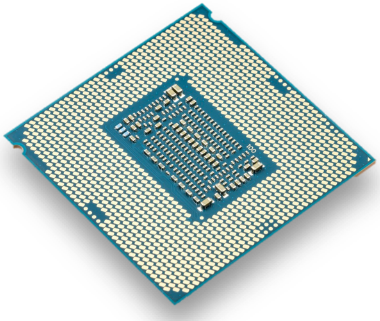
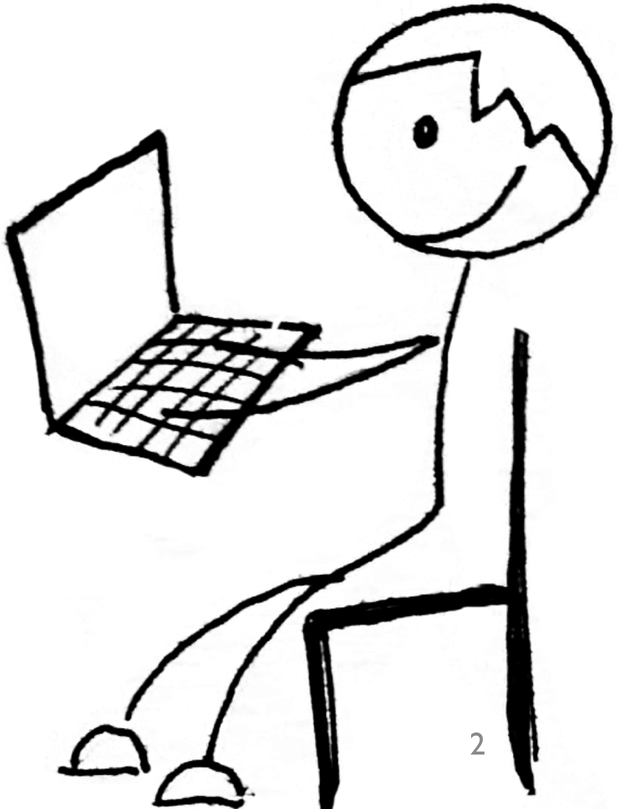
Toward Self-Evolving Compilers

