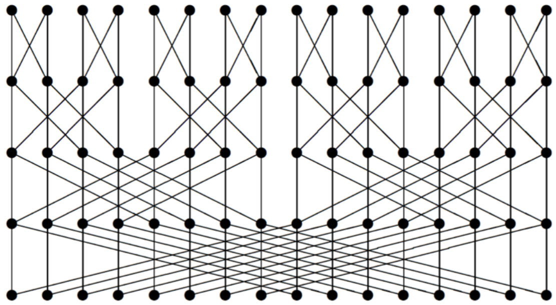
Swizzle Inventor

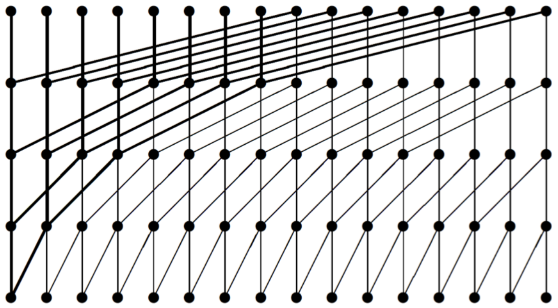
Phitchaya Mangpo Phothilimthana	UC Berkeley (now at Google Brain)
Archibald Samuel Elliott	University of Washington
An Wang	University of Washington
Abhinav Jangda	University of Massachusetts Amherst
Bastian Hagedorn	University of Münster
Henrik Barthels	AICES, RWTH Aachen University
Samuel J. Kaufman	University of Washington
Vinod Grover	NVIDIA
Emina Torlak	University of Washington
Rastislav Bodik	University of Washington

Swizzle

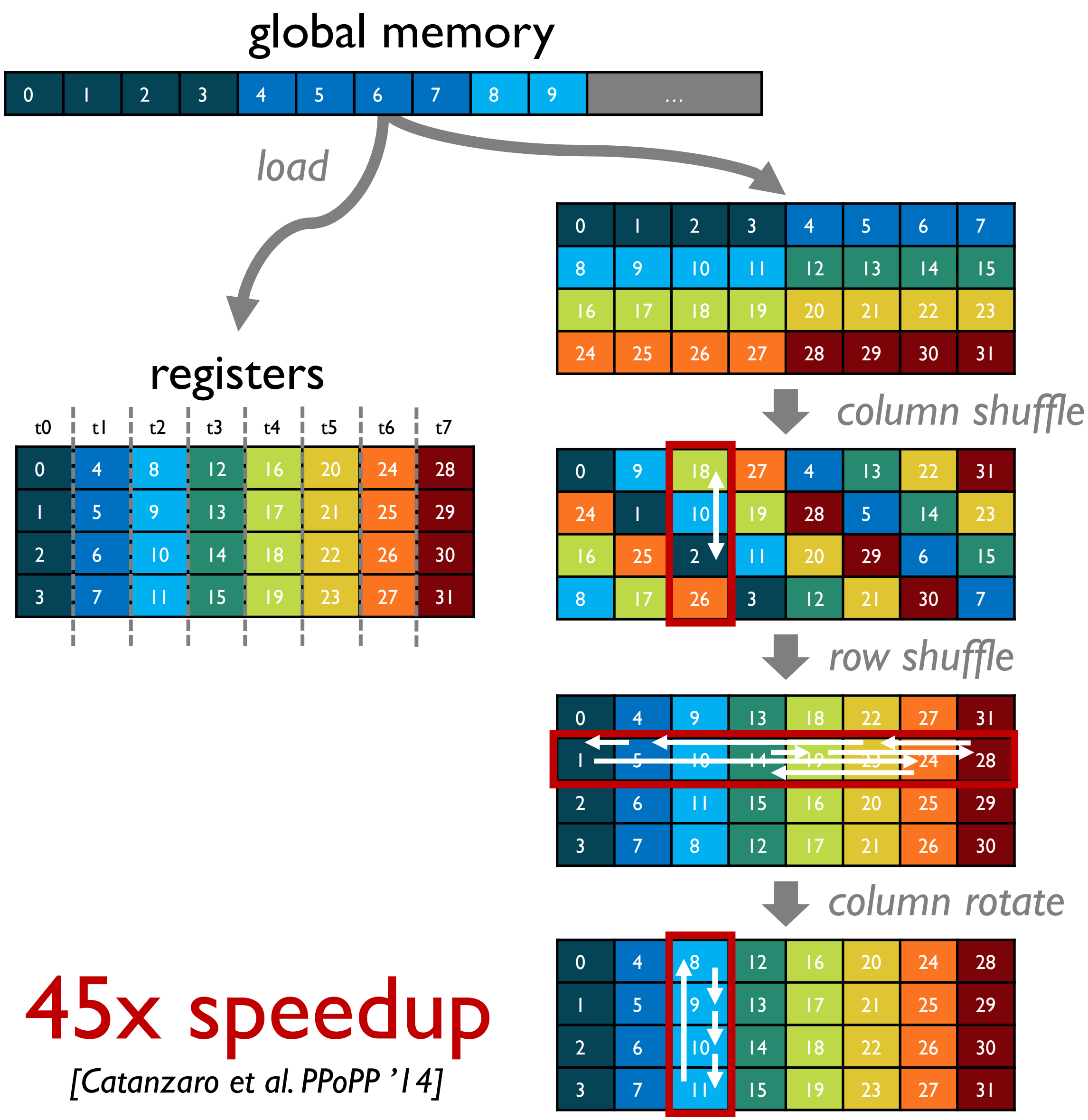
non-trivial movement of data or
non-trivial mapping of computations
to **hardware resources** and **loop iterations**

