

# Swizzle Inventor

**Phitchaya Mangpo Phothilimthana**

Archibald Samuel Elliott

An Wang

Abhinav Jangda

Bastian Hagedorn

Henrik Barthels

Samuel J. Kaufman

Vinod Grover

Emina Torlak

Rastislav Bodik

UC Berkeley (now at Google Brain)

University of Washington

University of Washington

University of Massachusetts Amherst

University of Münster

AICES, RWTH Aachen University

University of Washington

NVIDIA

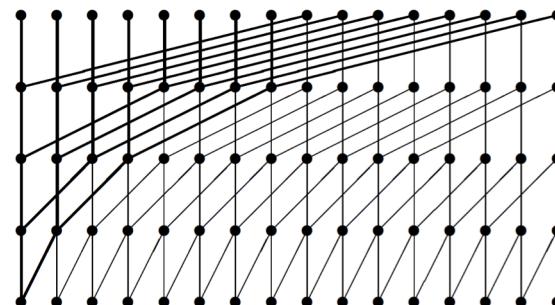
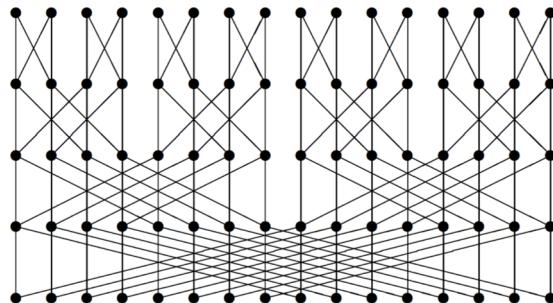
University of Washington

University of Washington

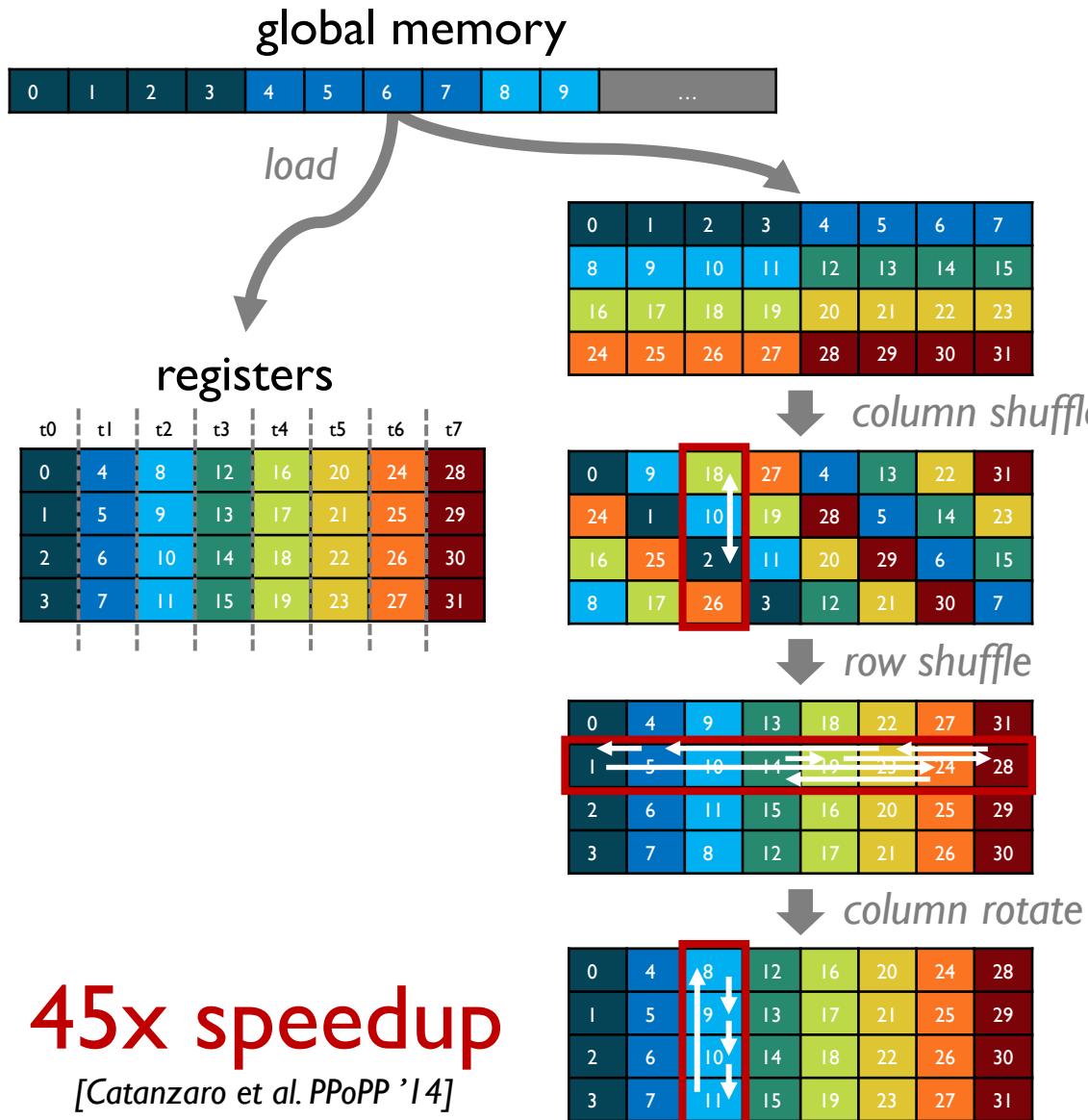
# Swizzle

**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



when data structures are dictated due to interface constraints or algorithmic requirements. Alternatively, programmers act

Define the standard row-major linearization, where  $l_{row}(i, j)$  creates a linear index from a row and column index, and

**Theorem 1.** The C2R transpose implements transposition for row-major arrays, and the R2C transpose implements

**Proof.** Defining versions of  $l_{row}$ ,  $l_{row}$ , and  $j_{row}$  where  $m$  and  $n$  have been swapped:

In fact,  $d_i(j)$  is periodic, which means there are guaranteed to be conflicts in the permutation. However, the periodicity is how to remove these conflicts.

**Lemma 2.**  $\forall x, y \in \mathbb{N} \mid 0 \leq x < b, 0 \leq y < b, x \bmod n = y \bmod n \text{ implies } x = y$ .

**Proof.** Proof by contradiction. Assume  $3x, 3y \mid 0 \leq x < b, 0 \leq y < b, x \neq y$  and also that  $x \bmod n = y \bmod n$ . Substituting, we get  $x = y \bmod n$  and by cancellation properties of congruence,  $x = y$ . This contradicts  $x \neq y$  and proves  $x \bmod n = y \bmod n \Rightarrow x = y$ . Since  $gcd(a, bc) = c$ , then  $nm \equiv 0 \pmod{c}$  if and only if  $b \equiv 0 \pmod{c}$ . Since  $a$  and  $b$  are coprime, the modular multiplicative inverse of  $a$  and  $b$  exists. Therefore,  $c \bmod n \equiv 0 \pmod{n} \Rightarrow a^{-1}b \equiv 0 \pmod{n}$ . We assumed  $a \leq c \leq b$  and  $0 \leq y < b$ , the modulus is extended, and we assumed  $a \leq c \leq b$  and  $0 \leq y < b$ , the modulus is extended, and we assumed  $a \leq c \leq b$ . But this is a contradiction, since we assumed earlier that  $x \neq y$ .  $\square$

**Lemma 3.** Let  $S = \bigcup_{k=0}^{b-1} \{km \bmod n\}$ , and let  $T = \bigcup_{k=0}^{b-1} \{kc\}$ . Then  $S = T$ .

**Proof.** By Lemma 2, we know  $|S| = b$ . We also know  $|T| = b$  by inspection. Next, we show that  $S \subseteq T$ . To do this, we show that  $\forall k \in [0, b), \forall i \in [0, b], km \bmod n = kc$ . By the definition of modulus,  $km \bmod n = kc \Leftrightarrow \frac{km}{n} = \lfloor \frac{km}{n} \rfloor + k \pmod{b} \Leftrightarrow km \equiv kc \pmod{b}, \text{ where } k \in \mathbb{Z}$ . To prove this, we note that since  $km \equiv kc \pmod{b}$ , we have  $km - kc \equiv 0 \pmod{b}$ . Accordingly,  $S \subseteq T$ , and since we already showed  $|S| = |T|$ , it must be true that  $S = T$ .  $\square$