Phitchaya Mangpo Phothilimthana

Google Brain

*based on the work of **many** people*

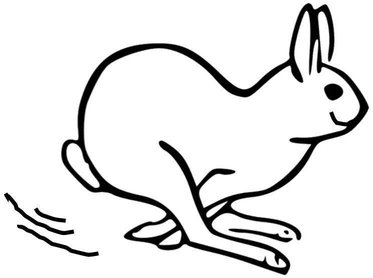


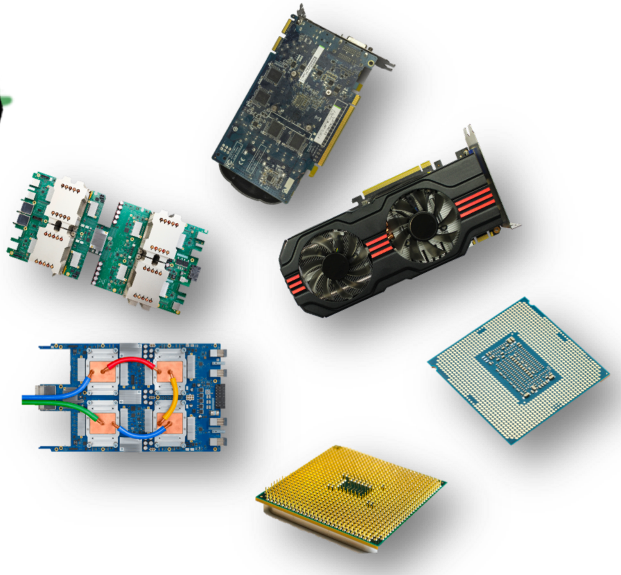


unusual ISA

heterogeneity

no register





restricted computations

limited resources

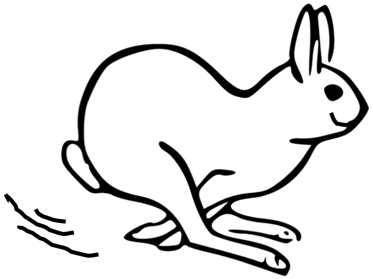
new memory hierarchy

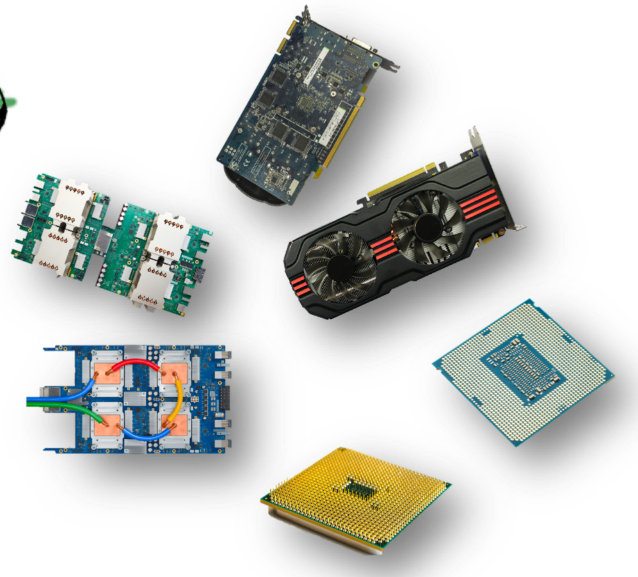
distributed memory

unusual ISA

heterogeneity

no register



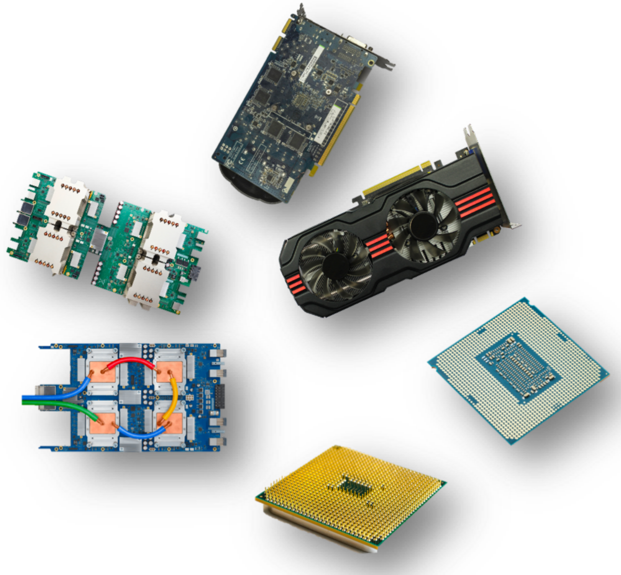


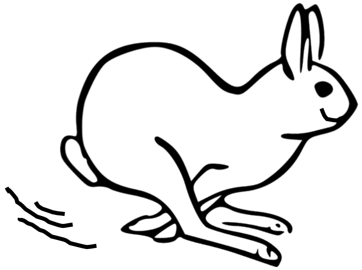
restricted computations

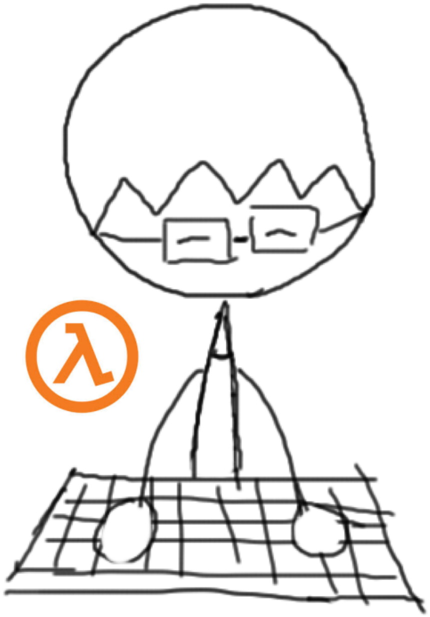
limited resources

new memory hierarchy

distributed memory







unusual ISA

heterogeneity

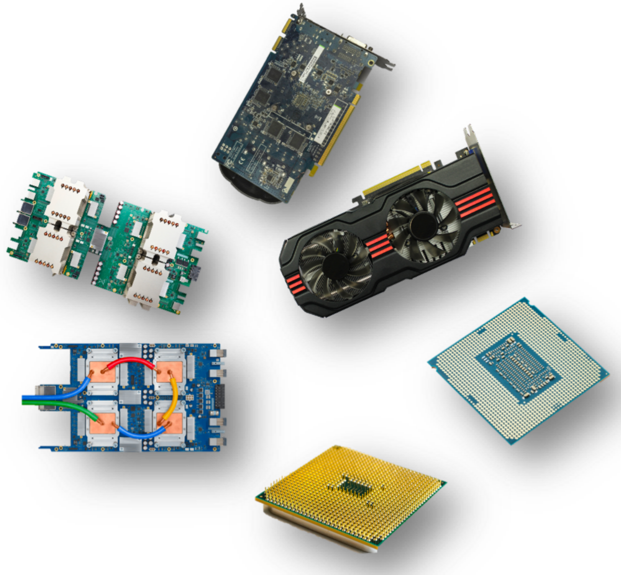
no register

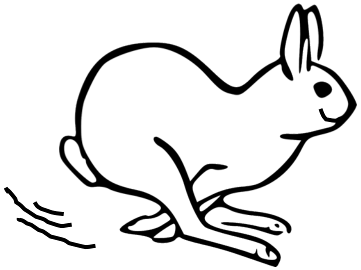
restricted computations

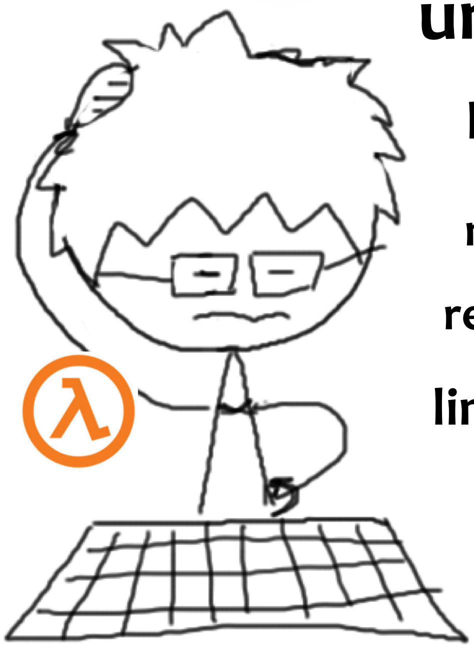
limited resources

new memory hierarchy

distributed memory







unusual ISA

heterogeneity

no register

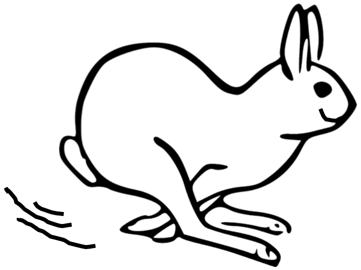
restricted computations

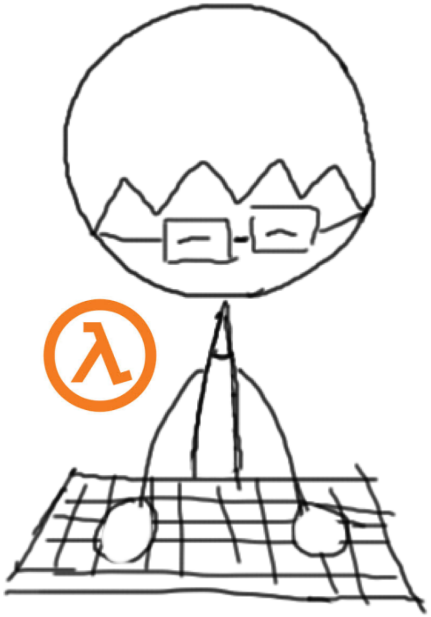
limited resources

new memory hierarchy

distributed memory

Synthesis-aided compilation





unusual ISA

heterogeneity

no register

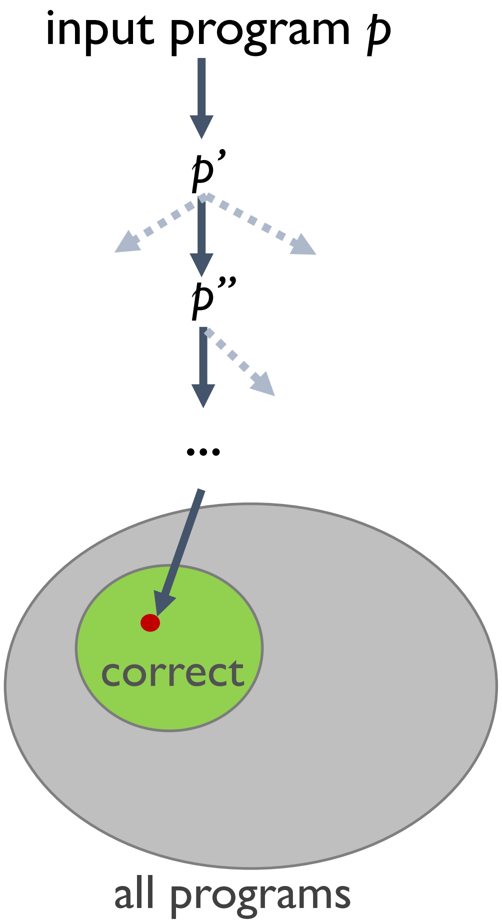
restricted computations

imited resources

new memory hierarchy

distributed memory

Classical



input program *p*

p’

p”

…

correct

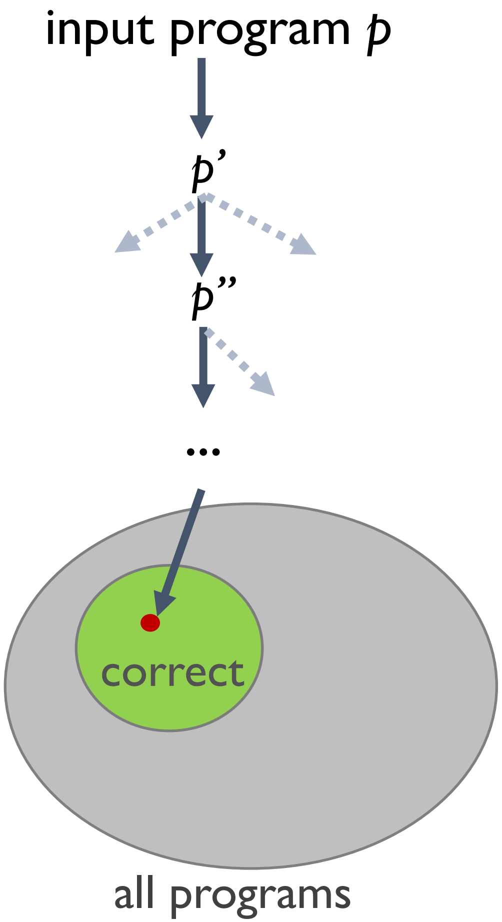
all programs

Pros: fast to compile

Cons: miss efficient programs

Synthesis-aided

Classical



input program p

p'

p''

…

correct

all programs

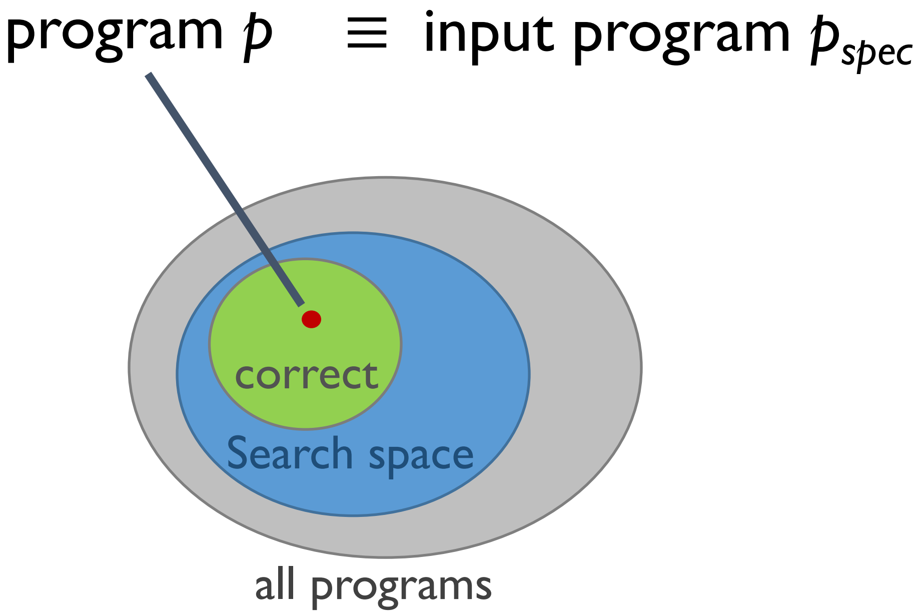
Pros: fast to compile

Cons: miss efficient programs

Synthesis-aided

Define search space $\boldsymbol{P}$

Find $p \in \boldsymbol{P}$



program p $\equiv$ input program p_{spec}

correct

Search space

all programs

Pros: find provably optimal program

Cons: slow to compile

Swizzle Inventor

semi-automatic
bypass rewrite rules & heuristics

GreenThumb
Superoptimization

fully-automatic
bypass rewrite rules & heuristics

AutoX

fully-automatic
bypass heuristics

Swizzle
Inventor

semi-automatic
bypass rewrite rules & heuristics

GreenThumb
Superoptimization

fully-automatic
bypass rewrite rules & heuristics

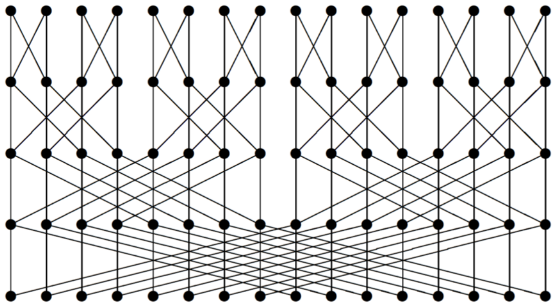
AutoX

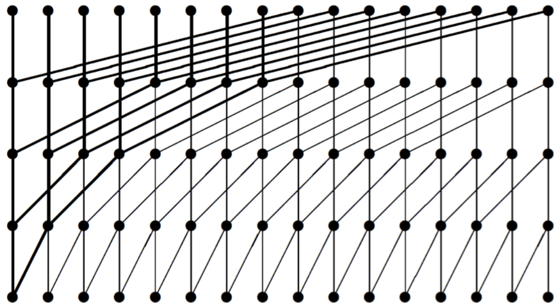
fully-automatic
bypass heuristics

Swizzle

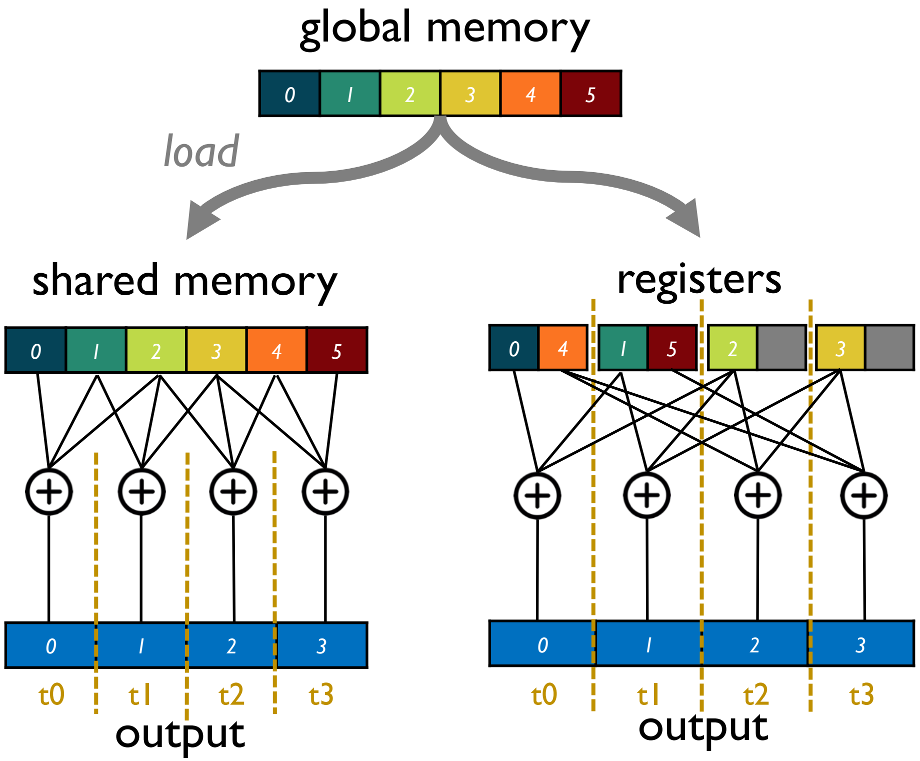
non-trivial movement of data or
non-trivial mapping of computations
to **hardware resources** and **loop iterations**

