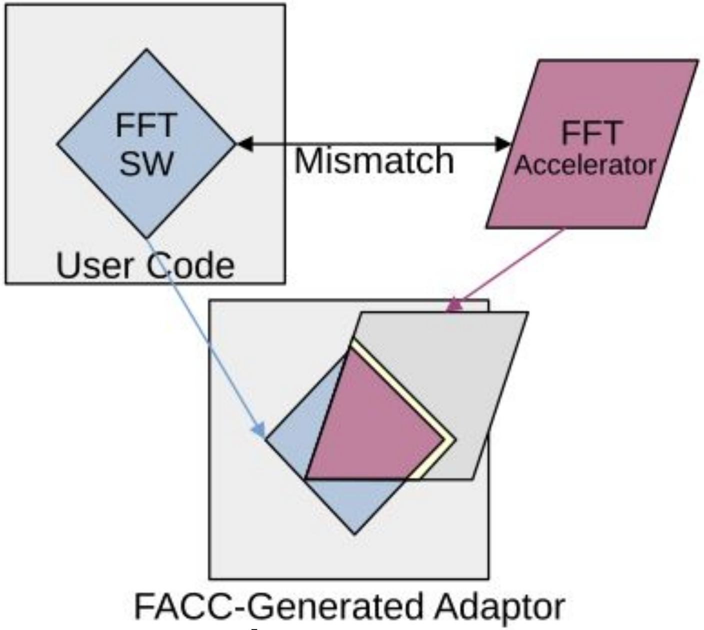
- Reduce Learning curve



Challenges of IDL: Mismatch [11]



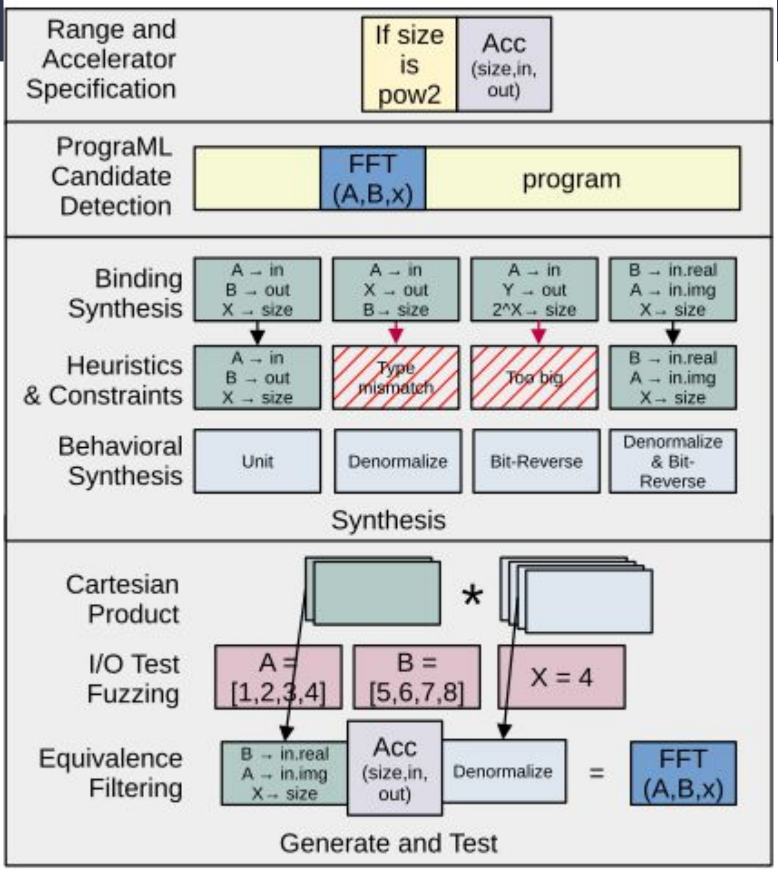
Synthesis as a Solution [11]



Code and pattern may not match: generate code so that they do.