**Theorem 4.**  $d_i(j)$  is a bijection on  $j \in [b, l + 1)$  for  $i \in [0, m)$ .

**Proof.** Observing that  $\lfloor \frac{i}{b} \rfloor = l$  is constant for  $j \in [b, l + 1)$ , we first analyze the sets

$$\begin{aligned} S_{i,j} &= \bigcup_{j=0}^{\lfloor \frac{i}{b} \rfloor + 1} \{d_i(j)\} \\ &= \bigcup_{j=0}^b \{((i+l) \bmod m) + jm \bmod n\} \\ &= \bigcup_{j=0}^{b-1} \{((i+l) \bmod m) + (b+h)jm \bmod n\} \\ &= \bigcup_{j=0}^{b-1} \{((i+l) \bmod m) + hm \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{((i+l) \bmod m) + hm \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{(i+l) \bmod m + hm \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{(i+l) \bmod m + hc \bmod n\}, \end{aligned}$$

where we first replace  $d_i(j)$  by its definition, followed by removing the offset  $lb$  from the index, which allows one to

"Rows to Columns" (R2C) and "Columns to Rows" (C2R). These two permutations are inverses of each other. These two permutations are illustrated in Figure 1.

We are not the first to view transposition in this manner, nor are we the first to implement it. In fact, in Leight's [4], where the C2R permutation is called "transposer", and the R2C permutation is called "untranspose", the R2C transpose is called "untranspose".



Figure 1: C2R and R2C transpositions,  $m = 3, n = 3$

We begin by discussing out-of-place versions of these transpositions, and showing how they relate to traditional matrix transposition.

inverses of each other, and scatter and gather transpositions are inverses of each other, we can also define the transposition in terms of scatter operations:

$$A^{C2R}[t(i, j), d_i(j)] = A[i, j] \quad (13)$$

$$A^{R2C}[e(i, j), d_i(j)] = A[i, j] \quad (14)$$

For example, consider the element with value 16 highlighted in Figure 1, where  $m = 3, n = 3$ . On the left, this value is located at  $t(1, 2) = 16$ . After performing the transpose, the element is located at  $t(2, 1) = 17$ . Looking at Equation 14, we can compute the destination indices:  $e' = t(2, 1) = (j + in) \bmod m = (0 + 2 \cdot 8) \bmod 9 = 8$ , and  $d_i(j) = t(2, 1) = 17$ .

Now we see the connection between the R2C and C2R transposes and the linearized transposition problem:

$$\begin{aligned} e(t_{row}(l), j_{row}(l)) &= \left\lfloor \frac{l - t_{row}(l)}{m} \right\rfloor \\ &= \left\lfloor \frac{l}{m} \right\rfloor = l_{row}(l) \end{aligned}$$

And so we can substitute to show

$$A^{C2R}[t(i, j), d_i(j)] = A_{row}(l_{row}(j_{row}(l)), l_{row}(l)) \quad (21)$$

Therefore,  $A^{C2R} = A_{row}^T$ . Symmetric reasoning shows  $A^{R2C} = A_{row}^T$ .

**Theorem 2.** Swapping dimensions  $m$  and  $n$  before performing the transpose, the C2R transpose implements transposition for column-major arrays, and the R2C transpose implements transposition for column-major arrays, and the R2C transpose implements transposition for row-major arrays.

We will restrict our attention to the C2R transposition in this section, as the R2C transposition is merely the inverse of the C2R transposition.

As shown in Equation 10, the destination column of element  $j$  is row  $i$ :

$$d_i(j) = (i + jm) \bmod n \quad (22)$$

where we have fixed  $i$  for presentation purposes. We would like to perform row-wise permutations to send each element to the correct column indexed by the transpose. This can be done by rotating each row by  $k$  elements, where  $k$  is the number of columns minus the number of elements in the row. Otherwise the row-wise operation is not a well-formed permutation, and the transposition is not decomposable.

However, in general,  $d_i(j)$  is not bijective on  $j \in [0, n]$ , meaning each element does not go to a unique column, and so the row-wise operation is not a well-formed permutation.

In this period  $b$ , we adjust the array to move the conflicts. Consider matrix  $A$ , by which we mean: for row  $i$  rotated by  $k$  elements, the  $(i, k)$  mod  $m$ . Consider rotating rows, or equivalently, column  $j$  extracted from the source array using

index equation:

$$r_j(i) = \left( i + \left\lfloor \frac{j}{b} \right\rfloor \right) \bmod m \quad (23)$$

Substituting, after rotating all columns of the array, the resulting destination column for each element of the new array is

$$d'_i(j) = \left( i + \left\lfloor \frac{j}{b} \right\rfloor \right) \bmod m + jm \quad (24)$$

Our task is to prove that Equation 24 is a bijection, which will show that the rotations have removed conflicts, decomposing the transposition.

To do this, the following lemmas are useful.

Figure 2 shows the state of a matrix as it is transposed using a C2R transposition. Each of the three steps corresponds to one of the three column-major loops in Algorithm 1.

The first step shows the upper bound  $c_{n-1}((k+1)b - 1) = (k+1)b - 1$ . Accordingly, over the domain  $0 \leq k < n$ , it follows that  $c_{n-1}((k+1)b - 1) \leq c_{n-1}(k+1)$ . That is, it holds that over this domain,  $c_{n-1}(k) \leq c_{n-1}(k+1) - 1$ .

In other words, the source column for all elements in group  $k$  were rotated by  $k$  elements.

It then establishes a one-to-one correspondence between subarrays of the original columns of the array that were rotated by  $k$  elements. This is a key observation for the C2R transpose.

Having established this correspondence, we need to adjust the source row indices to compensate for the rotation. Additionally, we need to make sure that the function  $c_{n-1}$  contracts this rotation. Accordingly,  $c_{n-1}(k)$  is the correct set of row indices to use for the column operations.

Summarizing, the C2R algorithm is performed in three steps:

- If  $gcd(n, m) > 1$ : Rotate columns by gathering from each column using  $r_j(i)$  from Equation 23 to temporary buffer, then copy back to the original column.

- Row shuffle: exchange each row into a temporary vector using  $d_i(j)$  from Equation 24, then copy the result over the original row.

- Column shuffle: gather from each column into a temporary vector using  $s_i(l)$  from Equation 26, then copy the result over the original column.

Considering the algorithm leads to a straightforward statement of the C2R transposition algorithm, using out-of-place permutations in a temporary buffer of size  $\max(m, n)$ . This is presented as algorithm 1.

**Algorithm 1.** In-place C2R transpose of array  $A$

```

if  $gcd(n, m) > 1$  then
    for  $j = [0, n)$  do
        for  $i = [0, m)$  do
             $tmp[j] = A[i, j]$  [Gather per eq. 23]
    end for
    for  $i = [0, m)$  do
         $A[i, j] = tmp[j]$  [End for]
    end for
end if

for  $i = [0, m)$  do
    for  $j = [0, n)$  do
         $tmp[j] = A[r_j(i), j]$  [Scatter per eq. 24]
    end for
    for  $j = [0, n)$  do
         $A[i, j] = tmp[j]$  [End for]
    end for
end for

for  $j = [0, n)$  do
    for  $i = [0, m)$  do
         $tmp[i] = A[d_i(j), i]$  [Gather per eq. 26]
    end for
    for  $i = [0, m)$  do
         $A[i, j] = tmp[i]$  [End for]
    end for
end for

```

**Proof.** In the worst case, the algorithm reads and writes each element 6 times, performing row and column permutations out-of-place. This gives the work complexity of  $O(mn)$ , which is known to be optimal. The algorithm requires a temporary vector of size  $\min(m, n)$  in order to carry out these out-of-place permutations.  $\square$

#### 4. Optimization

The C2R transpose is shown in algorithm 1, but it is C2R inverse and does not in terms of both scatter and gather permutations on both the rows and columns of the array. Practical considerations of these algorithms may motivate the use of alternative implementations. For example, gather based formulations are more memory efficient than scatter based due to functional restrictions. Additionally, we have found it useful to restrict the column operations: rather than allowing unrestricted column shuffles, we perform the column operations using a column-major loop, as shown in algorithm 2. Restricting the column operations allows us to optimize memory access patterns, and enables the in-register implementation using SIMD instructions.