for dramatic performance improvement



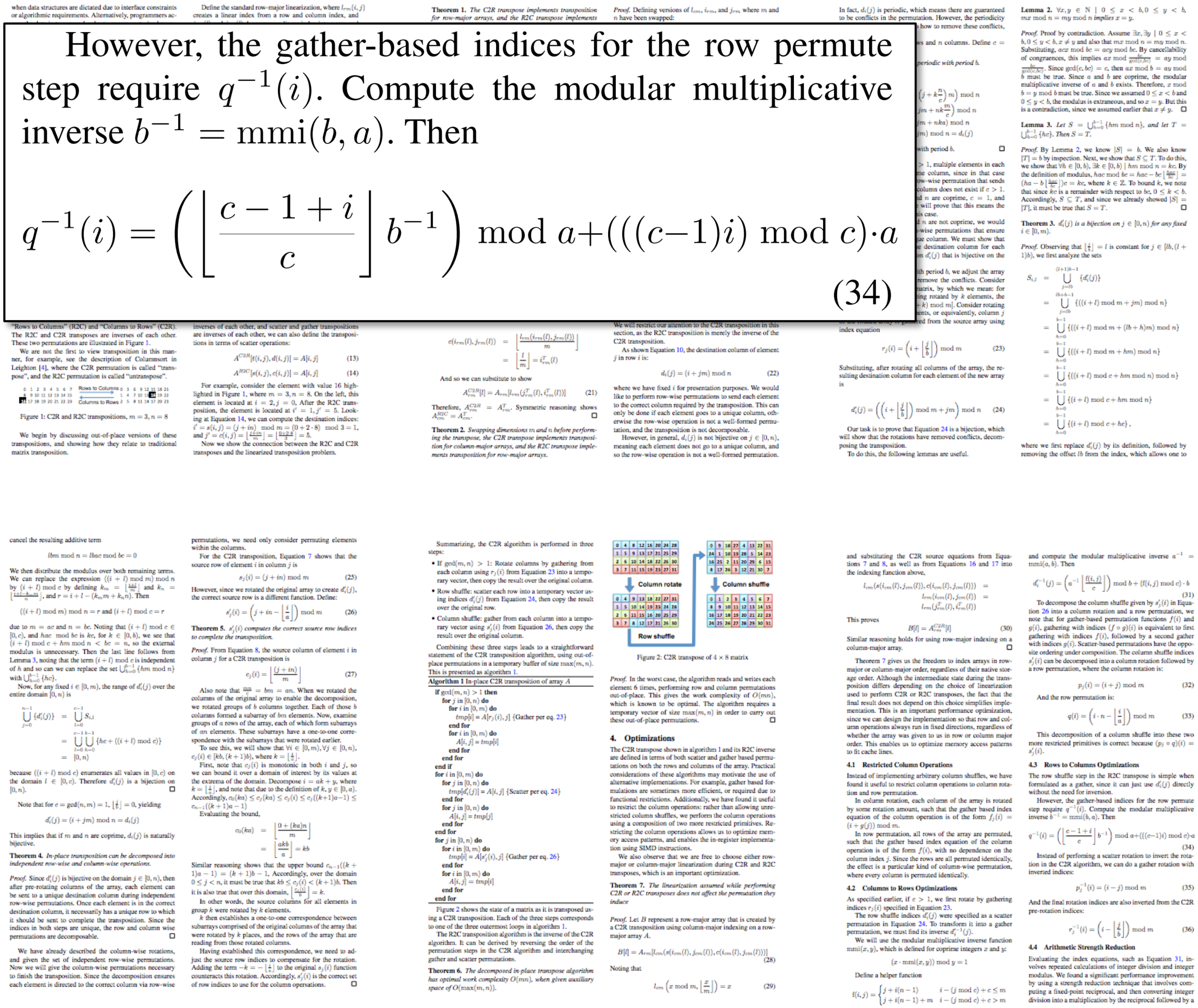


Load Array of Struct in GPU



45x speedup

[Catanzaro et al. PPoPP '14]

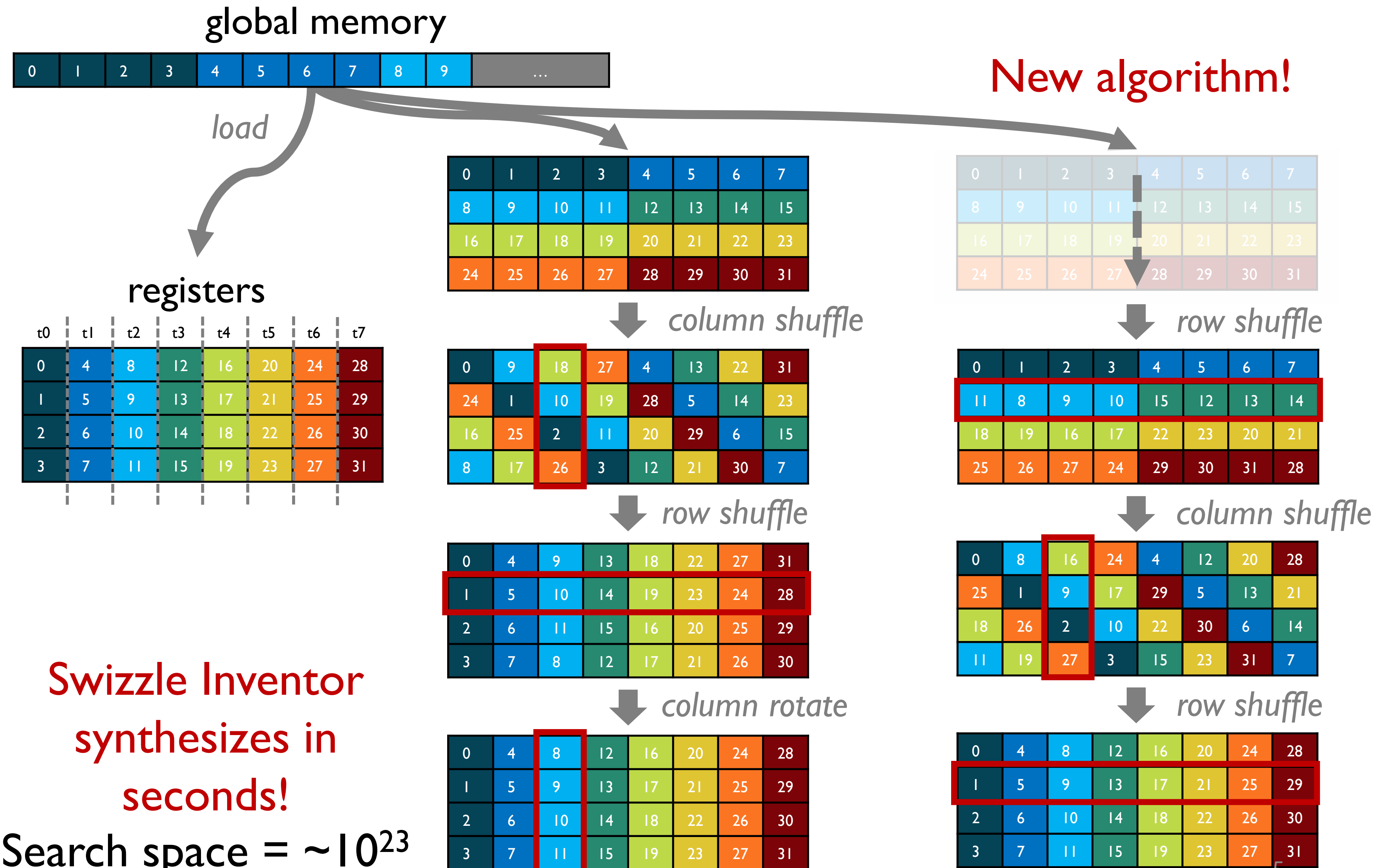


However, the gather-based indices for the row permute step require $q^{-1}(i)$. Compute the modular multiplicative inverse $b^{-1} = \text{mmi}(b, a)$. Then

$$q^{-1}(i) = \left(\left\lfloor \frac{c-1+i}{c} \right\rfloor b^{-1} \right) \bmod a + (((c-1)i) \bmod c) \cdot a \tag{34}$$

[Catanzaro et al. PPoPP '14]