for dramatic performance improvement



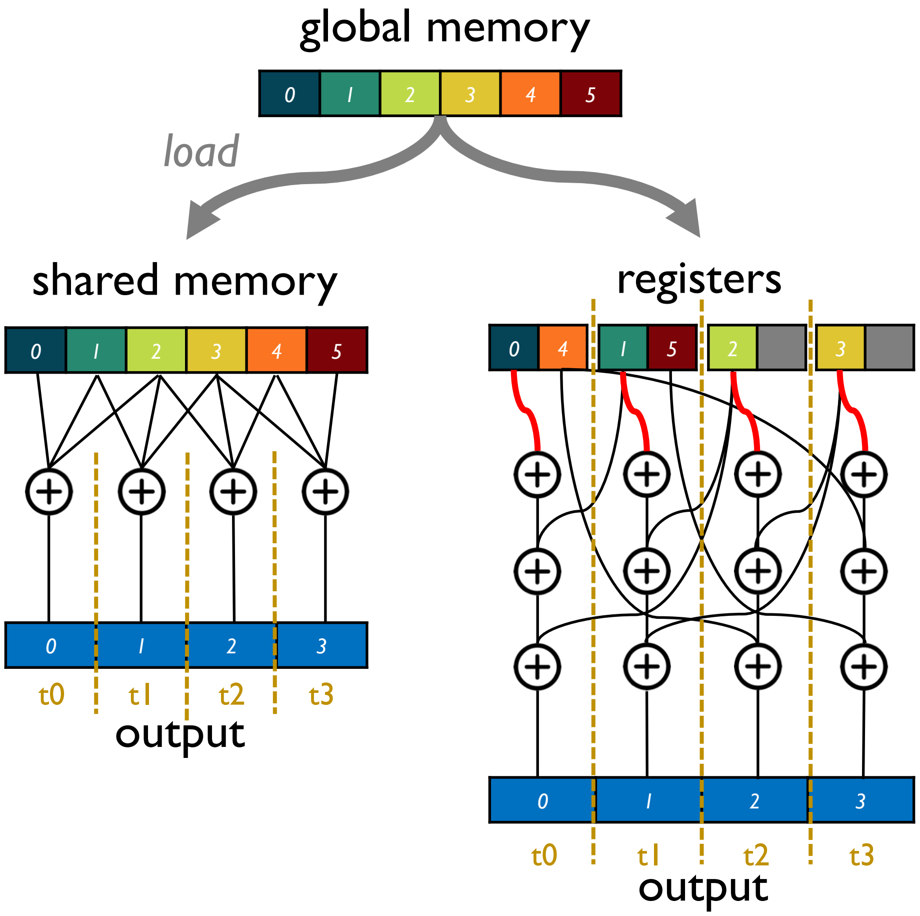


Register Cache: Stencil



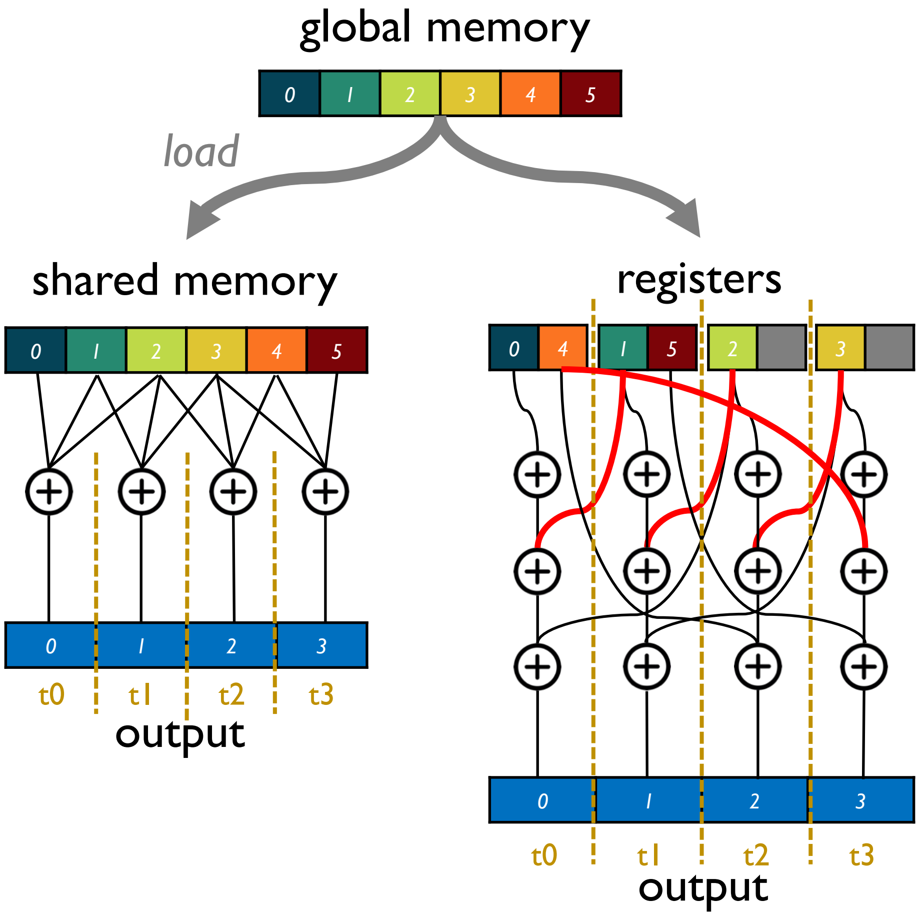
[Ben-Sasson et al. ICS' 16]

Register Cache: Stencil



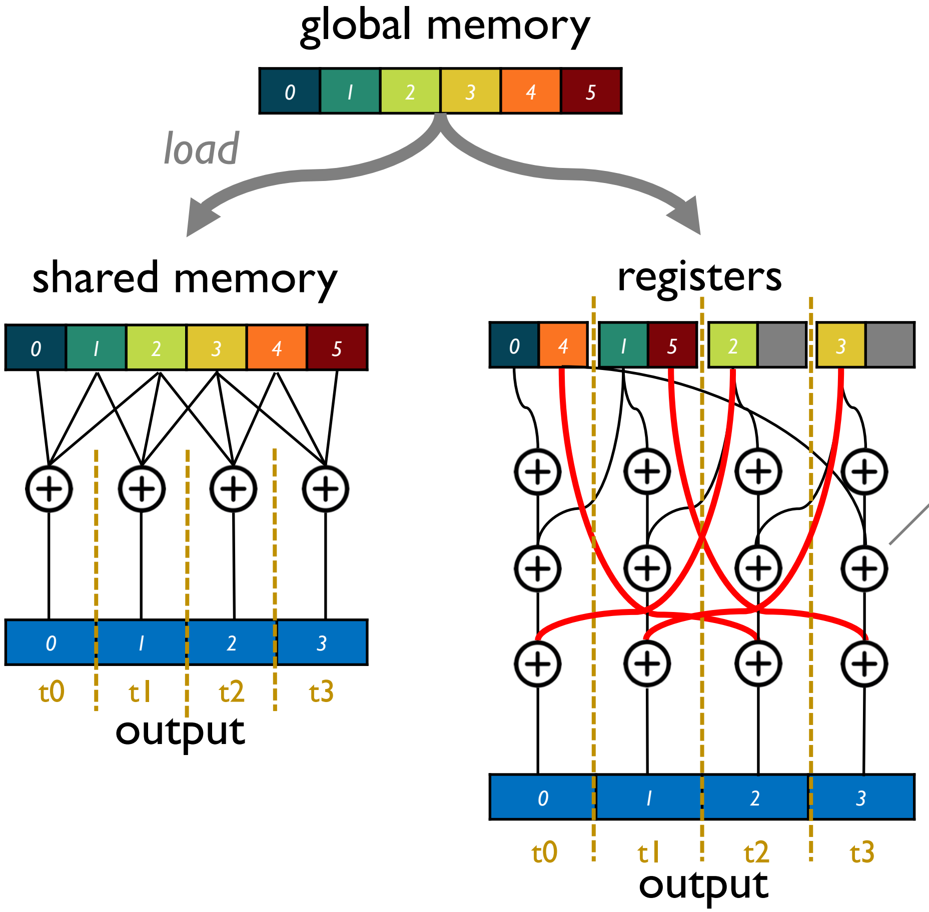
[Ben-Sasson et al. ICS' 16]

Register Cache: Stencil



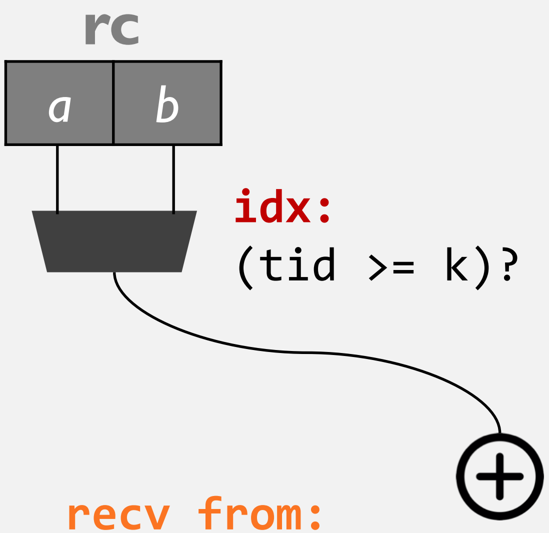
[Ben-Sasson et al. ICS' 16]

Register Cache: Stencil



In each iteration

```
__shfl_sync(mask, rc[idx],
               recv_from)
```



idx:

`(tid >= k)? 0 : 1`

recv_from:

`(tid + k) % warpSize`

[Ben-Sasson et al. ICS' 16]

Swizzle Inventor

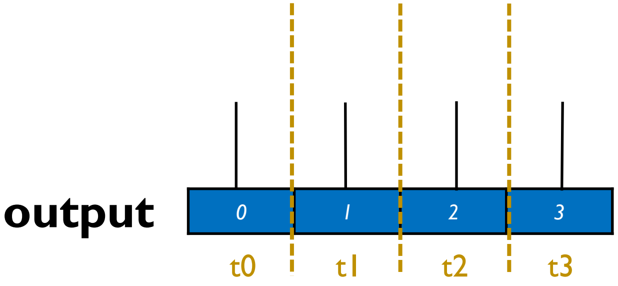
Helps programmers implement swizzle programs by:

- letting them **write program sketches that omit swizzles**
- **automatically synthesizing swizzles** to complete the programs