We have found it useful to choose either row-major or column-major linearization during C2R and R2C transposes, which is an important optimization.

**Theorem 5.** The linearization assumed while performing C2R and R2C transposes does not affect the permutation they induce

**Proof.** Let  $B$  represent a row-major array that is created by a C2R transposition using column-major indexing on a row-major array  $A$ .

$$B[i, j] = A_{row}(l_{row}(s_i(l_{row}(i), j_{row}(i))), l_{row}(i)) \quad (28)$$

Noting that

$$l_{row}(i) = x \pmod{m}, \quad l_{row}(i) = x \quad (29)$$

and substituting the C2R source equations from Equations 7 and 8 as well as from Equations 16 and 17 into the indexing function above,

$$\begin{aligned} l_{row}(s_i(l_{row}(i), j_{row}(i)), l_{row}(i), j_{row}(i)) &= \\ l_{row}(l_{row}(i), j_{row}(i), l_{row}(i), j_{row}(i)) &= l_{row}(i, l_{row}(i), j_{row}(i), l_{row}(i)) \end{aligned}$$

This proves

$$B[i, j] = A_{row}^{C2R}[i, j] \quad (30)$$

Similar reasoning holds for row-major indexing on a column-major array.

Thereafter, we give the freedom in index arrays to mix major and minor order, regardless of their native storage order. Although the intermediate step during the transposition differs depending on the choice of linearization used to perform C2R and R2C transposes, the fact that the final step is a gather based permutation is the same for both implementations. This is an important performance optimization, since we can implement the so-called  $m \times n$  and that row and column operations always run in fixed directions, regardless of whether the array was given as  $m \times n$  or  $n \times m$  storage order. This enables us to optimize memory access patterns by fitting cache lines.

And the now permutation is:

$$q(i) = \left( i - \left\lfloor \frac{i}{b} \right\rfloor \right) \bmod m \quad (31)$$

This decomposition of a column shuffle into these two more restricted primitives is correct because  $(p \circ q)(i) = q'(p(i))$ .

#### 4.3 Rows to Columns Optimizations

The row shuffle step in the R2C transpose is simple when formulated as a gather, since it can just use  $d'_i(j)$  directly without any computation.

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

$$q^{-1}(i) = \left( \frac{i - s'_i(i)}{a} \right) \bmod n \quad (34)$$

Instead of performing a scatter rotation to invert the rotation in the C2R algorithm, we can do a gather rotation with inverted indices:

$$p_i^{-1}(i) = (i - j) \bmod m \quad (35)$$

And the final rotation indices are also inverted from the C2R pre-rotation indices:

$$r'_j(i) = \left( i - \left\lfloor \frac{i}{b} \right\rfloor \right) \bmod n \quad (36)$$

#### 4.4 Arithmetic Strength Reduction

Evaluating the index equations, such as Equation 31, involves repeated calculations of integer division and integer modulus. We found a significant performance improvement by using a strength reduction technique that involves computing a reciprocal and a reciprocal, and then performing division into a multiplication by the reciprocal followed by a

canceling the resulting additive term

$$(bm \bmod n) = (bn \bmod bc)$$

We then distribute the expression  $((i + l) \bmod m) \bmod n$  by  $(i + l) \bmod m$  to define  $k_1 = \lfloor \frac{i}{b} \rfloor$  and  $k_0 = \lfloor \frac{i}{b} \rfloor + l$ , and  $i = k_0 + l(b - k_1)$ . Then

$$(i + l) \bmod m = n - i \bmod n$$

and  $bm \bmod n = bn \bmod bc$ . Now, we see that for  $i \in [0, m)$ ,  $b - i \bmod n = b - i \bmod bc$ . Therefore,  $i \bmod n = (b - i) \bmod bc$ .

Now, for any fixed  $i \in [0, m)$ , the range of  $d_i(j)$  over the entire domain  $[0, n)$  is

$$\begin{aligned} \bigcup_{j=0}^{b-1} \{d_i(j)\} &= \bigcup_{j=0}^{b-1} S_{i,j} \\ &= \bigcup_{j=0}^{b-1} \{((i+l) \bmod m) + jm \bmod n\} \\ &= \bigcup_{j=0}^{b-1} \{((i+l) \bmod m) + (b+h)jm \bmod n\} \\ &= \bigcup_{j=0}^{b-1} \{((i+l) \bmod m) + hm \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{((i+l) \bmod m) + hm \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{((i+l) \bmod m) + hc \bmod n\} \end{aligned}$$

because  $((i+l) \bmod m)$  enumerates all values in  $[0, n]$  on the domain  $i \in [0, c]$ . Therefore,  $d_i(j)$  is a bijection on  $[0, n]$ .

Note that if  $c = \text{gcd}(n, m) = 1$ ,  $\lfloor \frac{i}{b} \rfloor = 0$

This implies that if  $m$  and  $n$  are coprime,  $d_i(j)$  is a bijection.

**Theorem 4.** In-place transposition can be decomposed into independent row-wise and column-wise operations.

**Proof.** Since  $d_i(j)$  is bijective on the domain  $i \in [0, n]$ , then after pre-rotating columns of the array, each element can be sent to a unique destination column during independent row-wise permutations. Once each element has been moved to its correct destination row, it needs to have a new row which it should be sent to complete the transposition. Since the indices in both steps are unique, the row and column wise permutations are decomposable.  $\square$

We have already described the column-wise rotations, and the row-wise rotations are independent row-wise operations. Now, we will give the proof for the row-wise operations necessary to finish the transposition. Since the decomposition ensures each element is directed to the correct column via row-wise