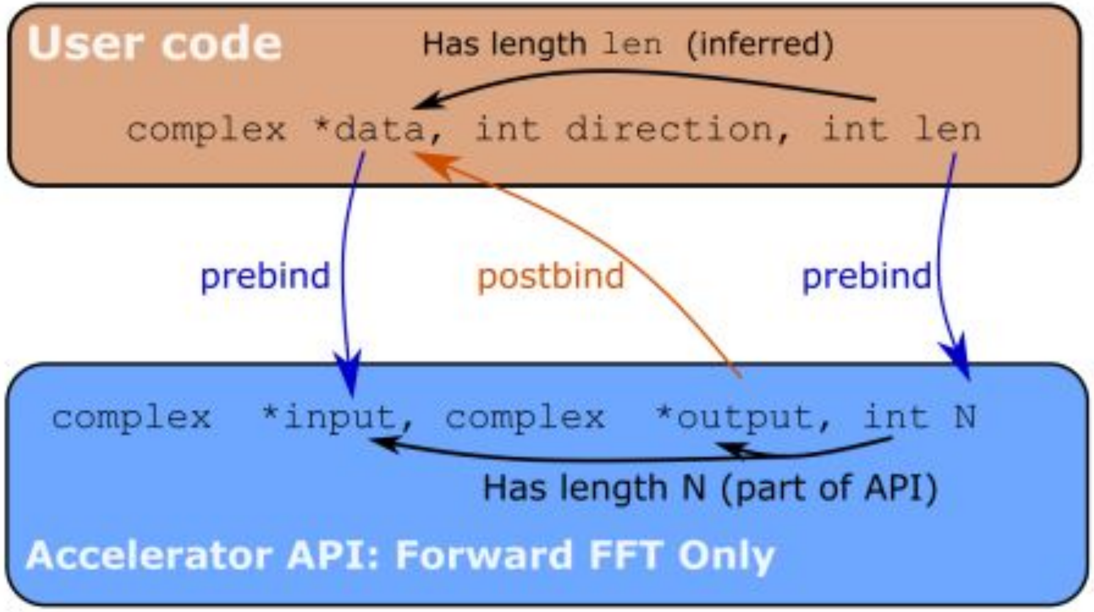
FACC Strategy (for APIs)

1. Use ML to identify code to match
2. Use program synthesis to bind it to the pattern
3. Use program synthesis to make behaviours line up
4. Test using IO to determine correct synthesized program



FACC Output: Binding Code to APIs [11]

User code and API are written differently

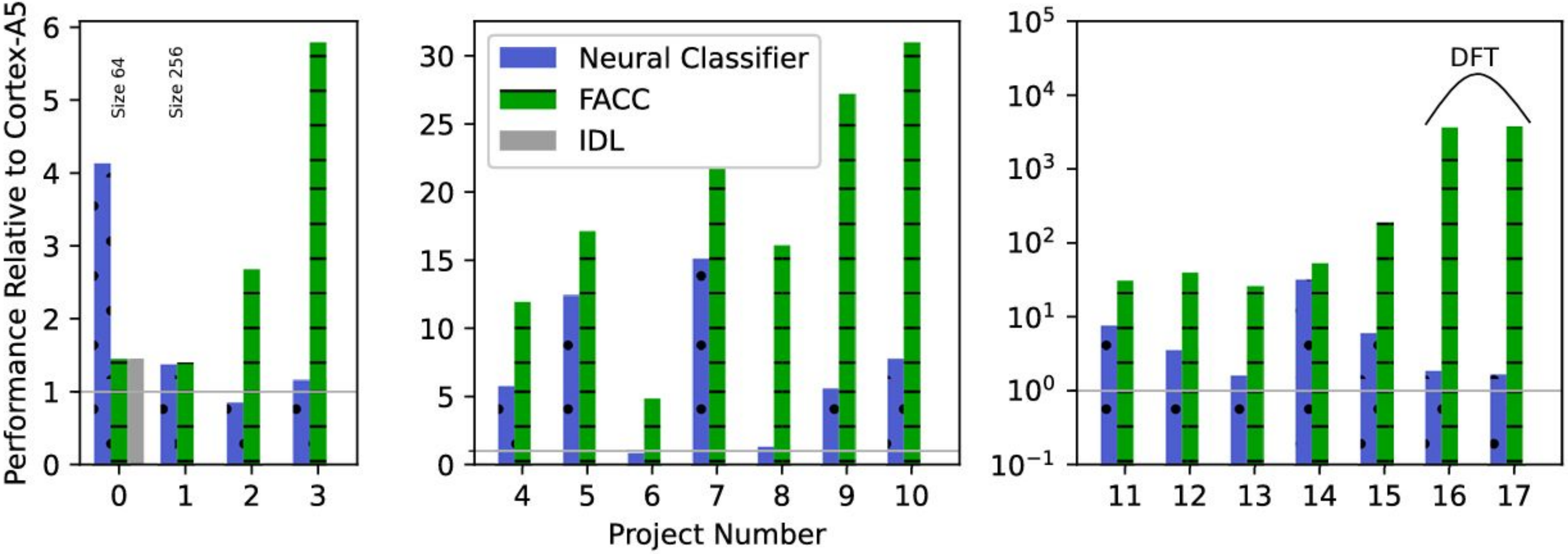


FACC Output: Binding Code to APIs [11]