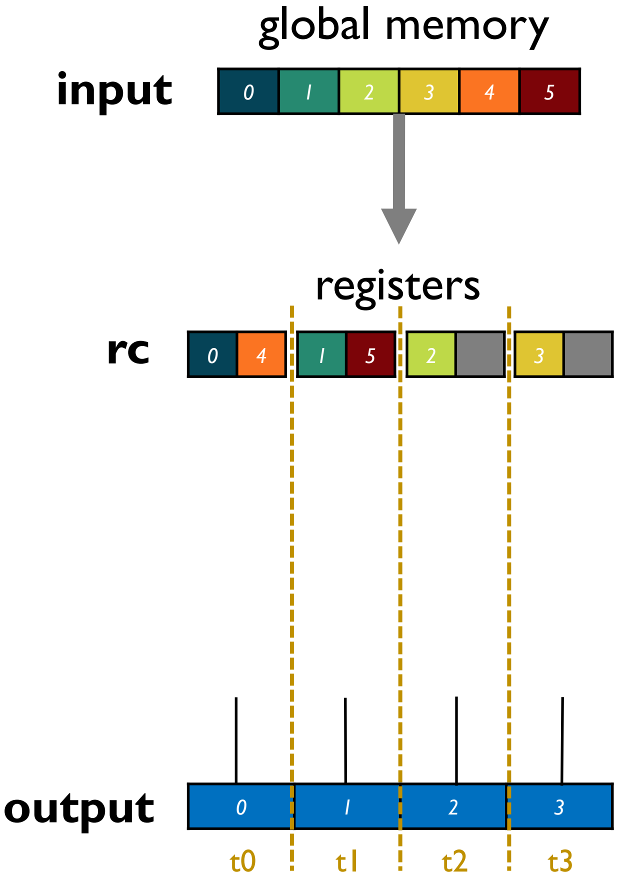
Stencil: Program Sketch

SIMT program

```
rc = load(input, warpOffset,
          /* slice */ 1,
          /* iterations */ 2);

int out = 0;
for(int k = 0; k < 3; k++) {
    int tmp = magic_get(rc);
    out += tmp;
}

output[tid] = out;
```



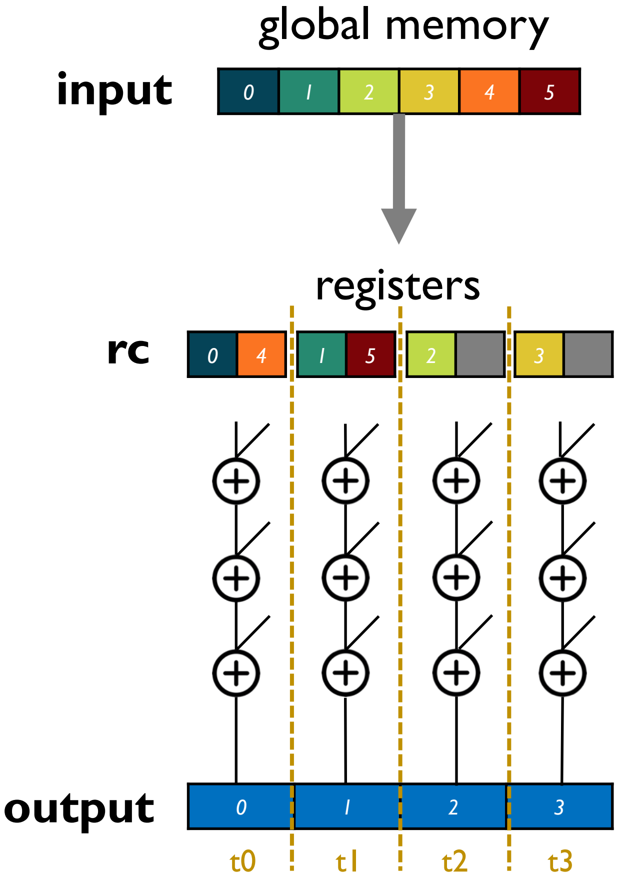
Stencil: Program Sketch



SIMT program

```
rc = load(input, warpOffset,
     /* slice */ 1,
     /* iterations */ 2);

int out = 0;
for(int k = 0; k < 3; k++) {
    int tmp = magic_get(rc);
    out += tmp;
}

output[tid] = out;
```

Stencil: Program Sketch

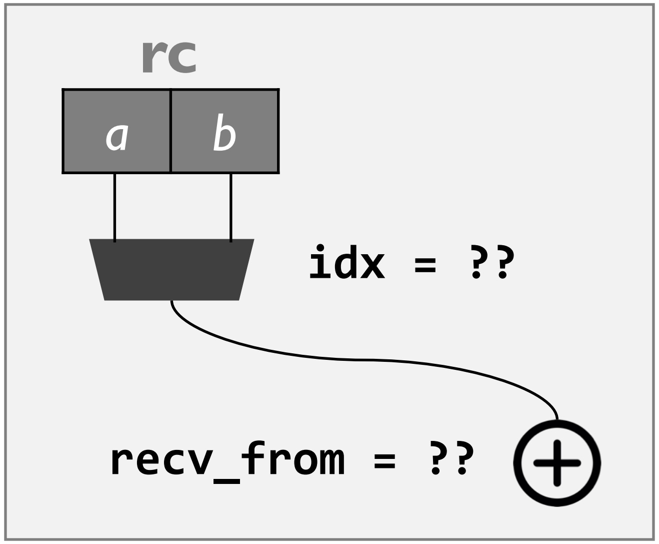


SIMT program

```
rc = load(input, warpOffset,
          /* slice */ 1,
          /* iterations */ 2);

int out = 0;
for(int k = 0; k < 3; k++) {
    int tmp = magic_get(rc);
    out += tmp;
}

output[tid] = out;
```

Stencil: Program Sketch

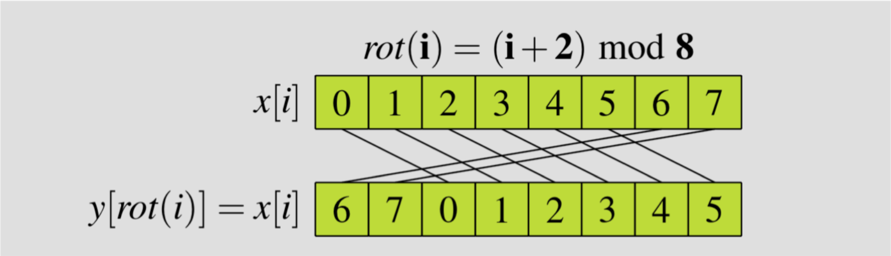
```
int tmp = magic_get(rc); -->

// Choose which input data to send
int idx = ?sw_part(2, tid, k);

// Choose which thread to read from
int recv_from =
?sw_xform(tid, warpSize, k);

// Perform intra-warp shuffle
int tmp = __shfl_sync(FULL_MASK, rc[idx], recv_from);
```



Transformation Swizzle Hole

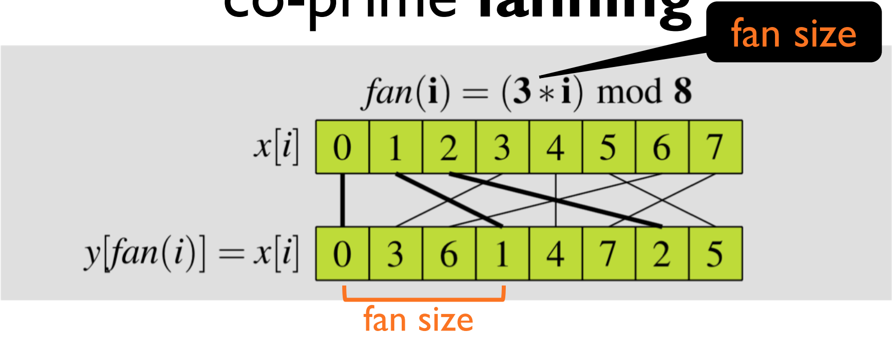
?sw_xform hole defines the search space that contains **grouping** permutations of **fanning** followed by **rotation**.

rotation



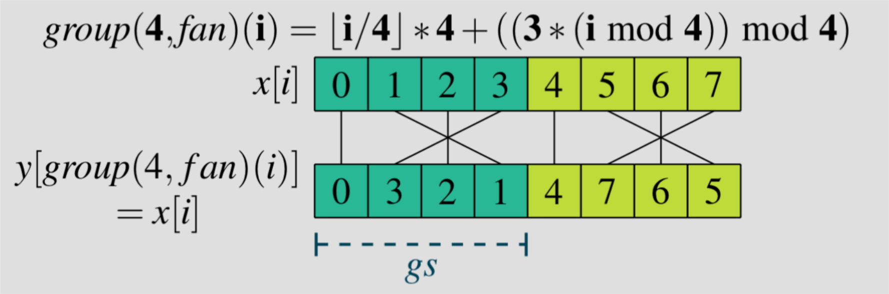
$$rot(\mathbf{i}) = (\mathbf{i} + \mathbf{2}) \bmod \mathbf{8}$$

co-prime **fanning**



$$fan(\mathbf{i}) = (\mathbf{3} * \mathbf{i}) \bmod \mathbf{8}$$

grouping



$$group(\mathbf{4}, fan)(\mathbf{i}) = \lfloor \mathbf{i}/\mathbf{4} \rfloor * \mathbf{4} + ((\mathbf{3} * (\mathbf{i} \bmod \mathbf{4})) \bmod \mathbf{4})$$

Correctness Condition

Spec: sequential program

```
void spec(
    const float *x,
    float *y, int n) {

  for(int i = 0; i < n; i++) {
    int out = 0;
    for(int k = 0; k < 3; k++)
      out += x[i+k];
    y[i] = out;
  }
}
```

$$\exists \boldsymbol{h}\, \forall \boldsymbol{x} \,.\, \boldsymbol{spec}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{n}) \\ \wedge\, \boldsymbol{sketch}(\boldsymbol{h})(\boldsymbol{x}, \boldsymbol{y}', \boldsymbol{n}) \\ \wedge\, \mathbf{y} = \mathbf{y}'$$