# Load Array of Struct in GPU



load

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

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

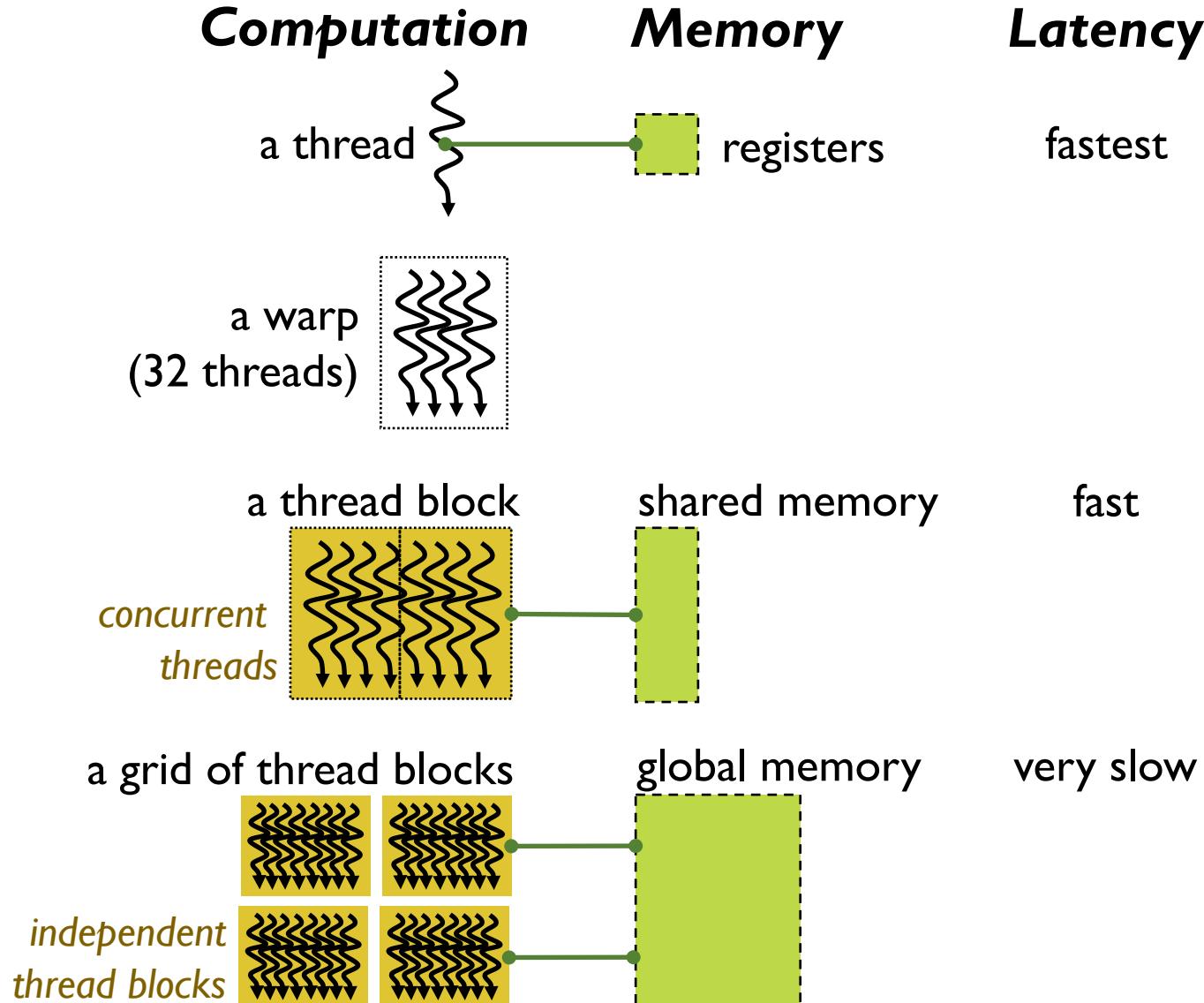
row shuffle

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

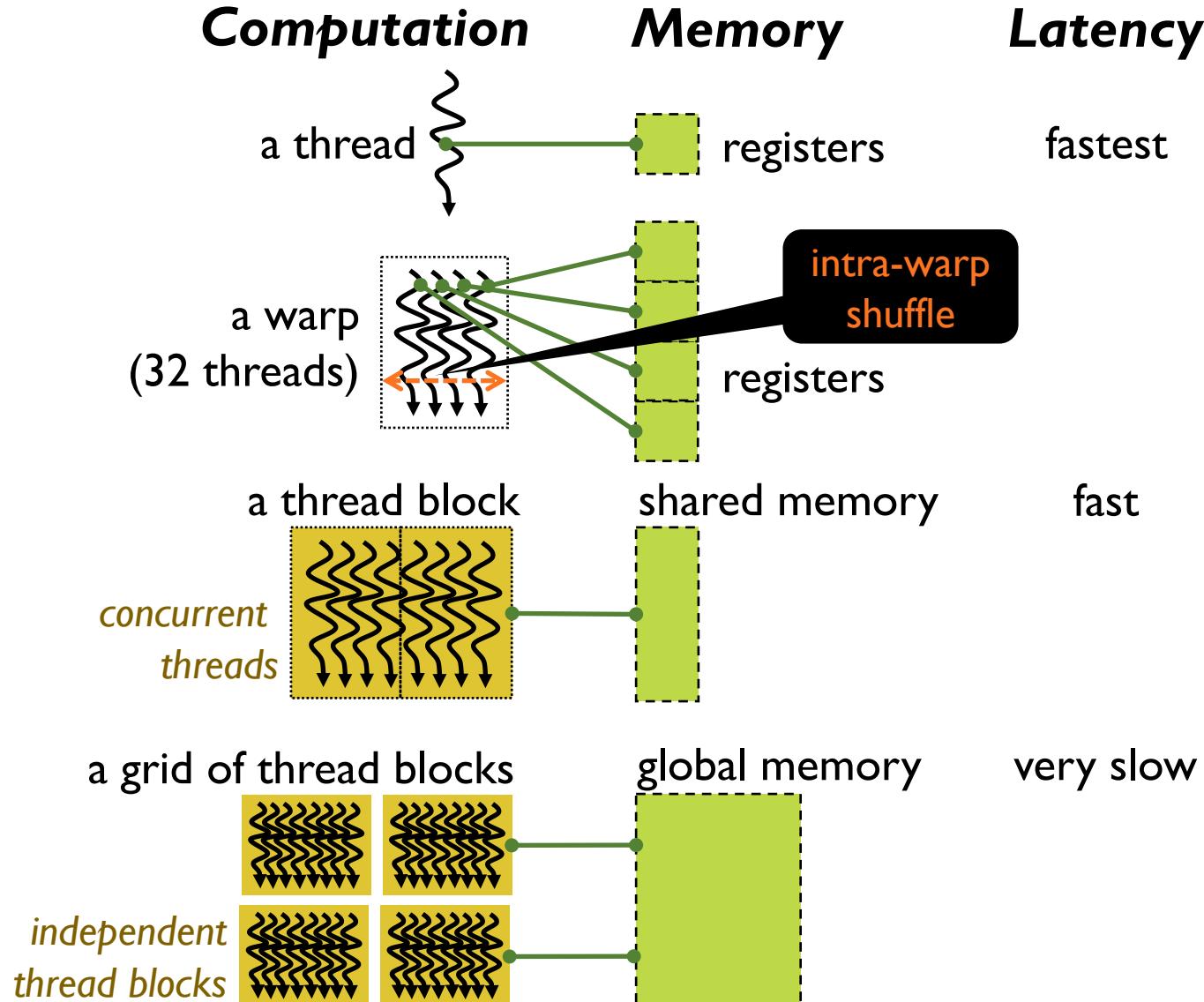
Swizzle Inventor  
synthesizes in  
seconds!