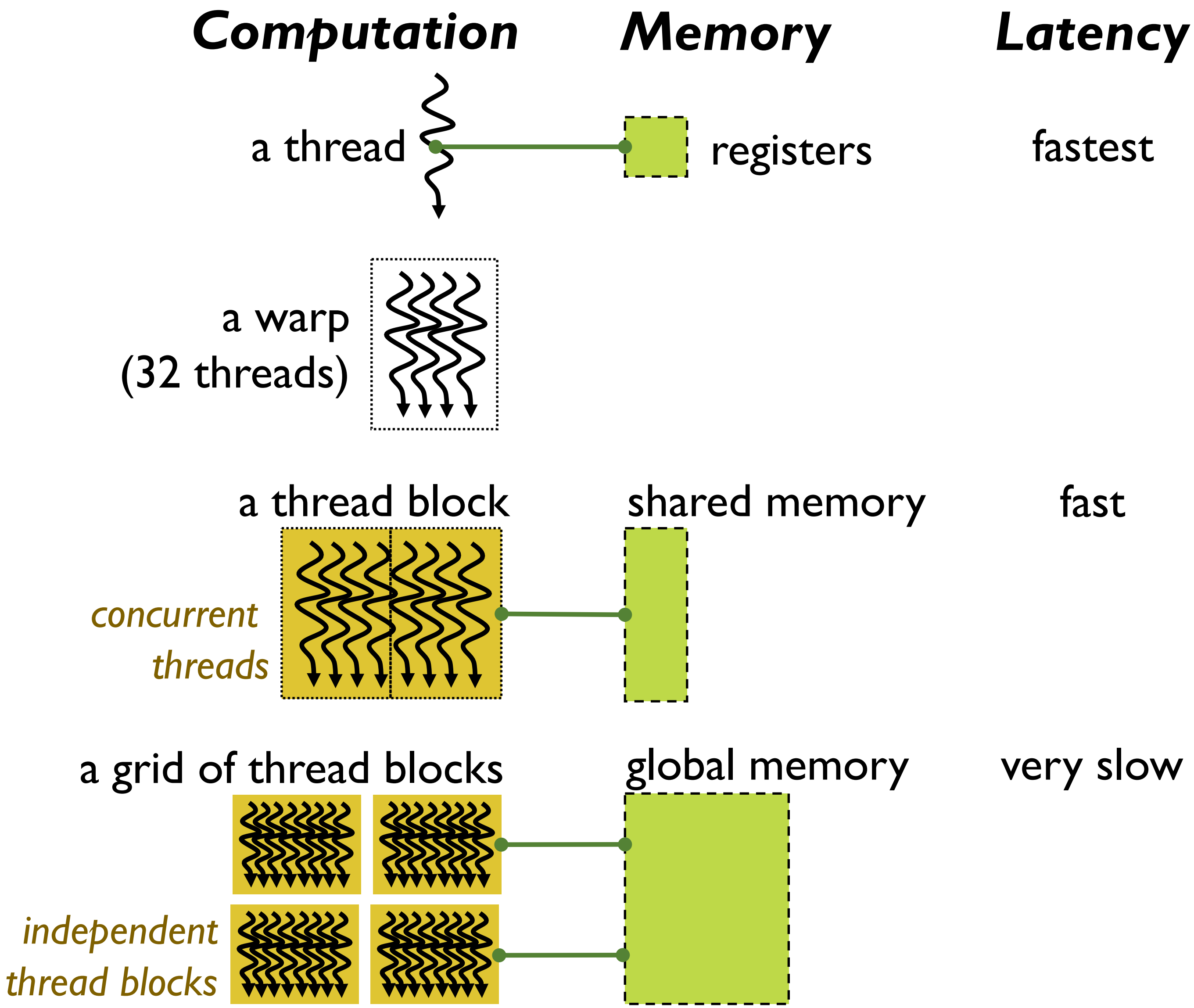
Load Array of Struct in GPU



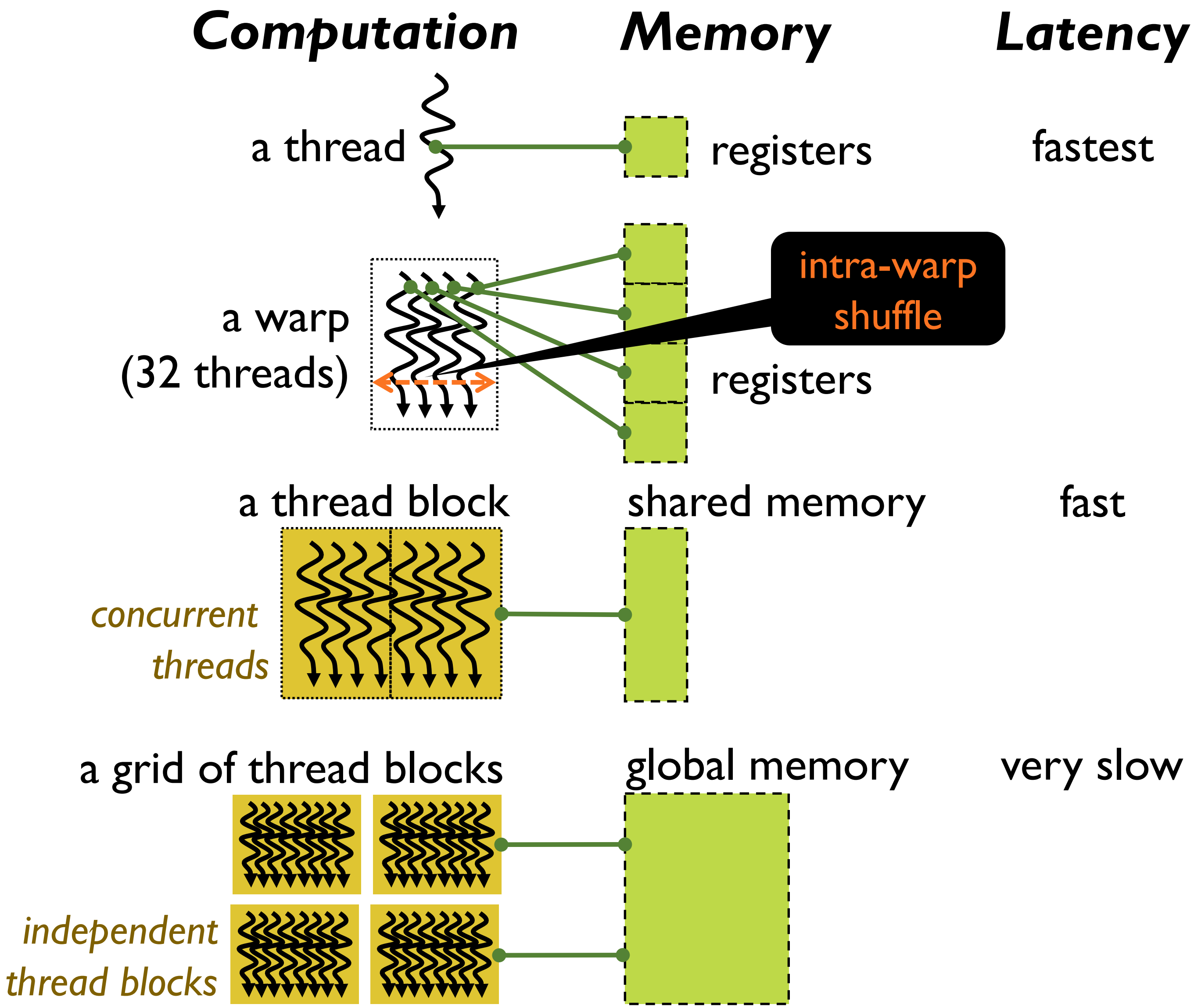
Swizzle Inventor synthesizes in seconds!