Sketch: CUDA sketch

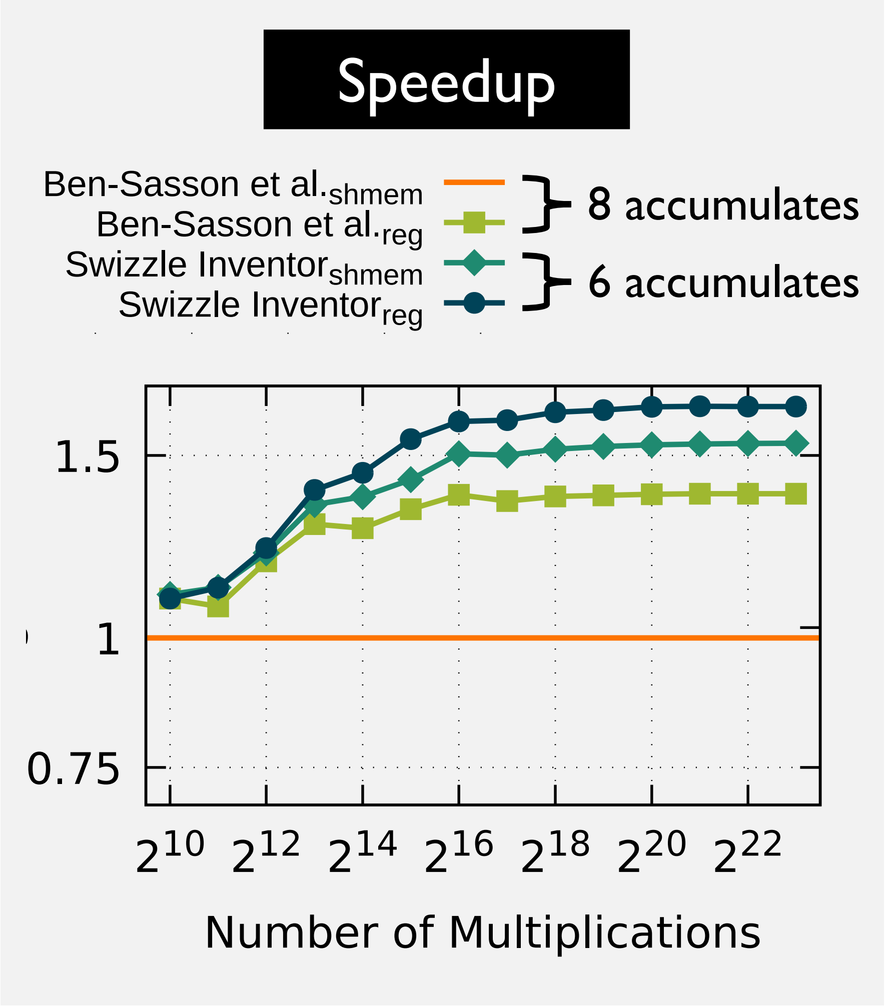
```
__global__ void sketch(
    const float *x,
    float *y, int n) {

  rc = load(x, warpOffset, 1, 2);

  int out = 0;
  for(int k = 0; k < 3; k++) {
    int tmp = magic_get(rc);
    out += tmp;
  }

  y[tid] = out;
}
```

Inventiveness:

Can Swizzle Inventor invent new optimizations?

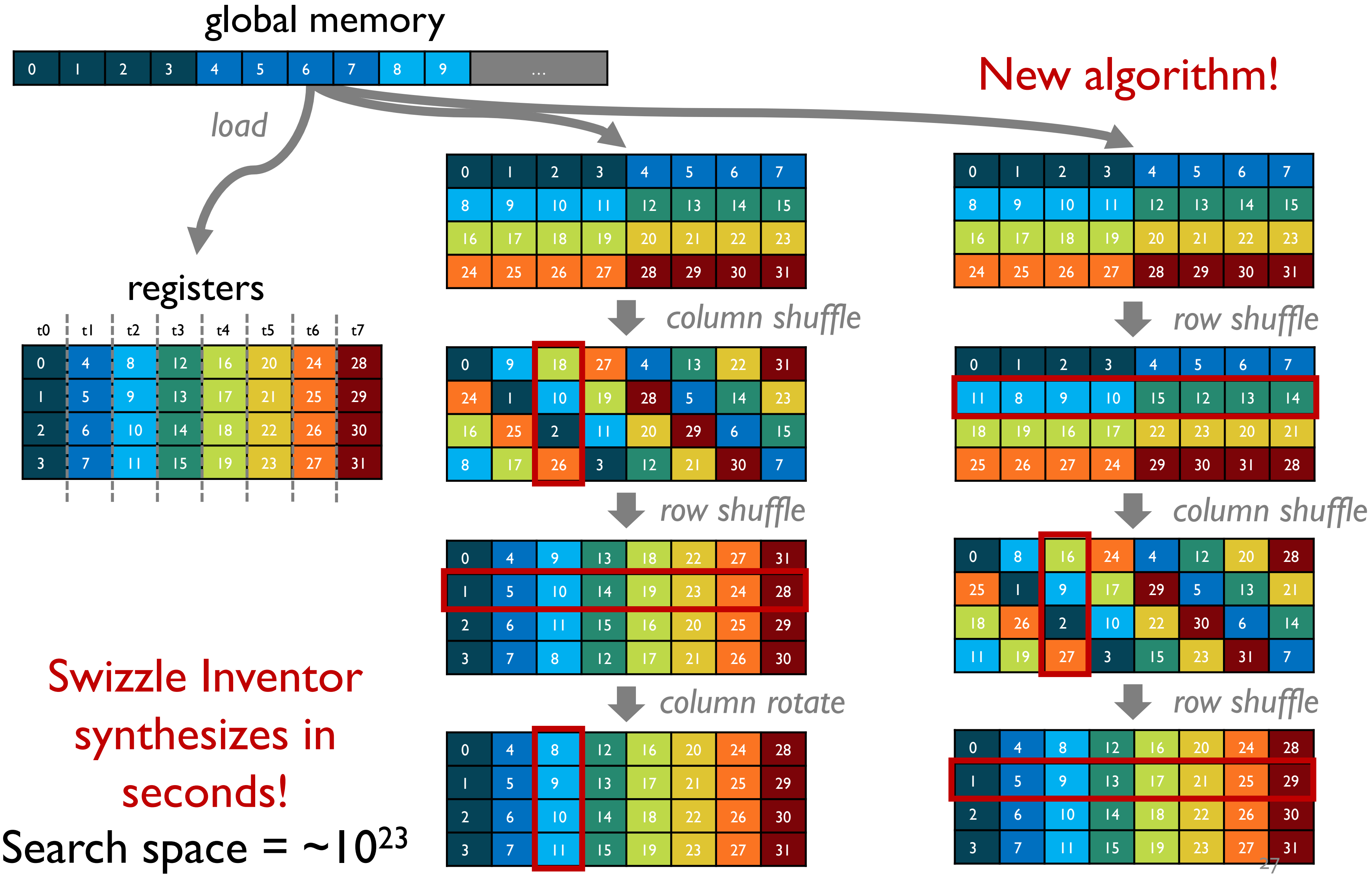
Finite Field Multiplication

```
// Create ans0, ans1, ans2, ans3
acc ans0 = create_accumulator(0, identity, ^, &);
...

for(int k = 0; k < 32; k++) {
  int a0 = __shfl_sync(mask, rA[?sw_part(2,tid,k)],
                       ?sw_xform(tid,32,k));
  int a1 = __shfl_sync(mask, rA[?sw_part(2,tid,k)],
                       ?sw_xform(tid,32,k));
  int b0 = __shfl_sync(mask, rB[?sw_part(2,tid,k)],
                       ?sw_xform(tid,32,k));
  int b1 = __shfl_sync(mask, rB[?sw_part(2,tid,k)],
                       ?sw_xform(tid,32,k));

  // Update ans0
  accumulate(ans0, [a0,b0], ?sw_cond(tid,k));
  accumulate(ans0, [a0,b1], ?sw_cond(tid,k));
  accumulate(ans0, [a1,b0], ?sw_cond(tid,k));
  accumulate(ans0, [a1,b1], ?sw_cond(tid,k));

  // Update ans1, ans2, ans3
  ...
}
```



Baseline: [Ben-Sasson et al. ICS' 16]

Matrix Transposition



global memory

0	1	2	3	4	5	6	7	8	9	…

registers

t0	t1	t2	t3	t4	t5	t6	t7
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

column shuffle

0	9	18	27	4	13	22	31
24	1	10	19	28	5	14	23
16	25	2	11	20	29	6	15
8	17	26	3	12	21	30	7

row shuffle

0	4	9	13	18	22	27	31
1	5	10	14	19	23	24	28
2	6	11	15	16	20	25	29
3	7	8	12	17	21	26	30

column rotate

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

[Catanzaro et al. PPoPP '14]

New algorithm!

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

row shuffle

0	1	2	3	4	5	6	7
11	8	9	10	15	12	13	14
18	19	16	17	22	23	20	21
25	26	27	24	29	30	31	28

column shuffle

0	8	16	24	4	12	20	28
25	1	9	17	29	5	13	21
18	26	2	10	22	30	6	14
11	19	27	3	15	23	31	7

row shuffle

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Swizzle Inventor synthesizes in seconds!

Search space = ~10^{23}

Swizzle Inventor

semi-automatic
bypass rewrite rules & heuristics

GreenThumb
Superoptimization

fully-automatic
bypass rewrite rules & heuristics

AutoX

fully-automatic
bypass heuristics

Swizzle
Inventor

semi-automatic
bypass rewrite rules & heuristics

GreenThumb
Superoptimization

fully-automatic
bypass rewrite rules & heuristics

AutoX

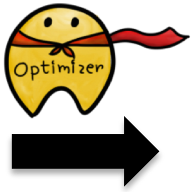
fully-automatic
bypass heuristics

ARM

register-based ISA

gcc –O3

```
cmp   r1, #0
mov   r3, r1, asr #31
add   r2, r1, #7
mov   r3, r3, lsr #29
movge r2, r1
ldrb  r0, [r0, r2, asr #3]
add   r1, r1, r3
and   r1, r1, #7
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r0, #1
```



82% speedup

```
asr   r3, r1, #2
add   r2, r1, r3, lsr #29
ldrb  r0, [r0, r2, asr #3]
and   r3, r2, #248
sub   r3, r1, r3
asr   r1, r0, r3
and   r0, r1, #1
```