- Orange: Range Check
- Grey: Pre-Binding
- Pink: Post-Binding
- Green: Behavioural Synthesis

```
complex *FFT_accel(complex *x, int N) {
  // Check for valid inputs to accelerator
  if (is_power_of_two(N) && N <= 65536) {
    // Bind user inputs to accelerator
    int len = N;
    #pragma align 64
    complex_float output[len];
    complex_float input[len];
    #pragma end
    for (int i = 0; i < len; i++) {
      input[i].re = x[i].real;
      input[i].im = x[i].imag;
    }
    // Call accelerator
    accel_cfft(input, output, len);
    // Bind accelerator outputs
    for (int j = 0; j < N; j++) {
      x[j].imag = output[j].im;
      x[j].real = output[j].re;
    }
    // De-normalize outputs
    for (int k = 0; k < N; i ++)  {
      x[k].imag *= N;
      x[k].real *= N;
    }
  } else { // Not valid accelerator input
    // Fallback to user code.
    UserFFT(x, N);
  }
}
```

FACC Output: Binding Code to APIs [11]



What does FACC Address?

- Handles Large Blocks of Code

Limits of FACC

- Does not tell you when to apply rewrites
- No correctness guarantees

Guaranteed Correctness with SMT

Mosiac: Integrating Large-Scale Rewrites into a Compiler [12]

- Large-Scale Rewrites for a Tensor compiler
- Integrate large-scale pattern matching and traditional optimization
- Prove transformations correct

Mosiac: Integrating Large-Scale Rewrites into a Compiler [12]

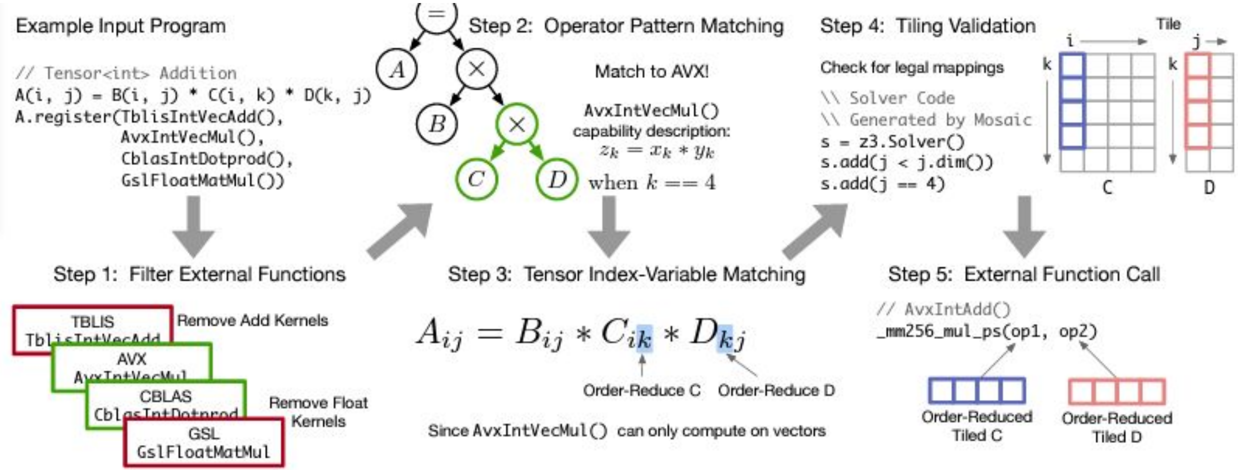
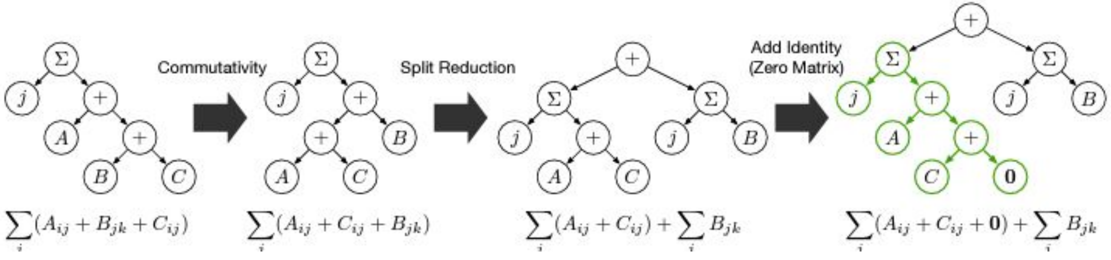