Search space = ~10^{23}

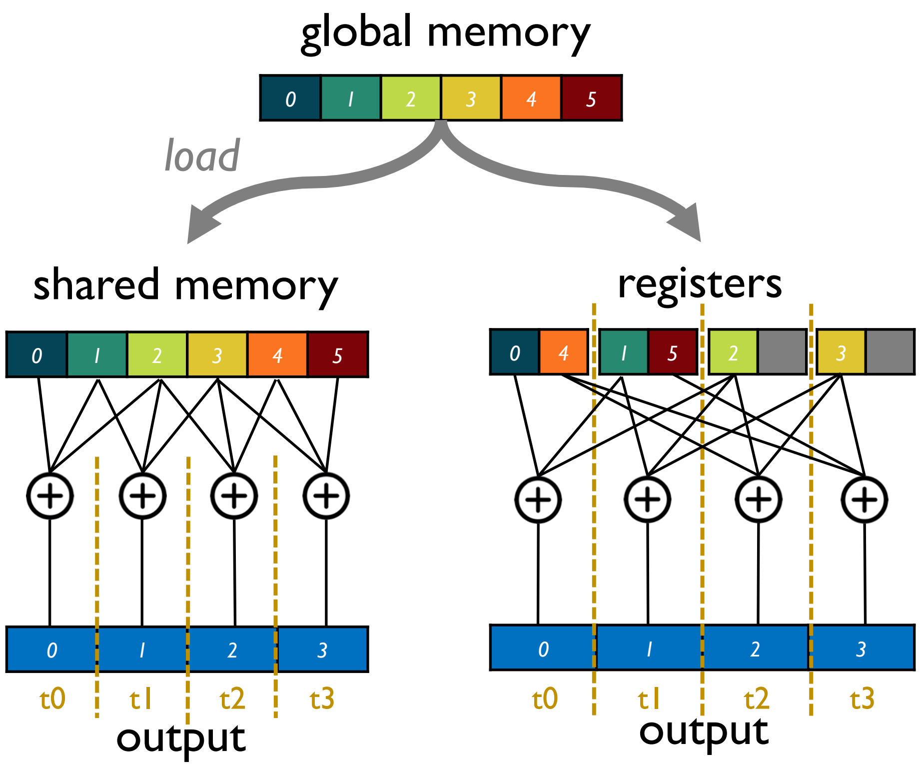
GPU Architecture Basic



GPU Architecture Basic

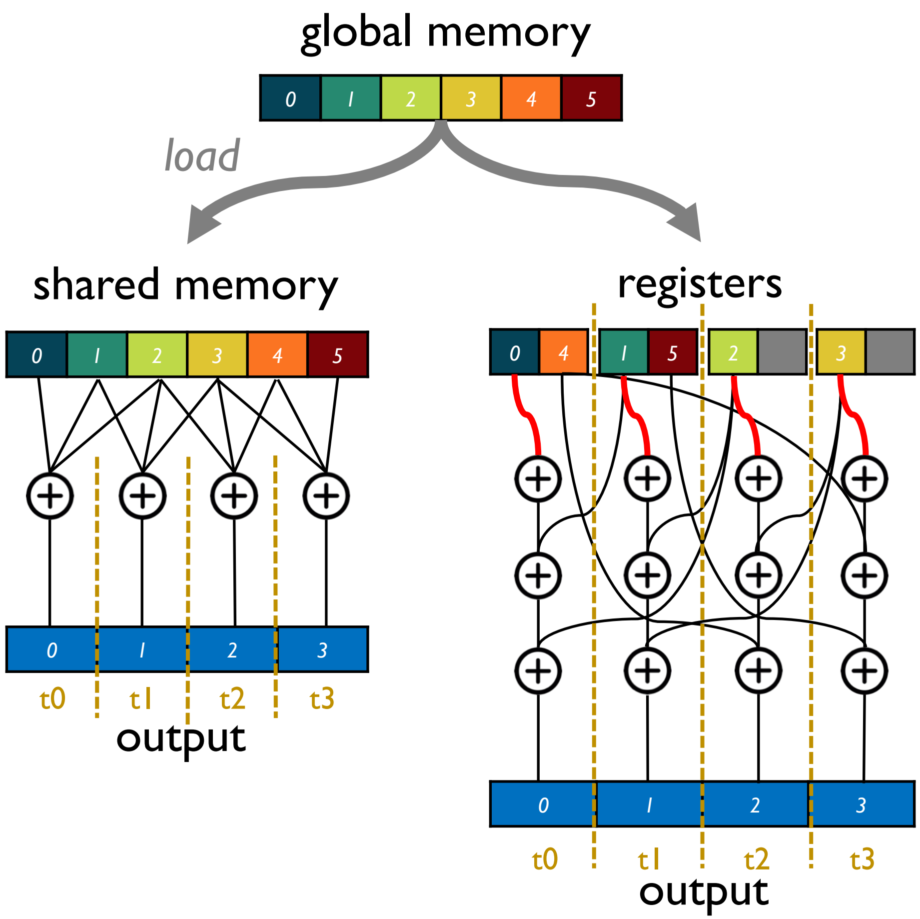


Register Cache: Stencil



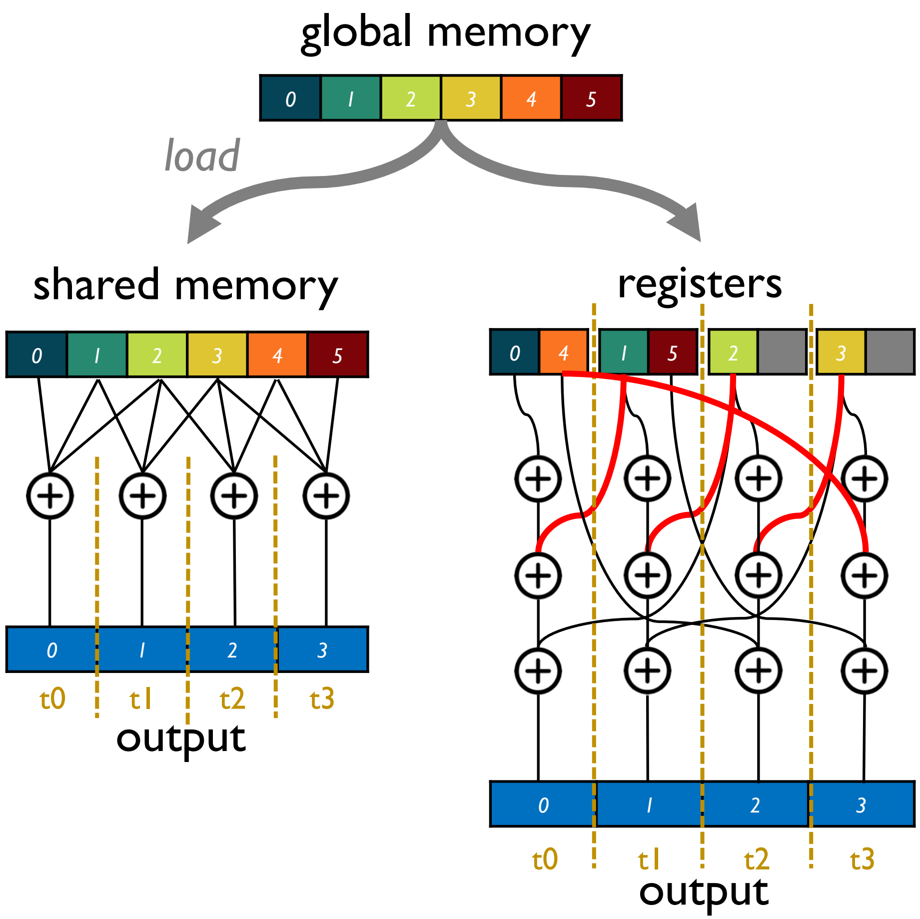
[Ben-Sasson et al. ICS' 16]

Register Cache: Stencil



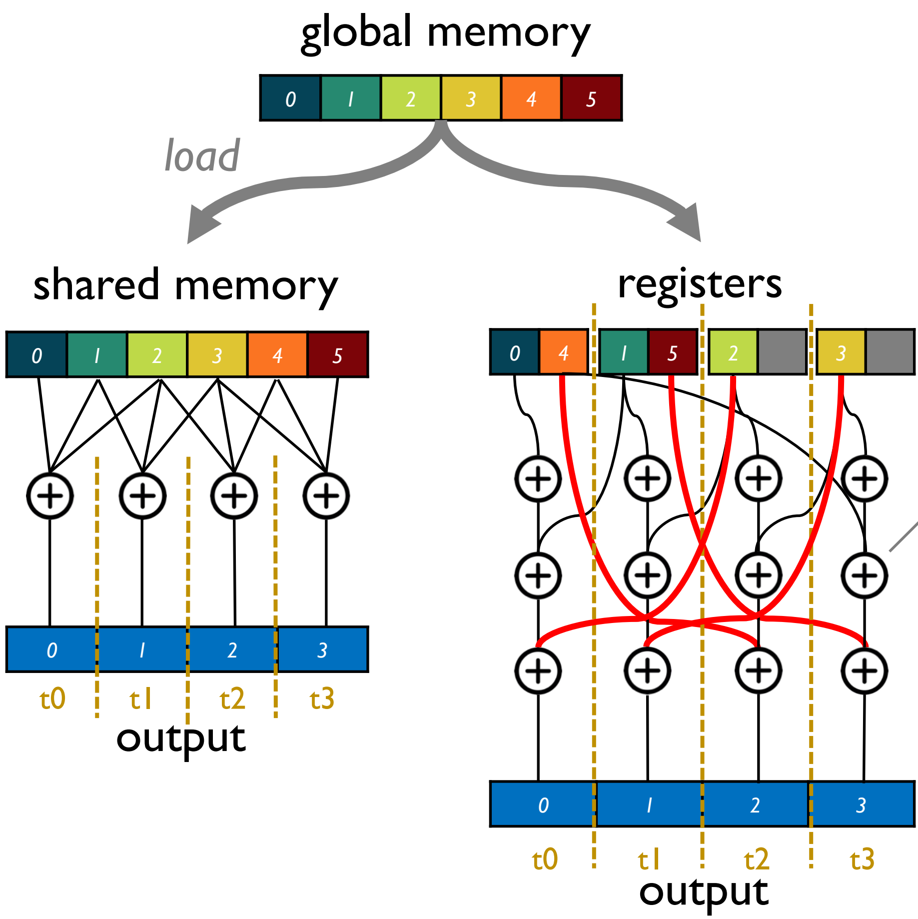
[Ben-Sasson et al. ICS' 16]