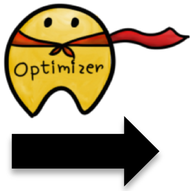
GreenArrays

stack-based ISA

Expert's

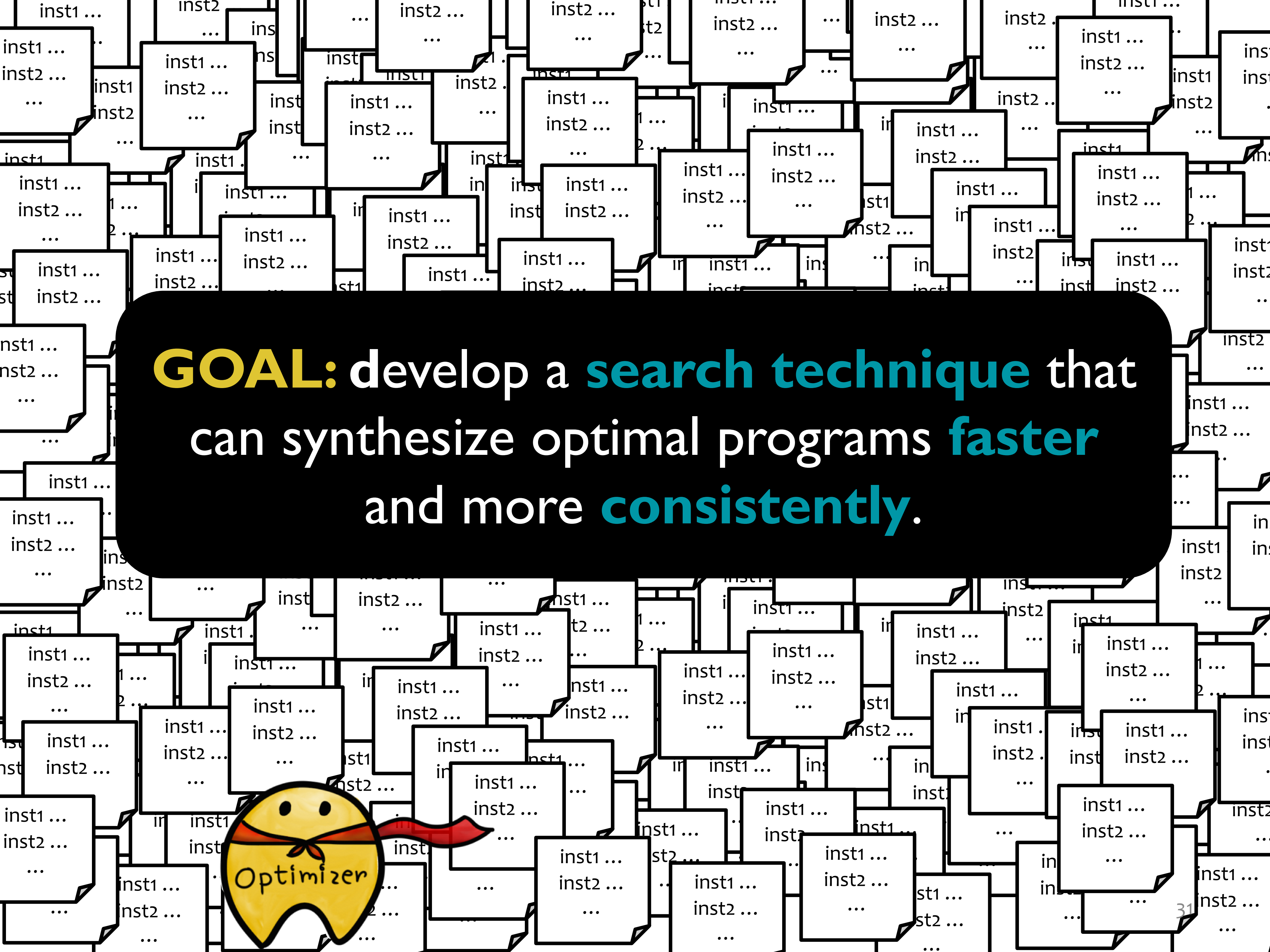
```
push over - push and
pop pop and over
0xffff or and or
```

Precondition: top 3 elements in the stack are <= 0xffff



2.5X speedup

```
dup push or and pop or
```



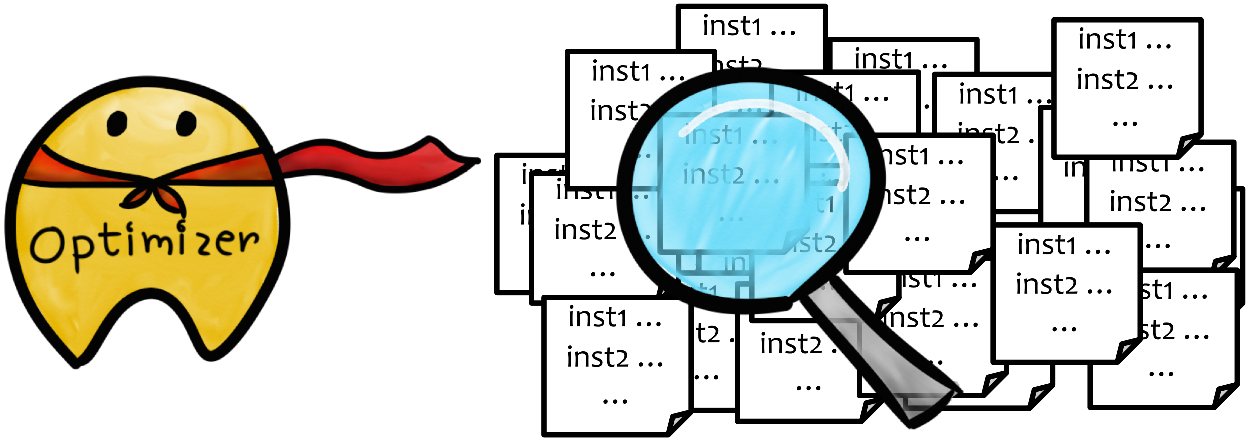
GOAL: develop a **search technique** that can synthesize optimal programs **faster** and more **consistently**.

We developed …

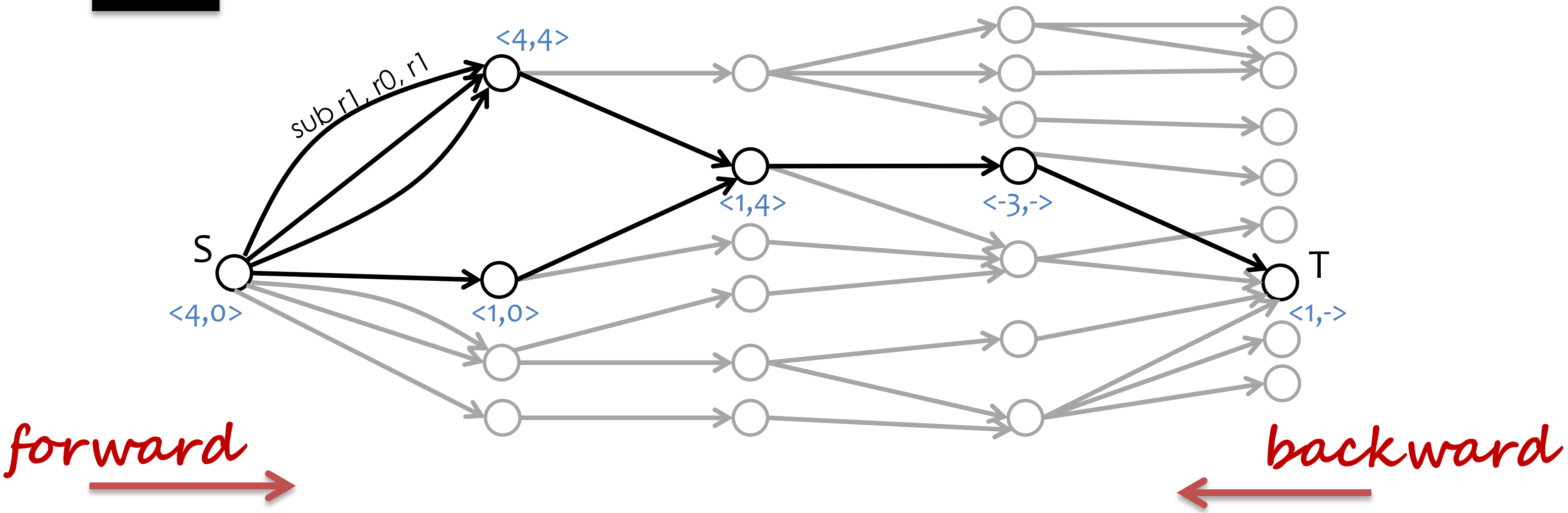
1 **Lens**

enumerative
search algorithm

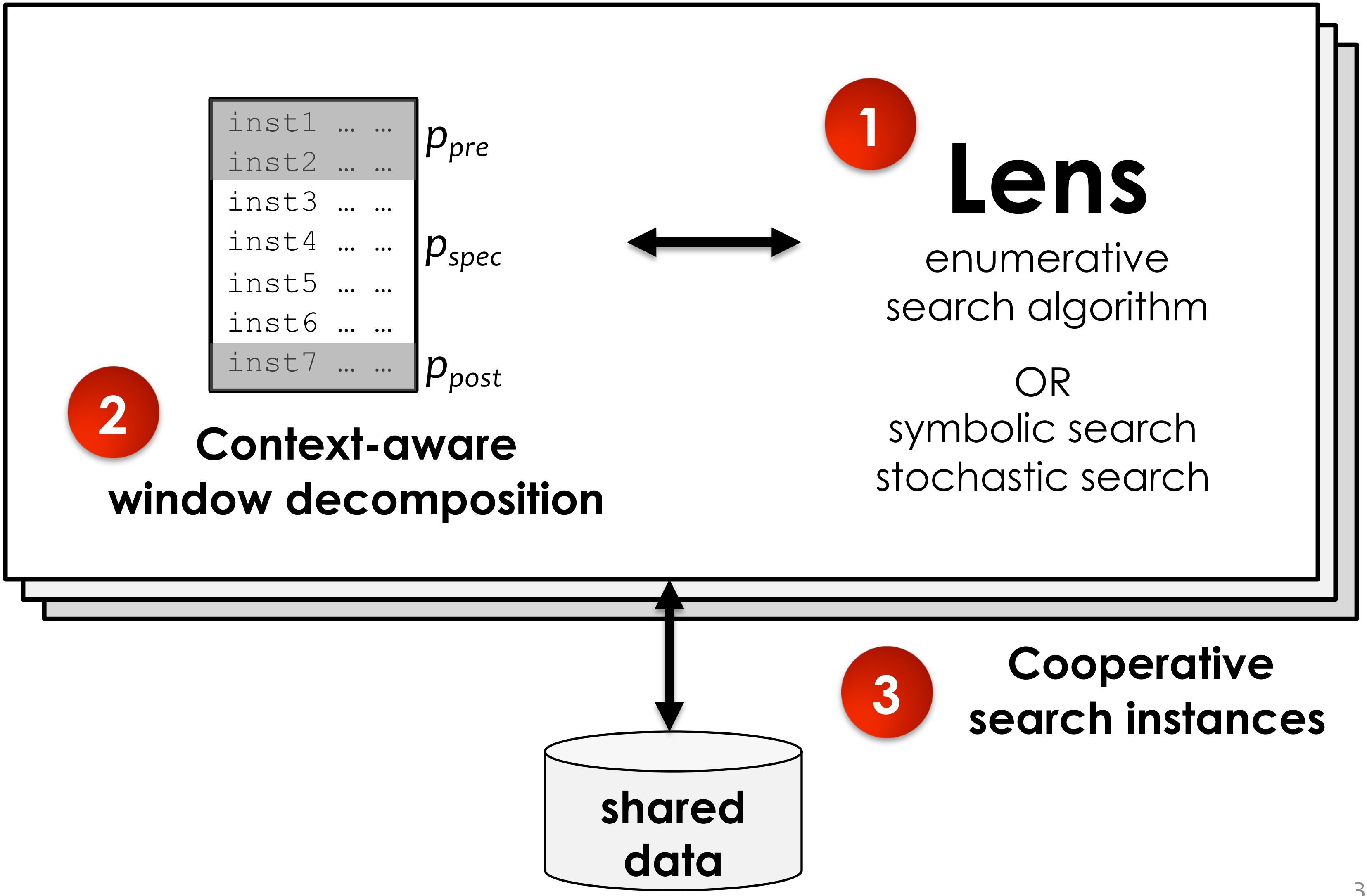
Lens Enumerative Search Algorithm



=



We developed …



```
inst1 … …
inst2 … …
inst3 … …
inst4 … …
inst5 … …
inst6 … …
inst7 … …
```

p_{pre}

p_{spec}

p_{post}

2 Context-aware window decomposition

1 Lens

enumerative search algorithm

OR

symbolic search

stochastic search

3 Cooperative search instances

shared data

Runtime Speedup

Runtime speedup over **gcc –O3** on an actual **ARM Cortex-A9**

Benchmarks Hacker's Delight, WiBench (wireless system kernel benchmarks), MiBench (embedded system kernel benchmarks)