Fig. 9. Steps in the automatic searching process.



Limits of Mosiac

- Large effort to integrate new APIs
- No overhead removal

Future Directions for Rewrite Rules

- Full system handling:
  - Automatic rule selection
  - Large-scale rules
  - No programmer interaction

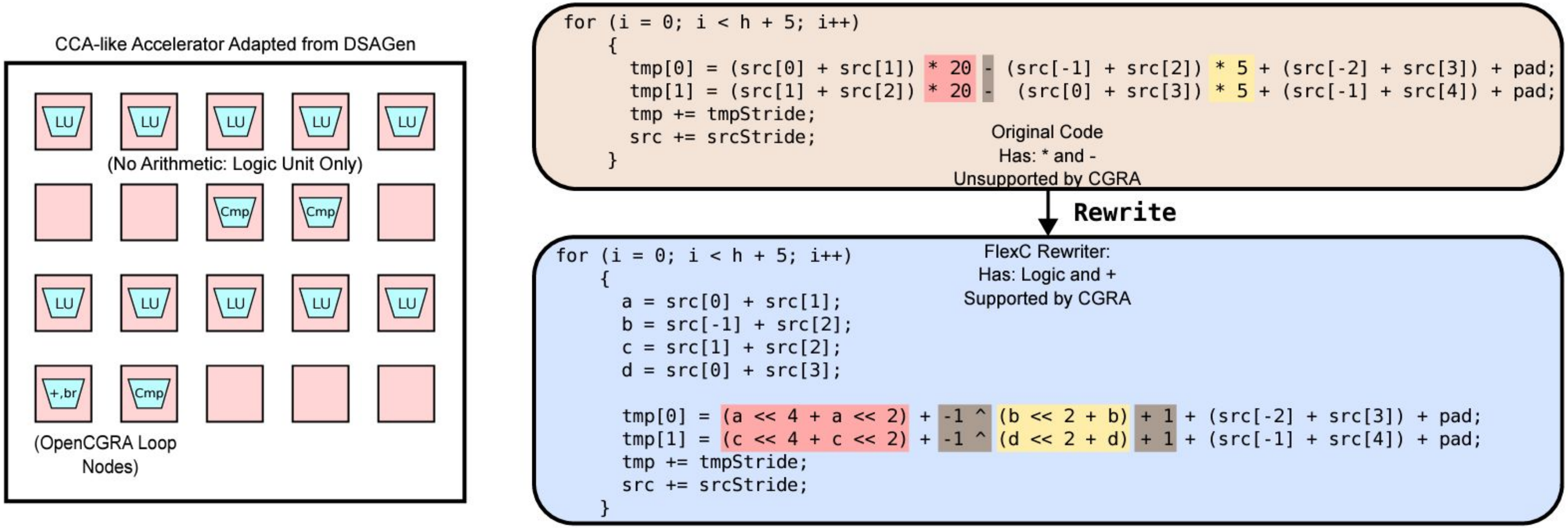
FlexC: Rewriting with EGraphs [9]

Heterogeneous CGRAs may not support all operations.