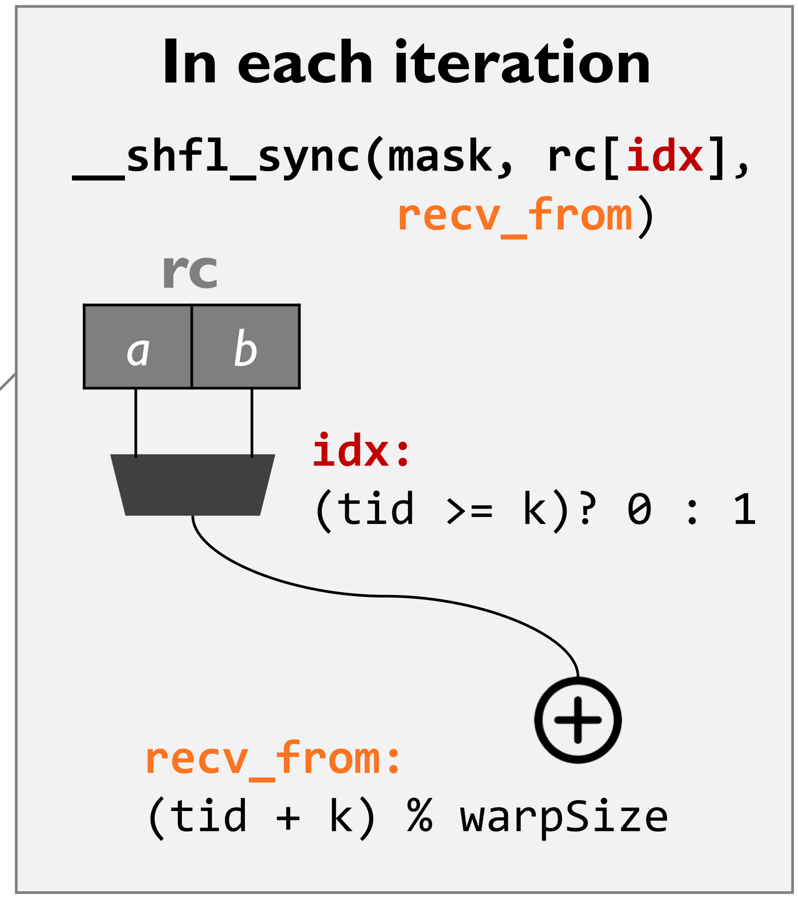
Register Cache: Stencil



[Ben-Sasson et al. ICS' 16]

Register Cache: Stencil





[Ben-Sasson et al. ICS' 16]

Automatic Optimization

These optimizations require:

- reasoning about program globally
- solving multiple constraints together
- rewriting multiple program fragments simultaneously

Cannot be done by a typical rewrite rule in a compiler.

Swizzle Inventor

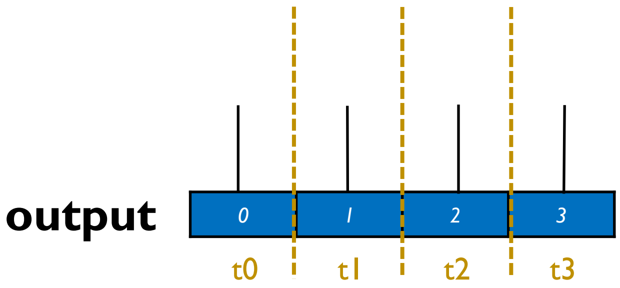
Helps programmers implement swizzle programs by:

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

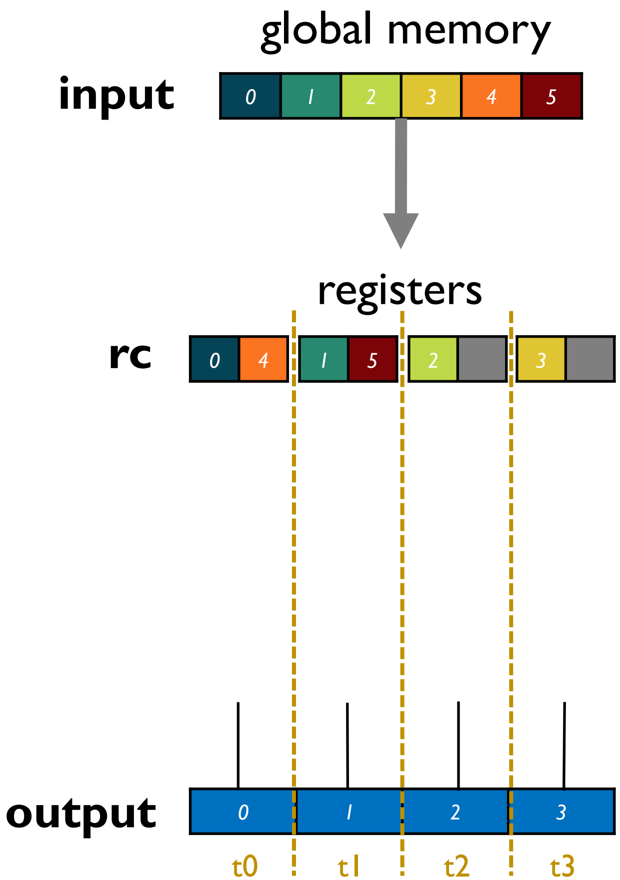
Stencil: Program Sketch

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;
```



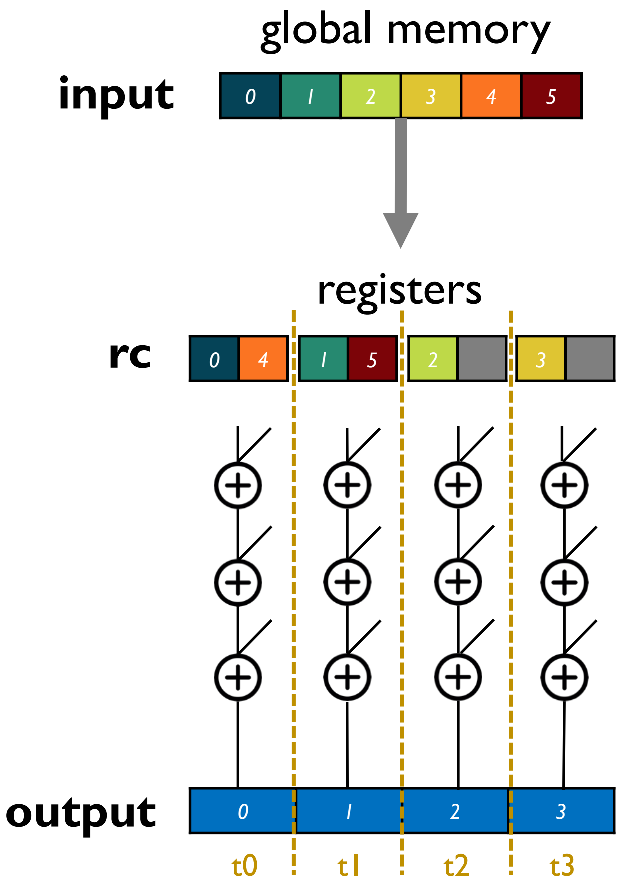
Stencil: Program Sketch



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

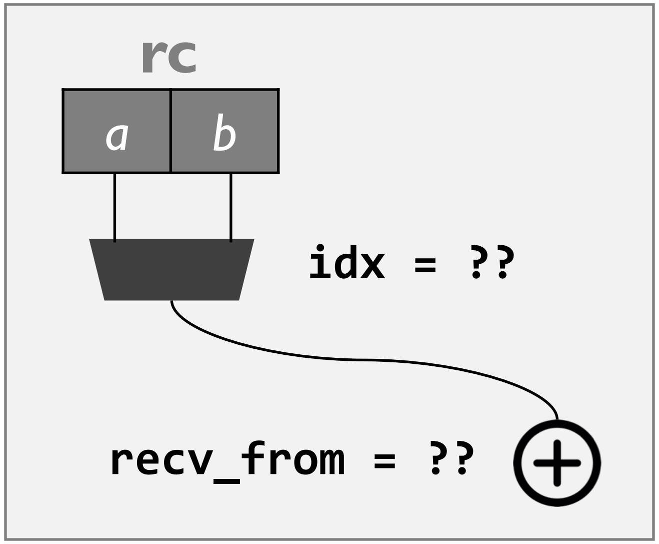


SIMT program