Search space = ~10<sup>23</sup>

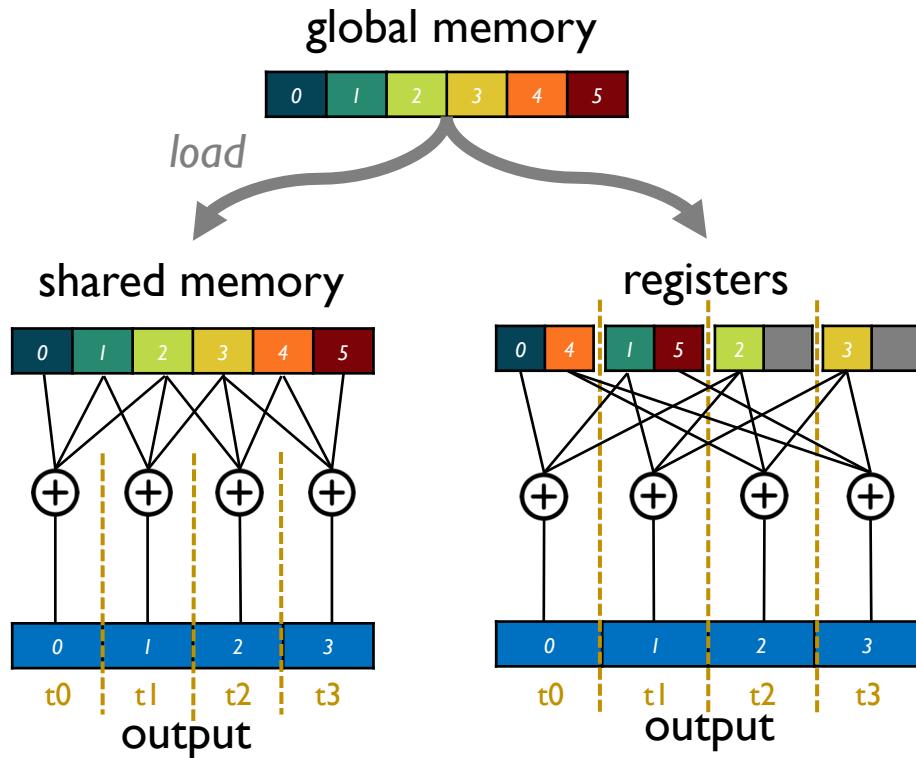
# GPU Architecture Basic



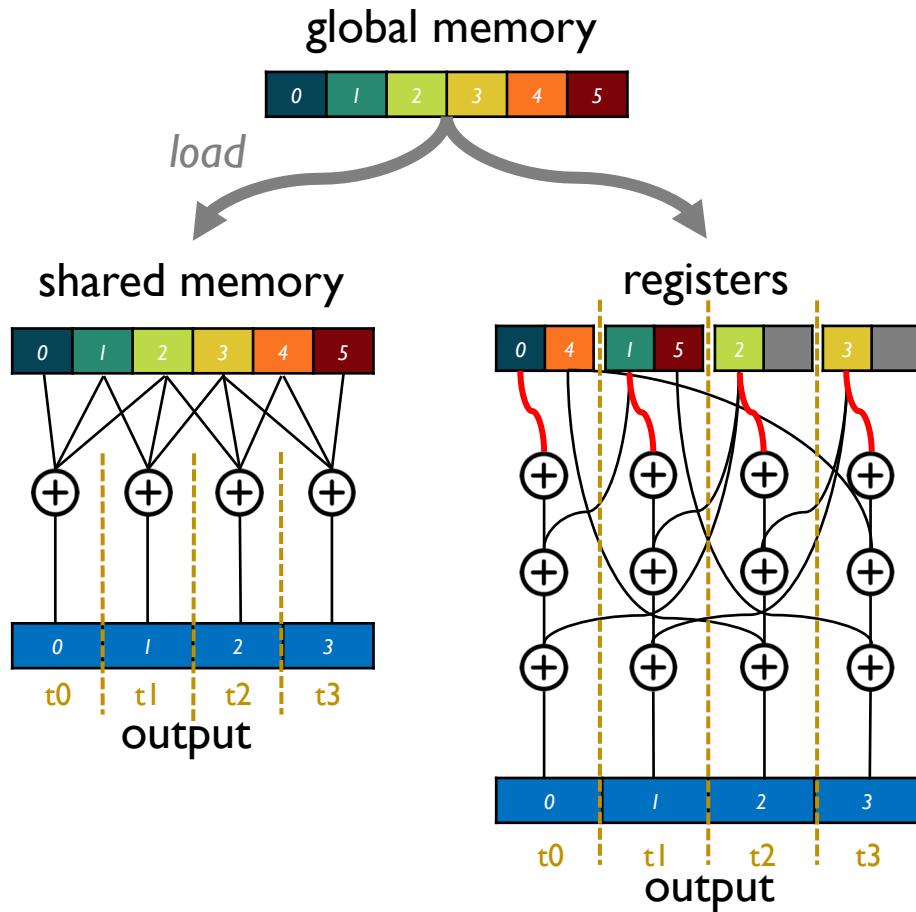
# GPU Architecture Basic



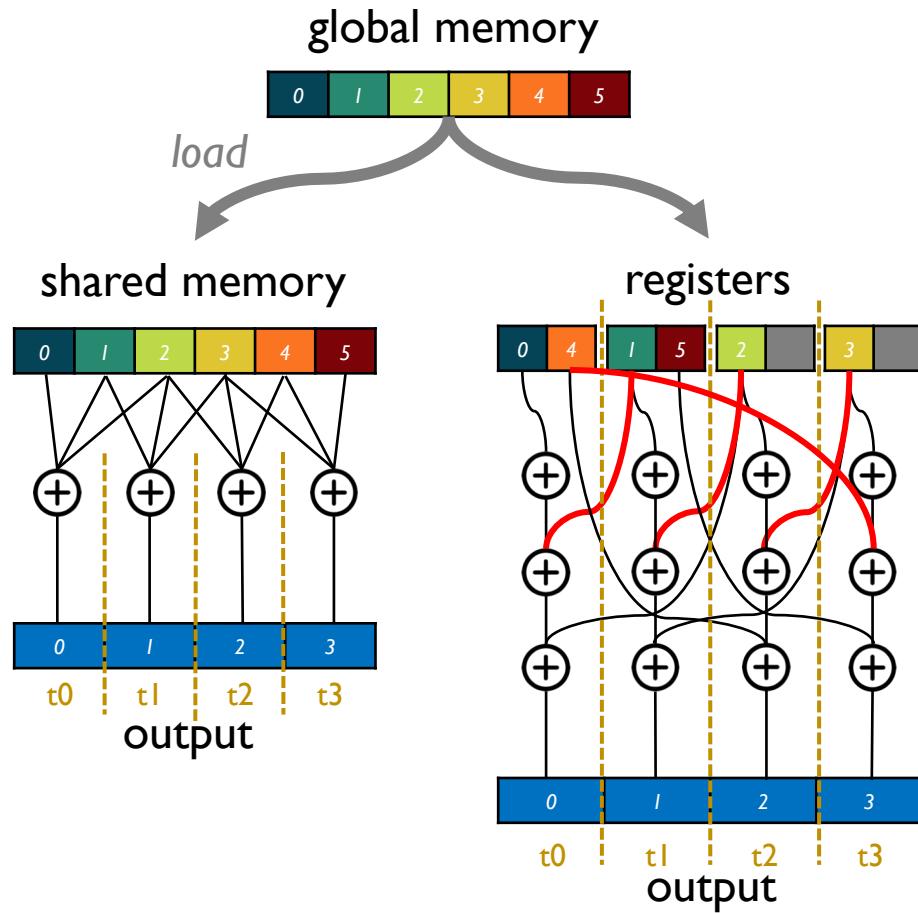
# Register Cache: Stencil



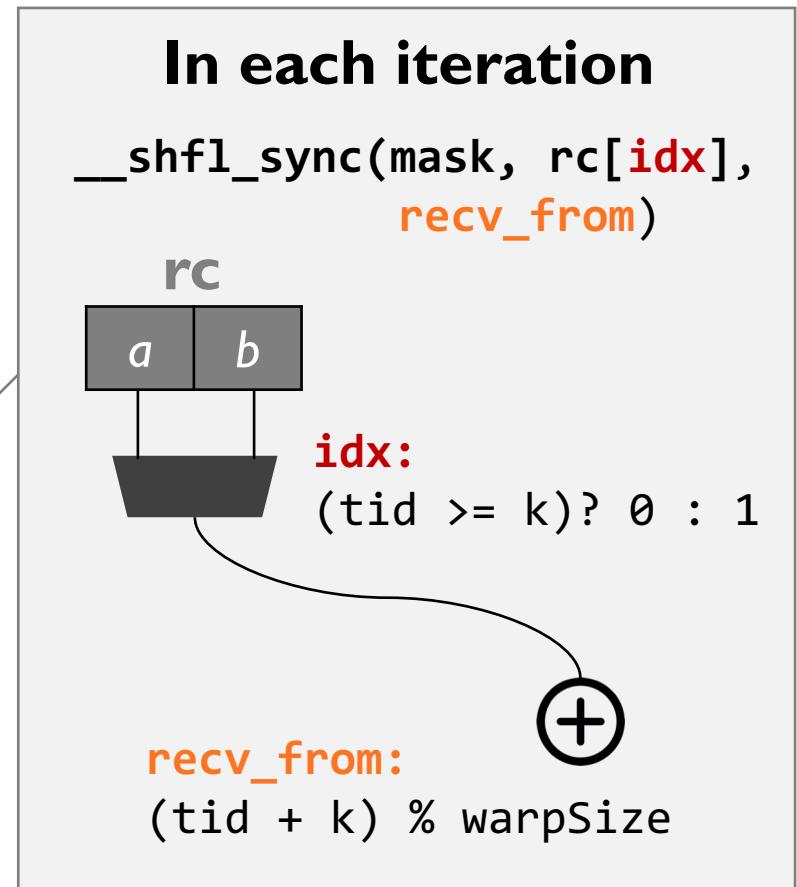
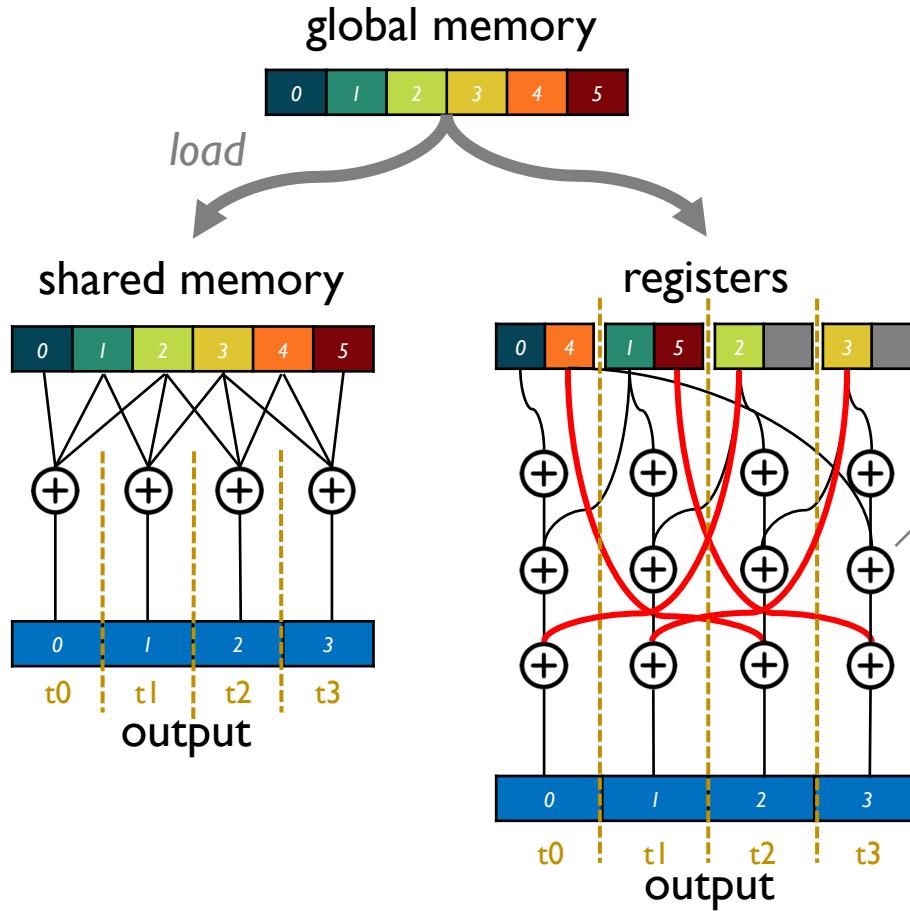
# Register Cache: Stencil