Key concept:

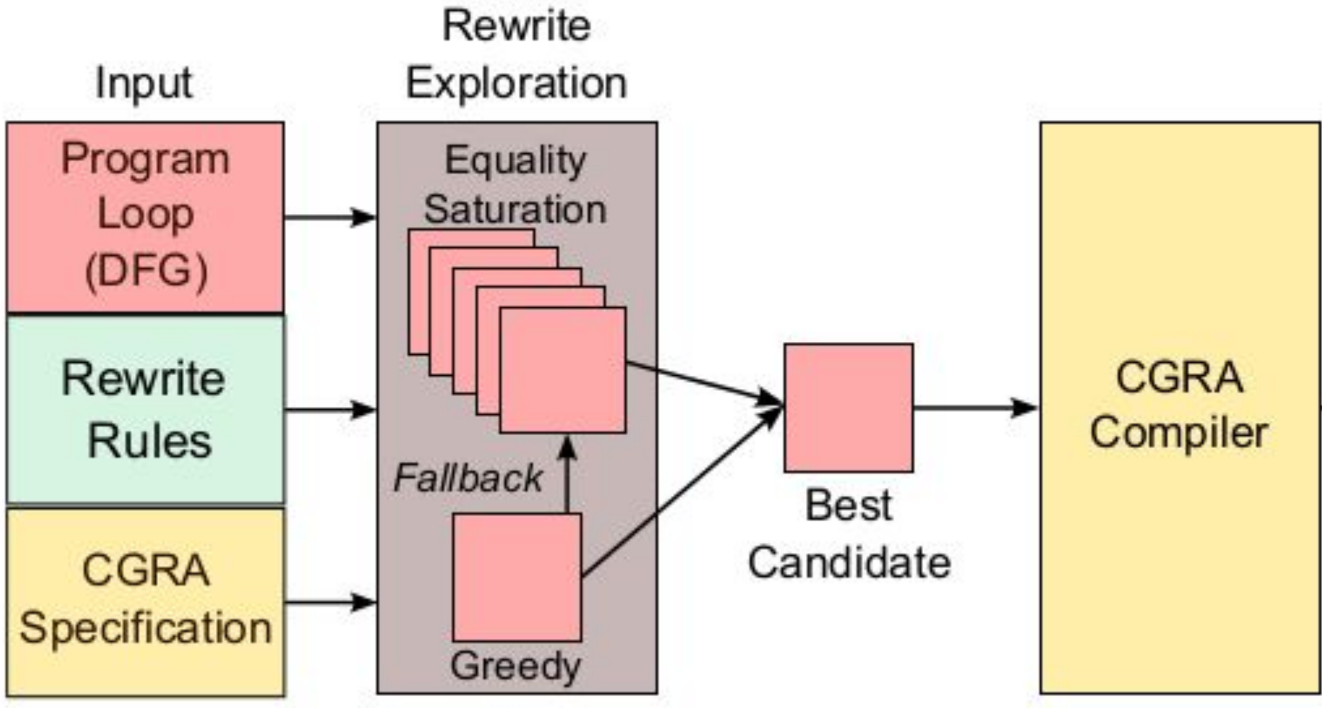
- Rewrite code to use only supported ops

FlexC: Rewriting with EGraphs [9]