```
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);
```



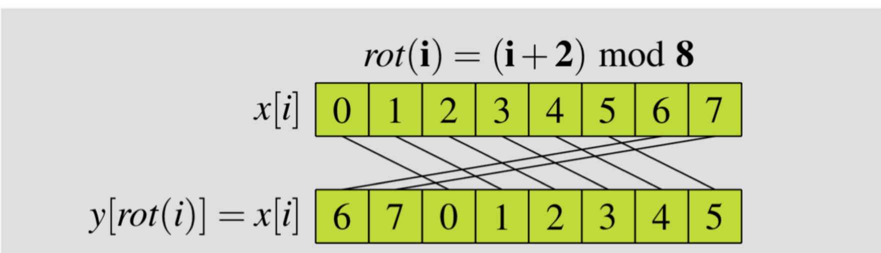
Use `?sw_xform` (transformation swizzle) when `recv_from` is permutation or broadcast of `tid`

Use `?sw_part` (partition swizzle) otherwise

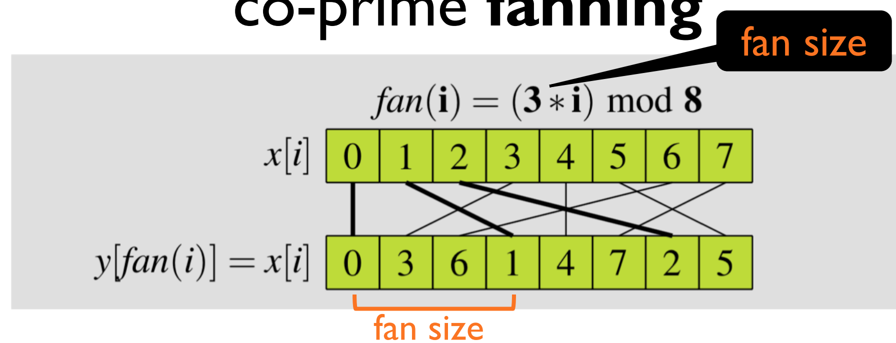
Transformation Swizzle Hole

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

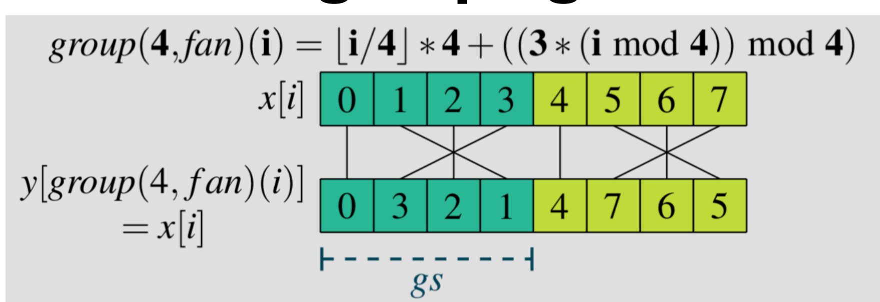
rotation



co-prime **fanning**



grouping



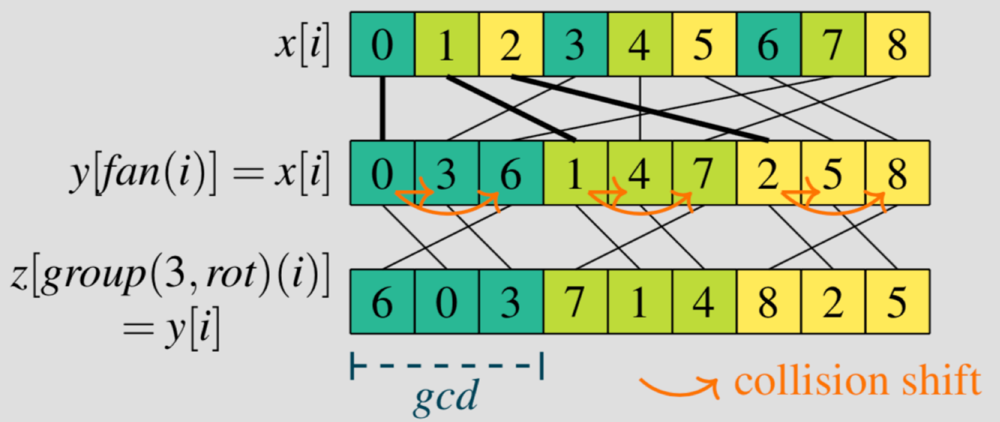
Transformation Swizzle: Example

fanning followed by **grouped rotation**

$$\mathbf{z}[(group(\mathbf{3}, rot) \circ fan)(\mathbf{i})] = \mathbf{x}[\mathbf{i}]$$

$$fan(\mathbf{i}) = (\mathbf{3} * \mathbf{i} + \lfloor \mathbf{i}/\mathbf{3} \rfloor) \bmod \mathbf{9}$$

$$group(\mathbf{3}, rot)(\mathbf{i}) = \lfloor \mathbf{i}/\mathbf{3} \rfloor * \mathbf{3} + (((\mathbf{i} \bmod \mathbf{3}) + \mathbf{1}) \bmod \mathbf{3})$$



Partition Swizzle Hole

```
?sw_part(n, v, …) :=
     if ?sw_cond(v, …)     then 0
     elif ?sw_cond(v, …)   then 1
     ...
     else n - 1
```

Condition Swizzle Hole

$$
\begin{aligned}
\textbf{?sw\_cond}(\mathtt{v}, \ldots) &:= (\mathtt{v} \mid \ldots) \odot_{\mathtt{cmp}} (\mathtt{I} \odot_{\mathtt{bin}} (\mathtt{v} \mid \ldots)) \\
\odot_{\mathtt{cmp}} &:= \; = \mid \neq \mid \geq \mid > \mid \leq \mid < \\
\odot_{\mathtt{bin}} &:= + \mid - \\
\mathtt{I} &:= \mathtt{integer}
\end{aligned}
$$

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;
}
```

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;
  }
}
```

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

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;
}
```

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;
  }
}
```

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;
}
```

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

array of symbolic variables

Symbolic Representation

x

x_0	x_1	x_2	x_3	x_4	x_5

Spec:

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

$x_0+x_1+x_2$	$x_1+x_2+x_3$	$x_2+x_3+x_4$	$x_3+x_4+x_5$

Sketch:

```
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;
```

$x_?+x_?+x_?$	$x_?+x_?+x_?$	$x_?+x_?+x_?$	$x_?+x_?+x_?$

Symbolic Representation

x

x_0	x_1	x_2	x_3	x_4	x_5

Spec:

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

$x_0+x_1+x_2$	$x_1+x_2+x_3$	$x_2+x_3+x_4$	$x_3+x_4+x_5$

Sketch:

```
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;
```

$x_?+x_?+x_?$	$x_?+x_?+x_?$	$x_?+x_?+x_?$	$x_?+x_?+x_?$

Symbolic Representation

x

x_0	x_1	x_2	x_3	x_4	x_5

Spec:

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

$\{x_0, x_1, x_2\}$ op: +	$\{x_1, x_2, x_3\}$ op: +	$\{x_2, x_3, x_4\}$ op: +	$\{x_3, x_4, x_5\}$ op: +