# Register Cache: Stencil



# Register Cache: Stencil



# 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**

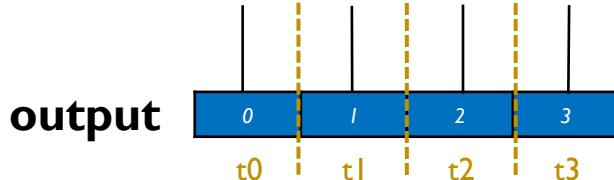
# Stencil: Program Sketch

SIMT program

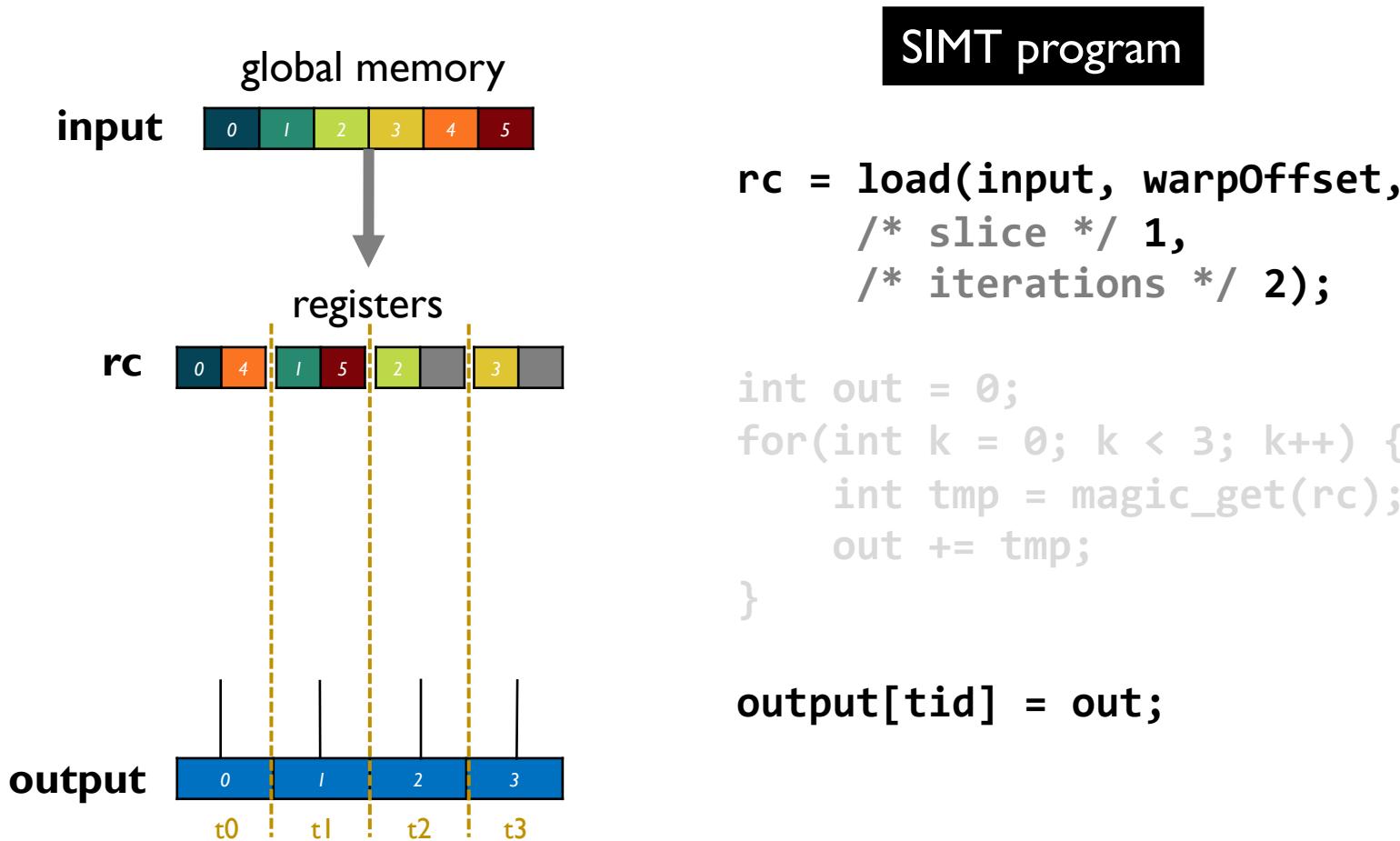
```
rc = load(input, warpOffset,  
/* slice */ 1,  
/* iterations */ 2);
```