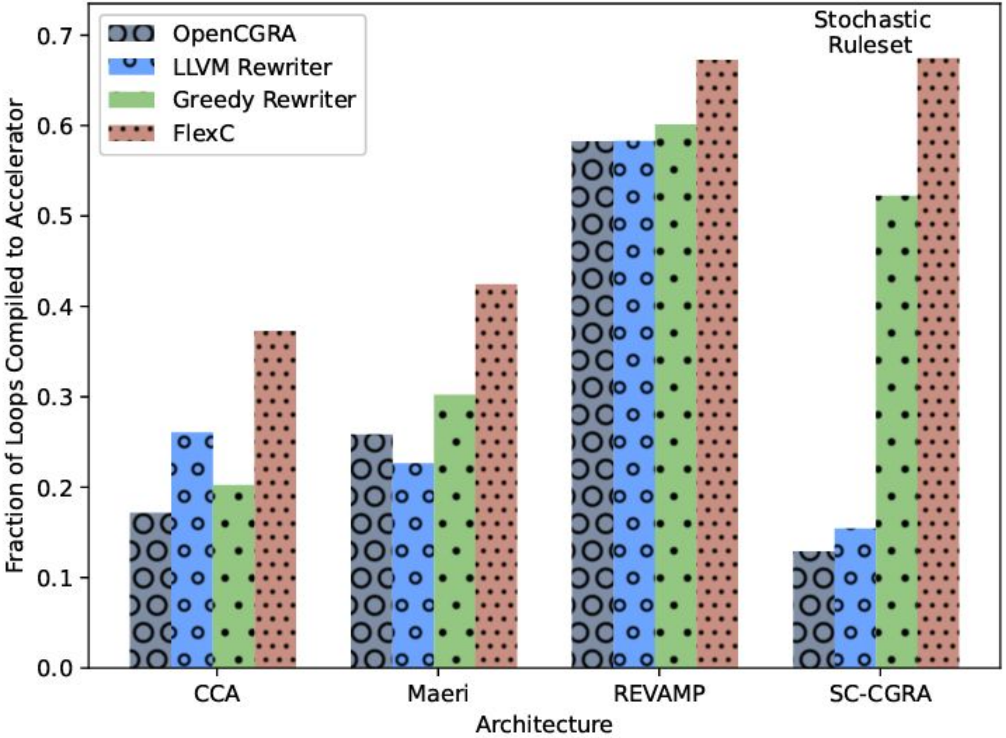


FlexC: Rewriting with EGraphs [9]

1. Rewrite using traditional rules (fast)
2. If no match found use EGraphs (complete)



FlexC: Rewriting with EGraphs [9]



Summary

- Rewrites are a powerful tool:
- Rewrites present key challenges:
  - What rule to apply?
  - When to apply rules?
  - How to scale?
- Research Projects to address key challenges:
  - LIFT/Halide
  - FACC/IDL
  - EGraphs

Overview

- L1: Motivation and brief survey of auto-tuning/machine learning for compilers
- L2: Program rewriting schemes - e-graphs and equality saturation
- Next lecture L3: Program embeddings and Graph Neural Networks
- L4: Program synthesis and neural synthesis
- L5: Neural Machine Translation,Transformers and Large language models

References

- [1] mlir.llvm.org/docs/DeclarativeRewrites
- [2] Joshi, Nelson, Randall, Denali: A Goal-directed Superoptimizer, PLDI 2002
- [3] Steuwer, Remmelg, Dubach, Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation, CGO 2017
- [4] Ragan-Kelley, Adams, Charlet, Barnes, Paris, Levoy, Amarasinghe, Durand, Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing, Communications of the ACM 2018

References

- [5] Mullapudi, Adams, Sharlet, Ragan-Kelley, Fatahalian, Automatically Scheduling Halide Image Processing Pipelines, SIGGRAPH 2016
- [6] Koehler, A Domain-Extensible Compiler with Controllable Automation of Optimizations, University of Glasgow, PhD Thesis, 2022
- [7] Willsey, Nandi, Wang, Flatt, Tatlock, Pancheckha, egg: Fast and Extensible Equality Saturation, POPL 2021
- [8] Deep Reinforcement Learning for Eqaulity Saturation, Singh, University of Cambridge Masters Thesis (2022)

References

- [9] Woodruff, Koehler, Brauckmann, Cummins, Ainsworth, Steuwer, O'Boyle, Rewriting History: Repurposing domain-specific hardware accelerator with rewrite exploration, Under submission 2023
- [10] Ginsbach, Remmelg, Steuwer, Bodin, Dubach, O'Boyle, Automatic Matcing of Legacy Code to Heterogeneous APIs: An Idiomatic Approach, ASPLOS 2018
- [11] Woodruff, Armengol-Estape, Ainsworth, O'Boyle, Bind the Gap: Compiling Real Software to Hardware FFT Accelerators, PLDI 2022
- [12] Bansal, Hsu, Olukoton, Kjolstad, Mosaic: An Interoperable Compiler for Tensor Algebra, PLDI 2023

References

- [13]https://tvm.apache.org/2021/03/03/intro-auto-scheduler
- [14] Adams, Ma, Anderson, Baghdadi, Li, Gharbi, Steiner, Johnson, Fatahalian, Durand, Regan-Kelly, Learning to Optimize Halide with Tree Search and Random Programs, ACM Trans Graph 2019.
- [15] Pizzuti, Steuwer, Dubach, Generating Fast Sparse Matrix Vector Multiplication from a High Level Generic Functional IR, CC 2020
- [16] https://gcc.gnu.org/onlinedocs/gccint/The-Language.html
- [17] Ragan-Kelley, Barnes, Adams, Paris, Durand, Amarasinghe, Halide: A Language and Compiler for Optimizing Parallelism, Locality and Recomputation in Image Processing Pipelines, PLDI 2013
- [18] Bacon, Graham, Sharp, Compiler Transformations for High-Performance Computing, ACM Computing Surveys, Vol. 26, No. 4, 1994