Sketch:

```
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;
```

$\{x_?, x_?, x_?\}$ op: +	$\{x_?, x_?, x_?\}$ op: +	$\{x_?, x_?, x_?\}$ op: +	$\{x_?, x_?, x_?\}$ op: +

Symbolic Representation

x

x_0	x_1	x_2	x_3	x_4	x_5

Spec:

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

$\{x_0, x_1, x_2\}$ op: +	$\{x_1, x_2, x_3\}$ op: +	$\{x_2, x_3, x_4\}$ op: +	$\{x_3, x_4, x_5\}$ op: +

Sketch:

```
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;
```

$\{x_?, x_?, x_?\}$ op: +	$\{x_?, x_?, x_?\}$ op: +	$\{x_?, x_?, x_?\}$ op: +	$\{x_?, x_?, x_?\}$ op: +

≡ *multiset*

Accumulator

x

x_0	x_1	x_2	x_3	x_4	x_5

Spec:

```
for(int i = 0; i < n; i++) {
  o = create_accumulator(0, +);
  for(int k = 0; k < 3; k++)
    accumulate(o, [x[i+k]], true);
  y[i] = eval(o);
}
```

$\{x_0,x_1,x_2\}$ op: +	$\{x_1,x_2,x_3\}$ op: +	$\{x_2,x_3,x_4\}$ op: +	$\{x_3,x_4,x_5\}$ op: +

Sketch:

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

o = create_accumulator(0, +);
for(int k = 0; k < 3; k++) {
  int tmp = magic_get(rc);
  accumulate(o, [tmp], true);
}

y[tid] = eval(o);
```

$\{x_?,x_?,x_?\}$ op: +	$\{x_?,x_?,x_?\}$ op: +	$\{x_?,x_?,x_?\}$ op: +	$\{x_?,x_?,x_?\}$ op: +

$\equiv$ ***multiset***

Accumulator

Sum stencil: $\bigoplus \rightarrow +$

$$\{x, y, x\} \equiv_{multiset} \{x, x, y\}$$

$$x + y + x = x + x + y$$

Convolution: $\bigoplus \rightarrow +$, $\bigodot \rightarrow \times$

$$\{\{w, x\}, \{u, y\}\} \equiv_{multiset} \{\{u, y\}, \{x, w\}\}$$

$$(w \times x) + (u \times y) = (u \times y) + (x \times w)$$

$\bigoplus$ and $\bigodot$ must be
associative and **communitive**.

Search Problem

Spec: sequential program

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

  for(int i = 0; i < n; i++) {
    o = create_accumulator(0,identiy,+);
    for(int k = 0; k < 3; k++)
      accumulate(o, [x[i+k]], true);
    y[i] = eval(o);
  }
}
```

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

Sketch: CUDA sketch

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

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

  o = create_accumulator(0,identiy,+);
  for(int k = 0; k < 3; k++) {
    int tmp = magic_get(rc);
    accumulate(o, [tmp], true);
  }