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

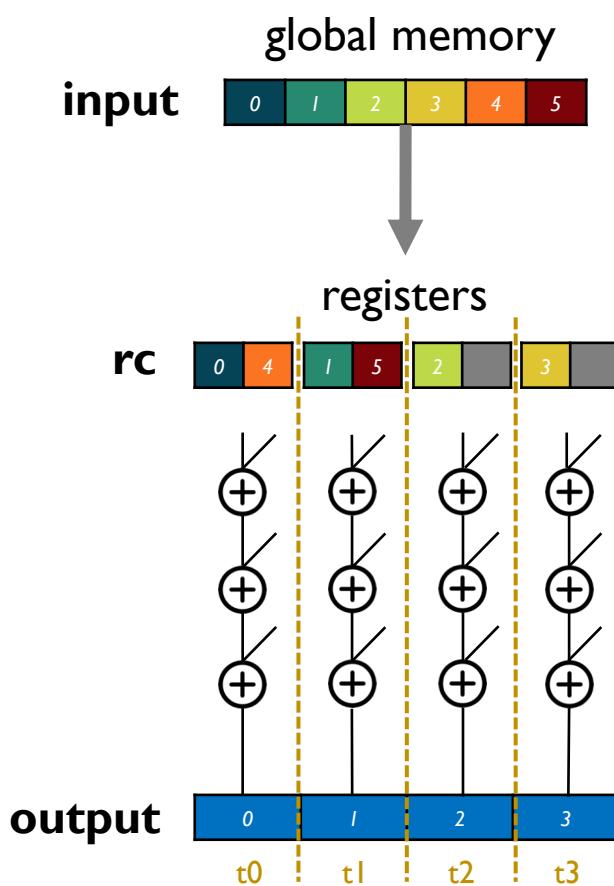
```
output[tid] = out;
```



# Stencil: Program Sketch



# Stencil: Program Sketch

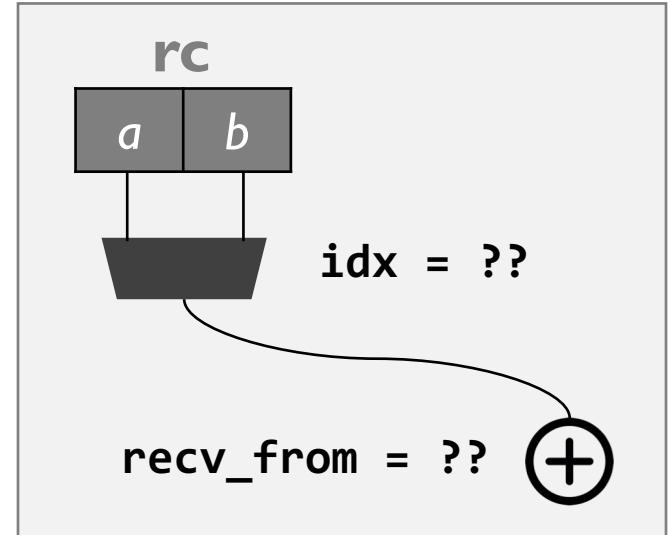


SIMT program

```
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

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

$x[i]$	0	1	2	3	4	5	6	7
--------	---	---	---	---	---	---	---	---

$y[rot(i)] = x[i]$	6	7	0	1	2	3	4	5
--------------------	---	---	---	---	---	---	---	---

## co-prime fanning

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

$x[i]$	0	1	2	3	4	5	6	7
--------	---	---	---	---	---	---	---	---

$y[fan(i)] = x[i]$	0	3	6	1	4	7	2	5
--------------------	---	---	---	---	---	---	---	---

fan size

## grouping

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

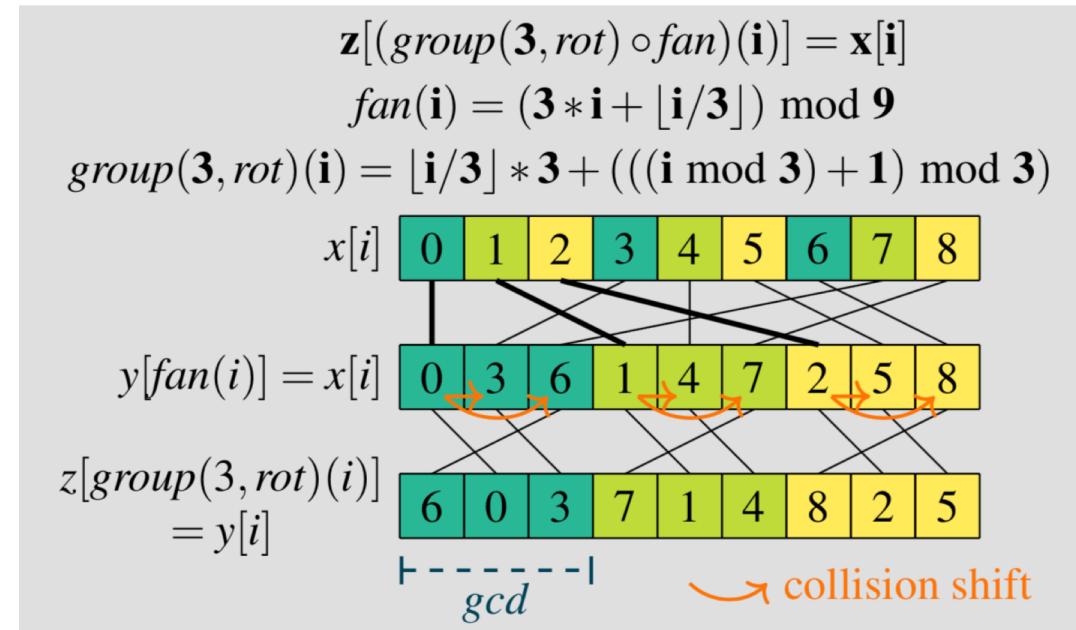
$x[i]$	0	1	2	3	4	5	6	7
--------	---	---	---	---	---	---	---	---

$y[group(4, fan)(i)] = x[i]$	0	3	2	1	4	7	6	5
------------------------------	---	---	---	---	---	---	---	---

gs

# Transformation Swizzle: Example

**fanning followed by grouped rotation**



# 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

```
?sw_cond(v, ...) := (v | ...) ⊕_{cmp} (I ⊕_{bin} (v | ...))  
⊕_{cmp} := = | ≠ | ≥ | > | ≤ | <  
⊕_{bin} := + | -  
I := integer
```

# 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;  
    }  
}
```

$$\begin{aligned} &\exists h \forall x . spec(x, y, n) \\ &\wedge sketch(h)(x, y', n) \\ &\quad \wedge y = y' \end{aligned}$$

## 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;  
    }  
}
```

$$\begin{aligned} &\exists h \forall x . spec(x, y, n) \\ &\wedge sketch(h)(x, y', n) \\ &\quad \wedge y = y' \end{aligned}$$

## 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;  
}
```

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

array of symbolic  
variables

# Symbolic Representation

**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	x <sub>0</sub>	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	x <sub>4</sub>	x <sub>5</sub>

**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 <sub>0</sub> +x <sub>1</sub> +x <sub>2</sub>	x <sub>1</sub> +x <sub>2</sub> +x <sub>3</sub>	x <sub>2</sub> +x <sub>3</sub> +x <sub>4</sub>	x <sub>3</sub> +x <sub>4</sub> +x <sub>5</sub>
--	--	--	--

x <sub>?</sub> +x <sub>?</sub> +x <sub>?</sub>			
--	--	--	--

# Symbolic Representation

**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	x <sub>0</sub>	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	x <sub>4</sub>	x <sub>5</sub>
---	----------------	----------------	----------------	----------------	----------------	----------------

**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;
```

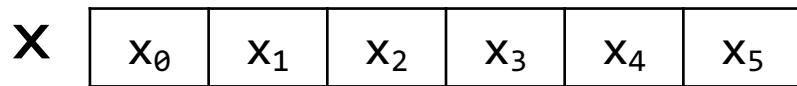
x <sub>0</sub> +x <sub>1</sub> +x <sub>2</sub>	x <sub>1</sub> +x <sub>2</sub> +x <sub>3</sub>	x <sub>2</sub> +x <sub>3</sub> +x <sub>4</sub>	x <sub>3</sub> +x <sub>4</sub> +x <sub>5</sub>
--	--	--	--

x <sub>?</sub> +x <sub>?</sub> +x <sub>?</sub>			
--	--	--	--

# Symbolic Representation

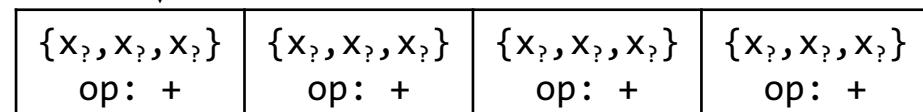
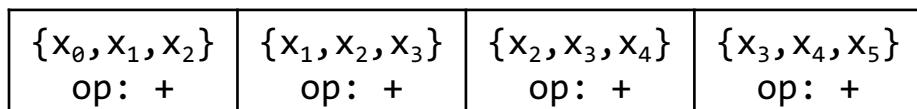
**Spec:**

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



**Sketch:**

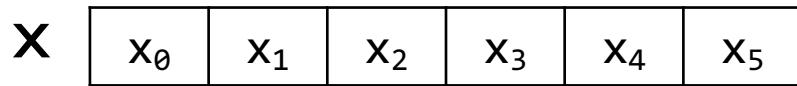
```
rc = load(x, warpOffset, 1, 2);  
  
int out = 0;  
for(int k = 0; k < 3; k++) {  
    int tmp = magic_get(rc);  
    out += tmp;  
}  
  
y[tid] = out;
```



# Symbolic Representation

**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;  
}
```



**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;
```

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

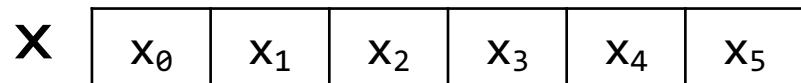
$\equiv$  *multiset*

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

# Accumulator

**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);  
}
```



**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 <sub>0</sub> , x <sub>1</sub> , x <sub>2</sub> }	{x <sub>1</sub> , x <sub>2</sub> , x <sub>3</sub> }	{x <sub>2</sub> , x <sub>3</sub> , x <sub>4</sub> }	{x <sub>3</sub> , x <sub>4</sub> , x <sub>5</sub> }
op: +	op: +	op: +	op: +

$\equiv$  *multiset*

{x <sub>?</sub> , x <sub>?</sub> , x <sub>?</sub> }	{x <sub>?</sub> , x <sub>?</sub> , x <sub>?</sub> }	{x <sub>?</sub> , x <sub>?</sub> , x <sub>?</sub> }	{x <sub>?</sub> , x <sub>?</sub> , x <sub>?</sub> }
op: +	op: +	op: +	op: +

# Accumulator

**Sum stencil:**  $\oplus \rightarrow +$

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

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

**Convolution:**  $\oplus \rightarrow +, \odot \rightarrow \times$

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

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

$\oplus$  and  $\odot$  must be  
**associative** and **commutative**.