Program	Search time (s)	gcc –O3 length	Output length	Runtime speedup on ARM Cortex-A9
p18	9	7	4	**2.11**
p21	1139	6	5	**1.81**
p23	665	18	16	**1.48**
p24	151	7	4	**2.75**
p25	2	11	1	**17.80**
WB-txrate5a	32	9	8	**1.31**
WB-txrate5b	66	8	7	**1.29**
MB-bitarray	612	10	6	**1.82**
MB-bitshift	5	9	8	**1.11**
MB-bitcnt	645	27	19	**1.33**
MB-susan-391	32	30	21	**1.26**

GreenThumb Framework

Provide cooperative search strategy.

Enable rapid retargeting of the superoptimizer to a new ISA.

github.com/mangpo/greenthumb

Supported ISAs: GreenArrays, ARM, subset of LLVM

LinkiTools’ S10 (https://linki.tools/s10.html): RISC-V

Swizzle Inventor

semi-automatic
bypass rewrite rules & heuristics

GreenThumb
Superoptimization

fully-automatic
bypass rewrite rules & heuristics

Auto

fully-automatic
bypass heuristics

Swizzle Inventor

semi-automatic
bypass rewrite rules & heuristics

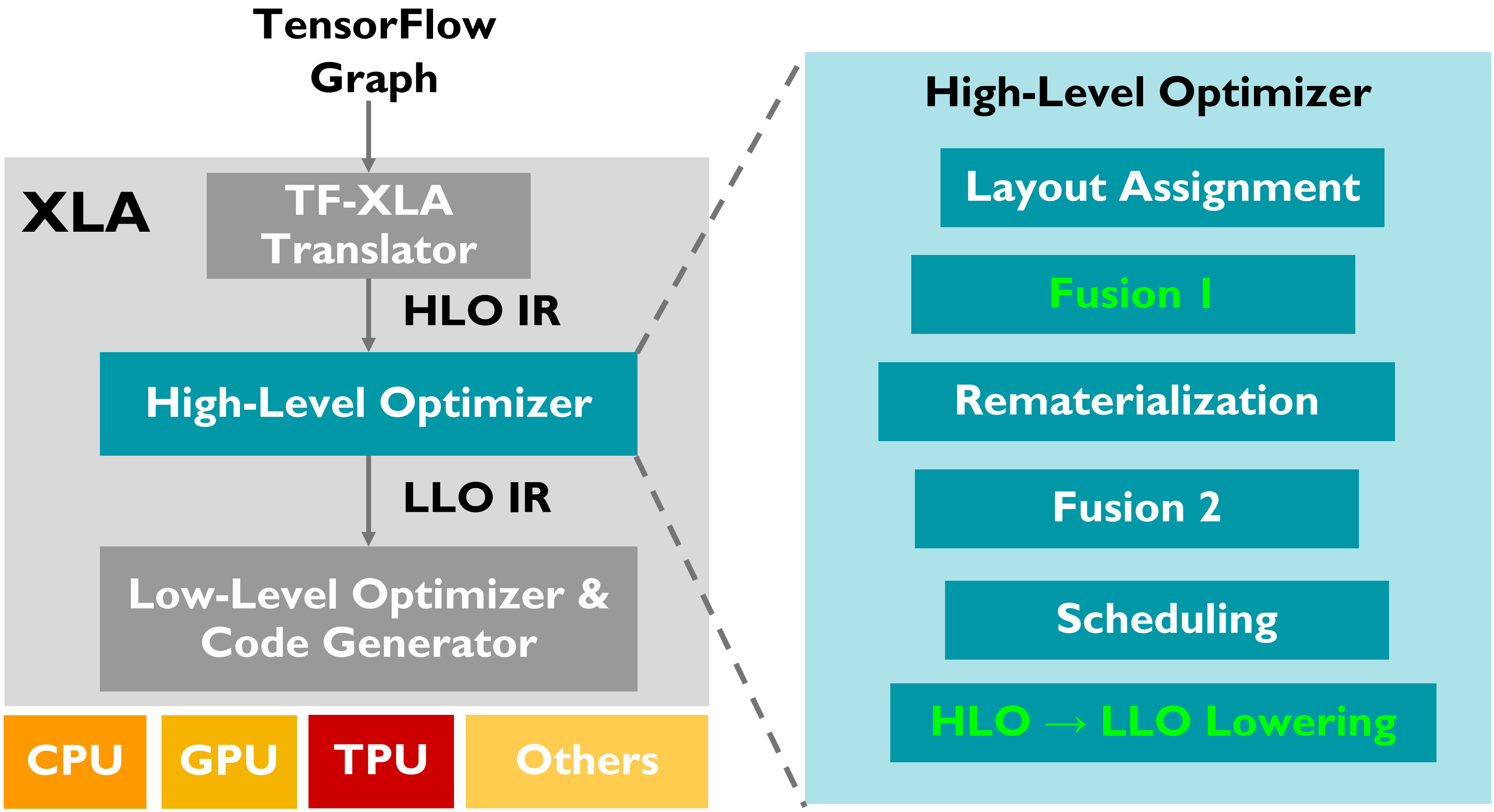
GreenThumb
Superoptimization

fully-automatic
bypass rewrite rules & heuristics

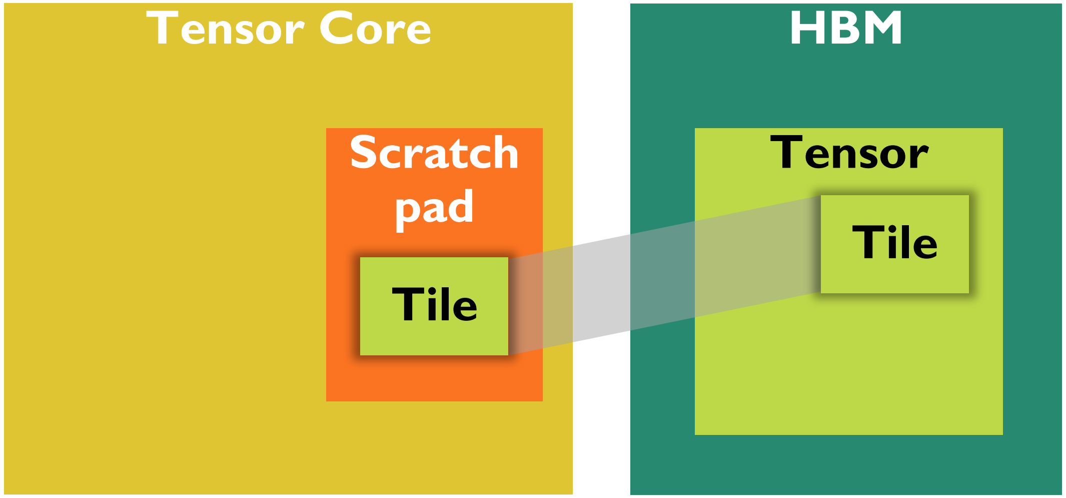
AutoX

fully-automatic
bypass heuristics

XLA: Accelerated Linear Algebra



Tile Size in Lowering Pass



TPUs process one (fused) tensor op at a time.

- Entire tensors don’t fit in scratchpad.
- To process one tile of an output tensor, copy input tiles into scratchpad.
- Store intermediates in scratchpad.

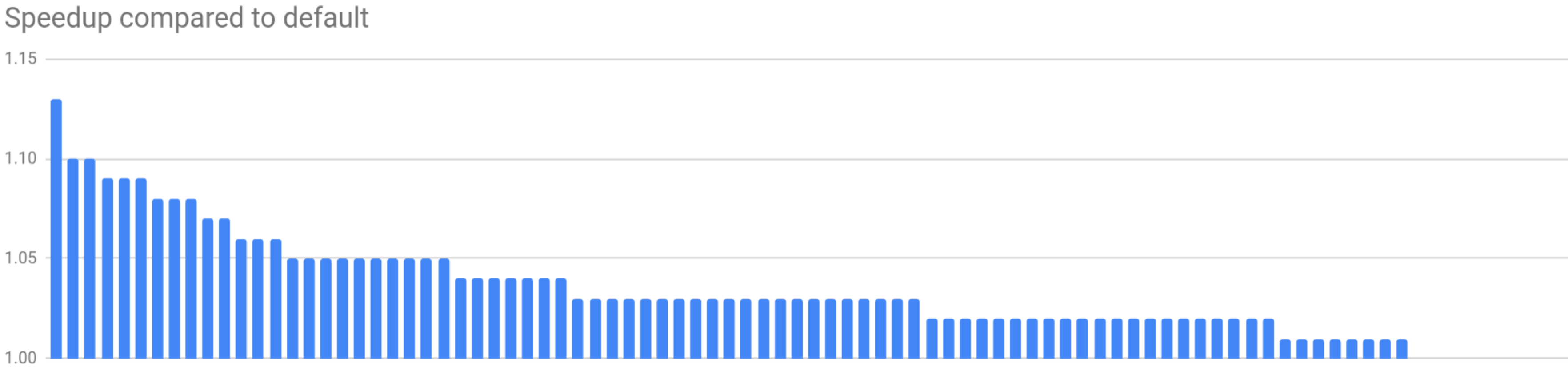
Autotuning Tile Size

Autotuner

- Exhaustive search (100-1M choices per program)
- Evaluate by running on real hardware.
- Fast mode: tune subset of candidates.