  y[tid] = eval(o);
}
```

Expressiveness:

Can Swizzle Inventor synthesize GPU kernels with swizzling optimizations in the literature?

Stencil computations

Finite field multiplication

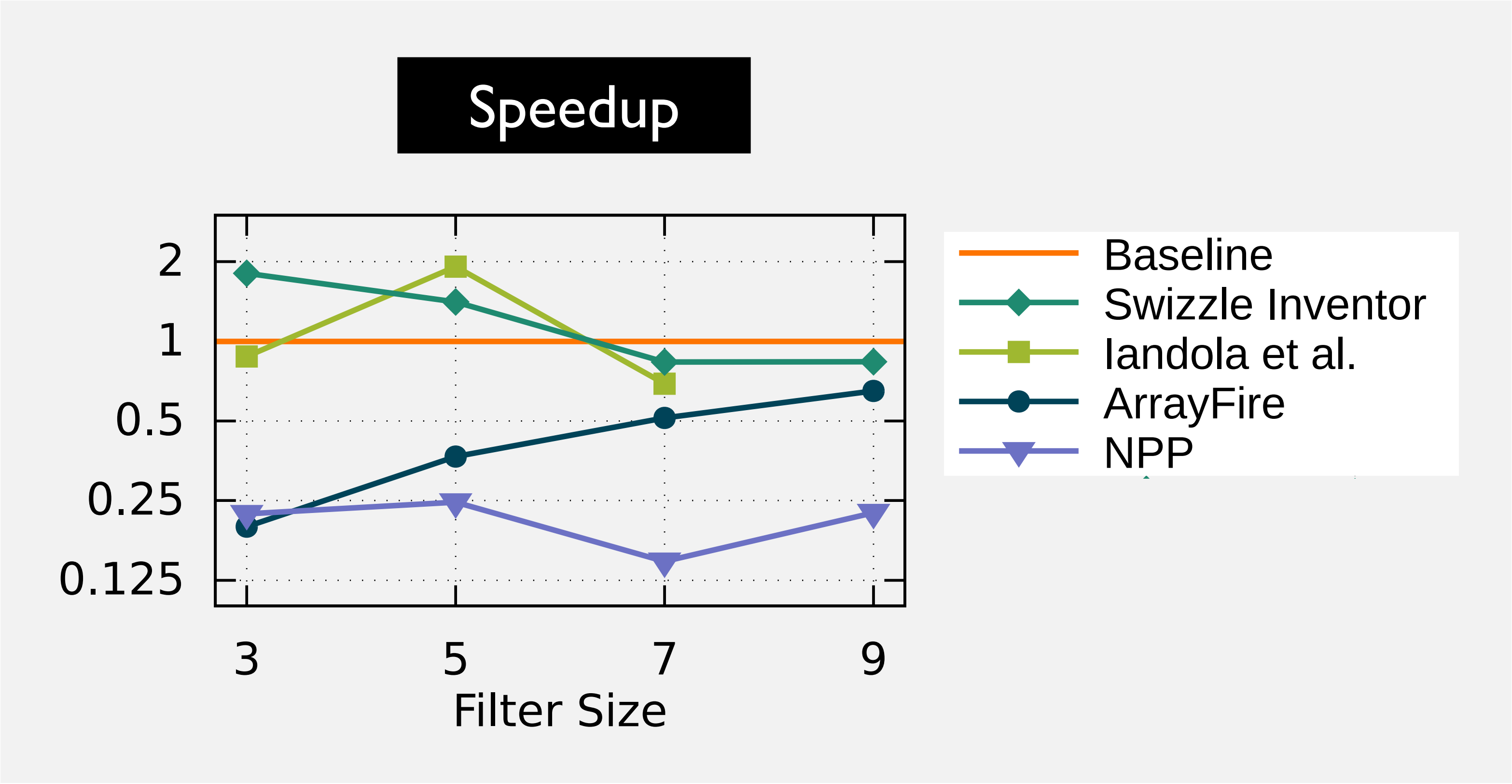
Matrix transposition

Inventiveness:

Can Swizzle Inventor invent new optimizations?

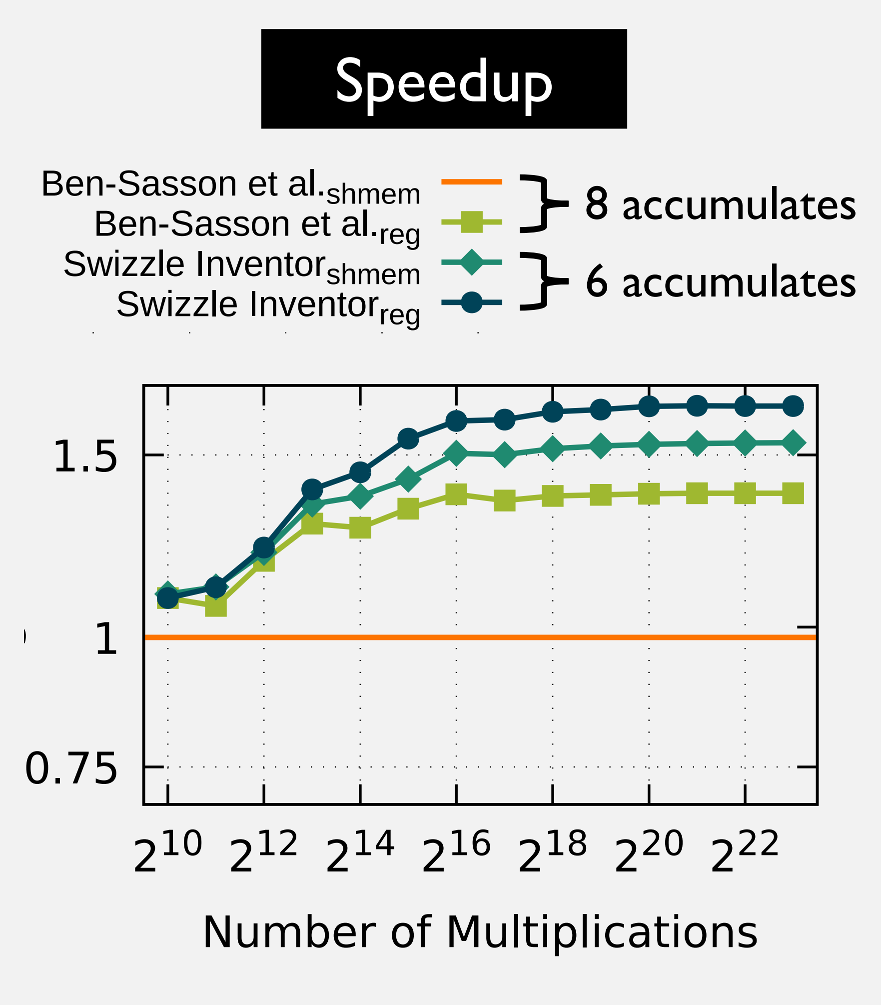
Stencil: 2D Convolution

Use registers to cache input image.

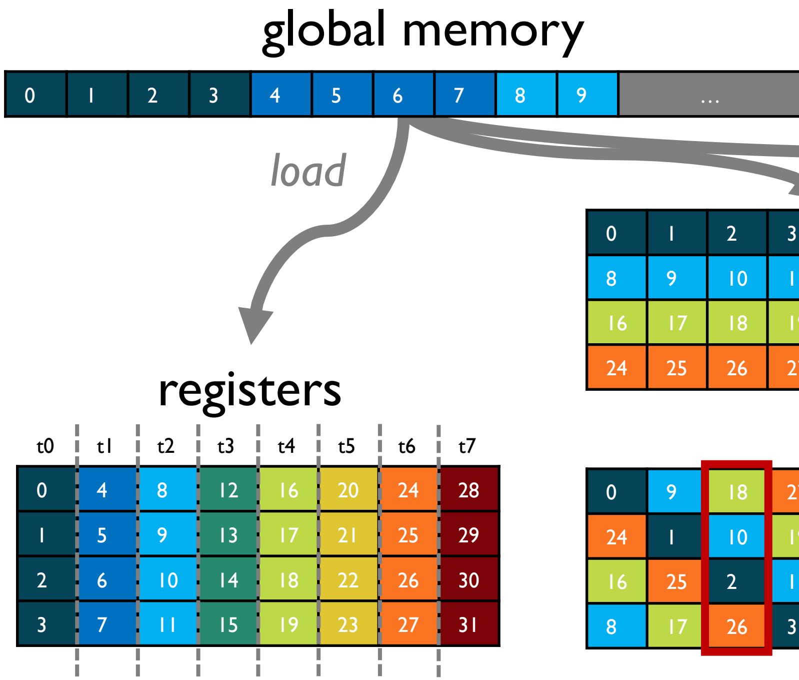


Finite Field Multiplication

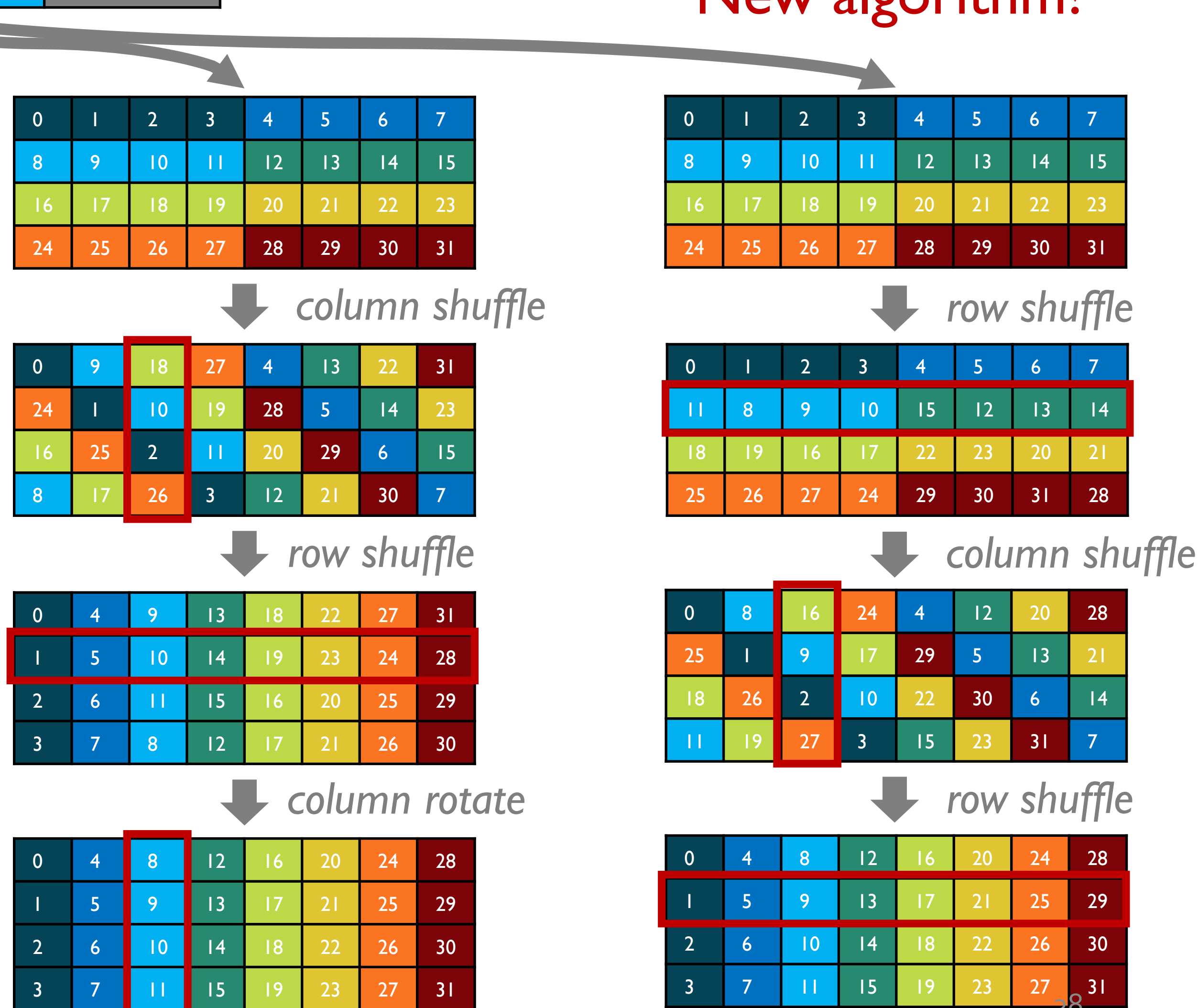
```
// Create ans0, ans1, ans2, ans3
acc ans0 = create_accumulator(0, identity, ^, &);
...

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
  ...
}
```



Matrix Transposition



New algorithm!



Swizzle Inventor synthesizes in seconds!

Search space = ~10^{23}

Swizzle Inventor

Helps programmers implement swizzle programs by:

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

github.com/mangpo/swizzle-inventor