# 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);
    }
}
```

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

## 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*

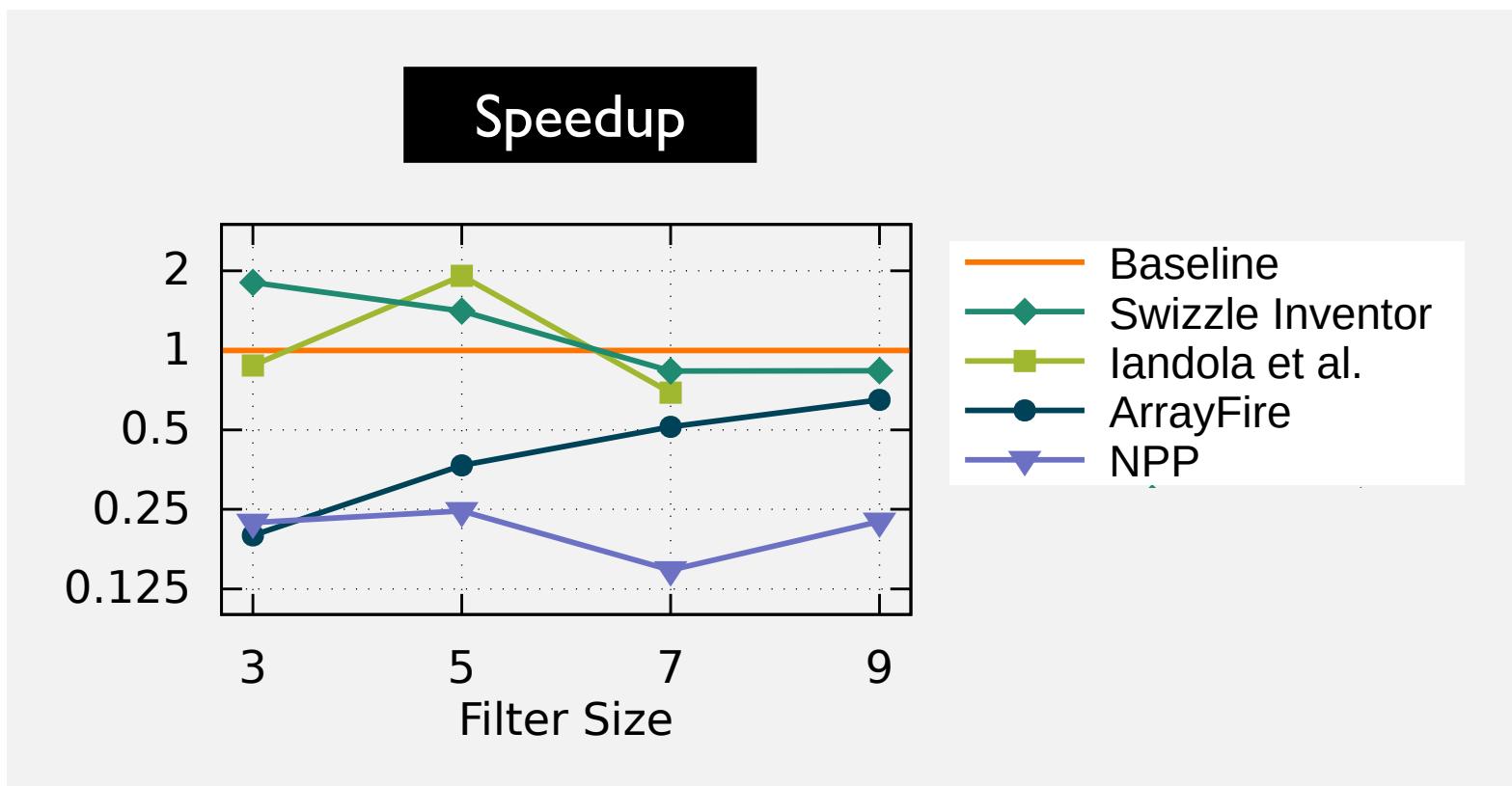
*Finite field multiplication*

*Matrix transposition*

**Inventiveness:**  
Can Swizzle Inventor invent new optimizations?

# Stencil: 2D Convolution

Use registers to cache input image.

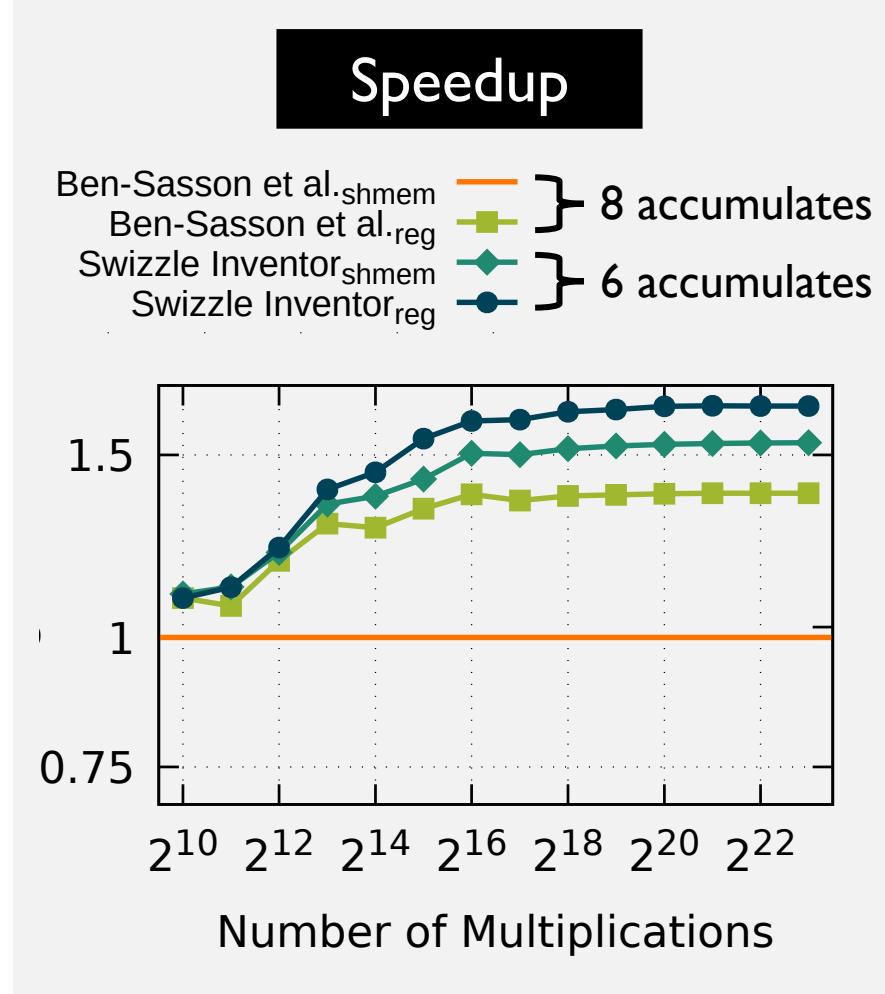


# Finite Field Multiplication

```
// 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



load

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

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 =  $\sim 10^{23}$

# 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](https://github.com/mangpo/swizzle-inventor)