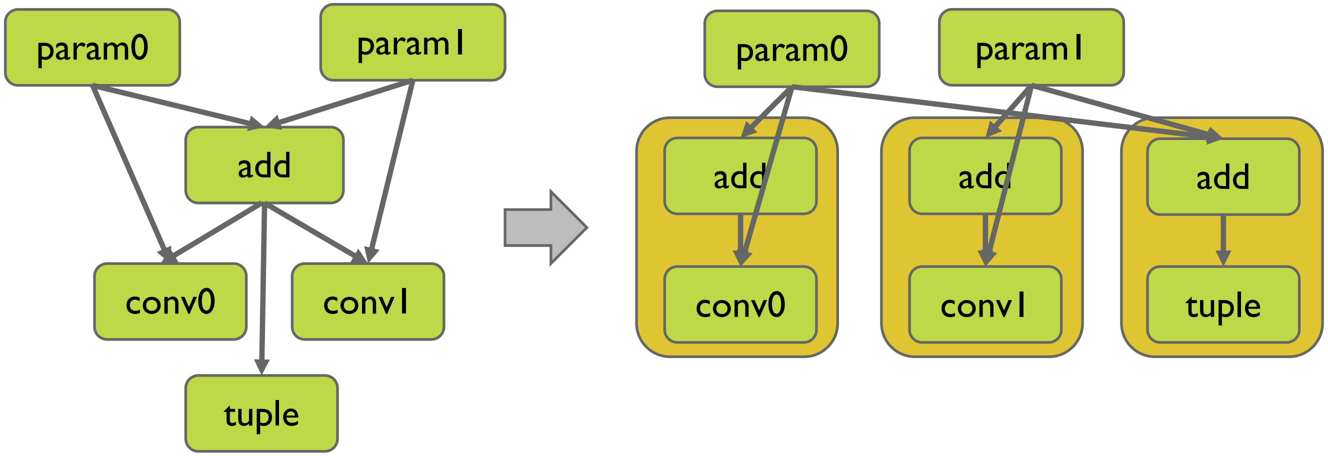
Result:

24 benchmarks gain >= 5% improvement



Fusion Decisions

Example:



Fusion configuration:

- fuse or do-not-fuse per **node**
- If a node is marked fuse, it is fused into all its consumers.
- 100 - 100,000 nodes per graph

Autotuning Fusion Decisions

Autotuner

- Partition graph into <=1000-node clusters.
- Run simulated annealing in each cluster.
- Start the search from the default or random configuration.

Highlighted results

- 9% speedup on search ranking inference model
- 13% speedup on TPU compute time of ad recommendation training model
- 6-9% speedup on ML Perf inference models

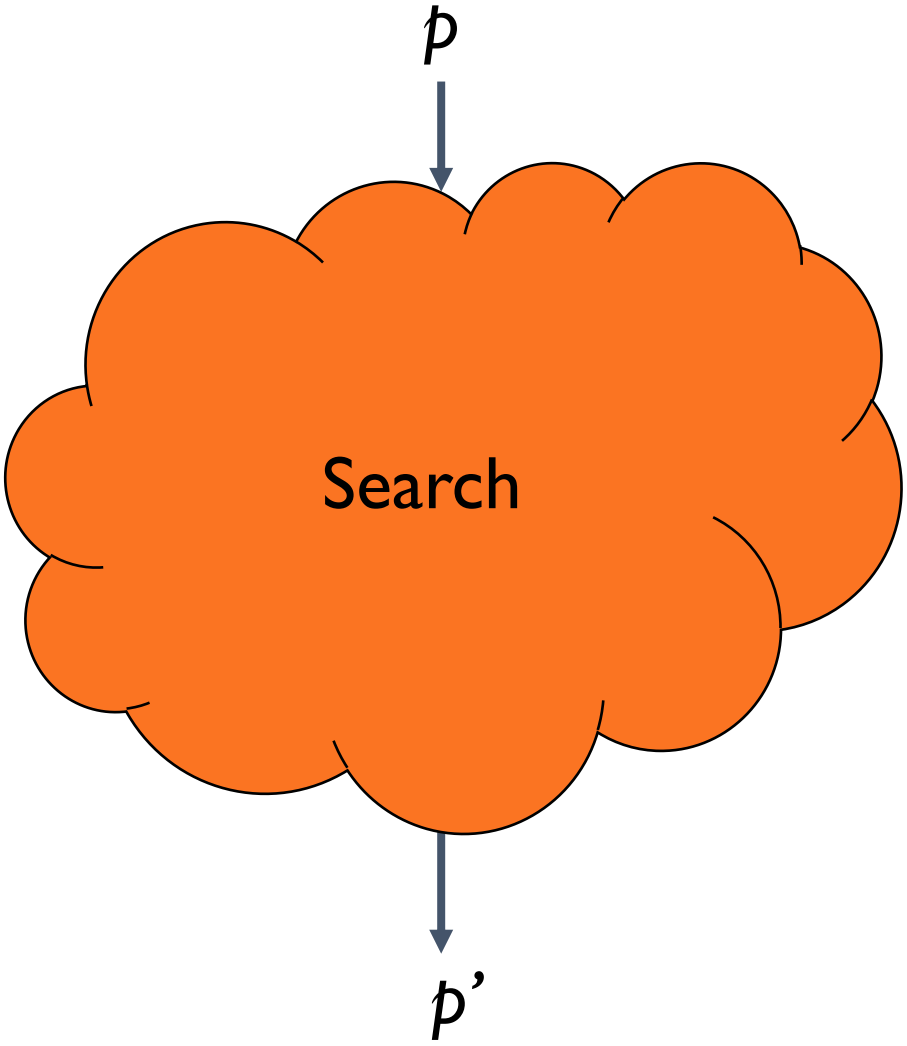
Swizzle Inventor	semi-automatic bypass rewrite rules & heuristics
GreenThumb Superoptimization	fully-automatic bypass rewrite rules & heuristics
AutoX	fully-automatic bypass heuristics

Problems of Search-Based Compilers

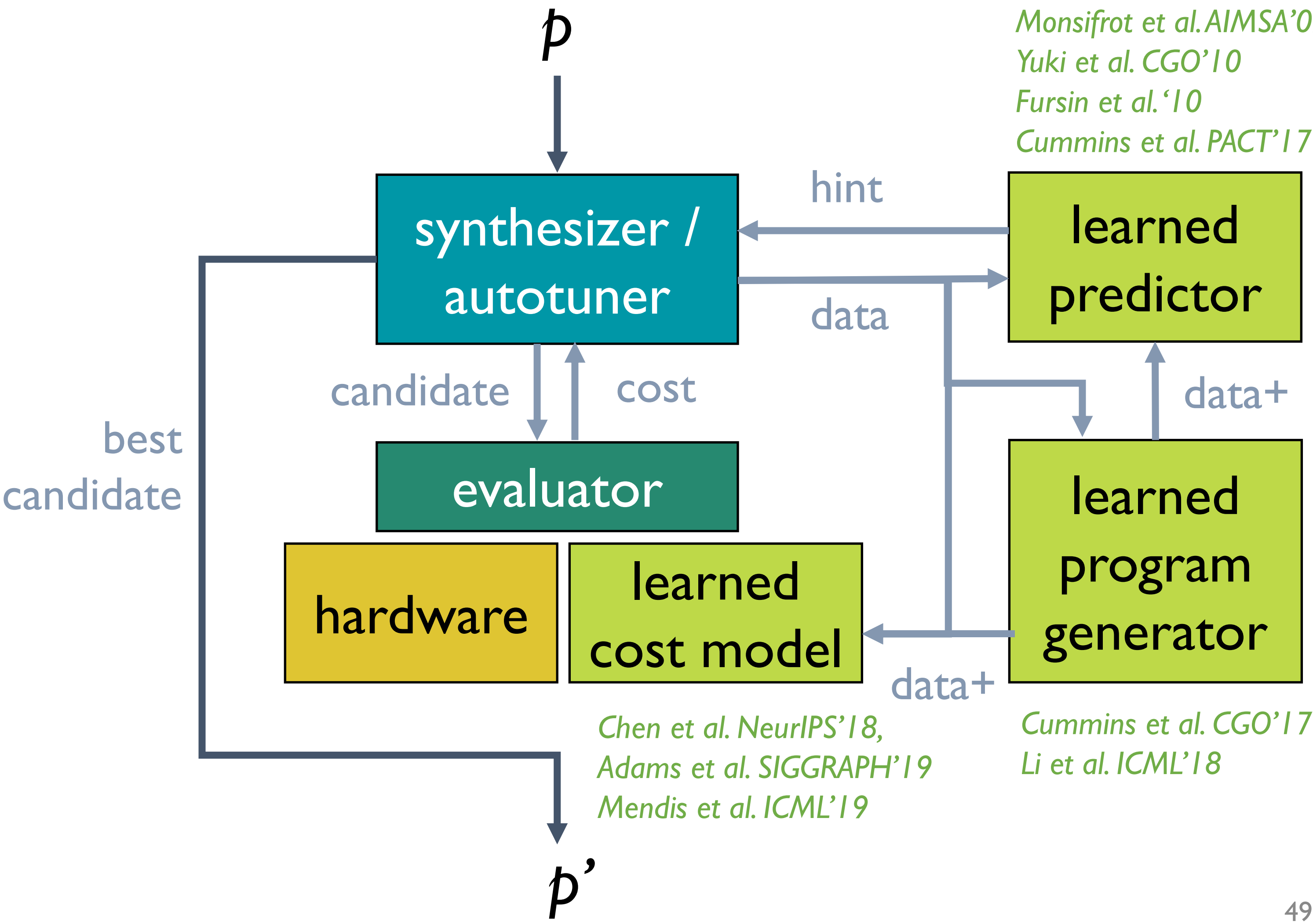
slow

do not learn from past experience

Toward Self-Evolving Compilers



Toward Self-Evolving Compilers



Monsifrot et al. AIMSA'02
Yuki et al. CGO'10
Fursin et al.'10
Cummins et al. PACT'17

Chen et al. NeurIPS'18,
Adams et al. SIGGRAPH'19
Mendis et al. ICML'19

Cummins et al. CGO'17
Li et al. ICML'18

Toward Self-Evolving Compilers

Is it possible to build
a self-evolving compiler that is
as fast as a heuristics-base compiler
and **produces better code**?

Collaborators

Swizzle Inventor

Archibald Samuel Elliott
An Wang
Abhinav Jangda
Bastian Hagedorn
Henrik Barthels
Samuel J. Kaufman
Vinod Grover
Emina Torlak
Rastislav Bodik

GreenThumb

Aditya Thakur
Rastislav Bodik
Dinakar Dhurjati

Auto

Mike Burrows
Bjarke Roune
Amit Sabne
Sam J. Kaufman
Cambridge Yang
Dimitrios Vytiniotis
Dominik Grewe