Swizzleflow: Synthesis of Irregular Data Mappings in Accelerator Kernels Using Novel Pruning Abstractions

Krzysztof Drewniak

November 20, 2020

Abstract

Hardware accelerators, such as GPUs and vector coprocessors (for example, Qualcomm's HVX), are key tools for obtaining high performance for mathematical operations that underpin the fast execution of tasks in fields from machine learning to scientific modeling. One reason creating programs that take full advantage of these accelerators is difficult is that they often require complex data access and movement patterns, or swizzles, that existing state-of-the-art compilers cannot discover and whose manual creation by experts is labor-intensive. Therefore, we have developed Swizzleflow, a system that provides a portable model of these swizzling kernels across different accelerators and allows programmers to synthesize such kernels from a sketch. To make this synthesis process suitable for a wider range of problems, we have developed viability tests based on dataflow analysis that allow us to prune away, on average, 99.6% of the search space across benchmarks from multiple domains and platforms. This improved search algorithm, which was used to synthesize kernels for both GPUs and the HVX processor, has also allowed us to obtain significant speedups in synthesis time over our previous work, Swizzle Inventor, especially when using a less restrictive search space.

1 Introduction

Hardware accelerators, such as GPUs and vector coprocessors (for example, Qualcomm's HVX), are key tools for obtaining high performance for mathematical operations that underpin the fast execution of tasks in fields from machine learning to scientific modeling [1]. These accelerators offer substantially higher performance on operations such as matrix multiplication and convolution compared to traditional CPUs.

These speedups do not come for free, however, since accelerators impose constraints on programs so they can execute certain classes of computations more effectively when programmed correctly. For example, on GPUs, a group of threads, or warp, obtains significantly higher performance when each thread loads data from the same cache block. Meeting this constraint causes the algorithm for matrix transpose shown in Listing 1 to be 45x faster than a naive implementation that loads an array from memory directly in transposed order (which would violate this coalescing load requirement). Similarly, failure to distribute traffic across the many parallel memory banks seen on many accelerators can cause unnecessary computation stalls.

Meeting these constraints often requires programmers to use *swizzles* — irregular mappings of data to memories or compute elements that are frequently non-affine. These swizzles are frequently written by hand, since neither compilers nor existing accelerator code generation tools such as AutoTVM [6] or Lift [18] explore the full space of swizzle expressions used in practice. Swizzles exist in high-performance code on various platforms. They contribute to the superior performance cuBLAS has over typical framework-generated matrix multiplication code [9], and the developers of the HVX processor published many swizzling programs, such as various image filters, to demonstrate how to use their platform effectively [10]. We have shown examples of these swizzles in the transposition code in Listing 1 (illustrated in Figure 1) and the convolution code in Listing 3 (as shown in Figure 2).

We have developed Swizzleflow to help developers create these swizzling programs. Swizzleflow allows programmers to model the core data movement seen in swizzle-based programs for dense mathematical operations by abstracting away the details of the target hardware into a functional language that focuses on the core of the problem: specifying data movement through mappings between multidimensional arrays that represent time and space dimensions in the algorithm, as shown in Figures 1 and 2.

Using Swizzleflow, we have expressed and synthesized programs that use human-created (or previously

synthesized) irregular mappings which compilers were not able to generate. These include a Gaussian filter written by HVX developers as an example of how to use their platform effectively [10], the transposition code in Listing 1 (which we show our model of in Listing 2) and optimized convolutions and stencils for GPUs (Listing 3, modelled by Listing 4).

Swizzleflow not only enabled us to model these programs, but also to synthesize them. Swizzleflow abstracts away complexities such as imperative assignment and loops, enabling easier analysis. This abstraction let us develop a highly effective enumerative synthesis algorithm that uses novel abstractions to prune almost all partial programs that cannot be completed to create a solution, bypassing, on average, 99.6% of the search space. Our algorithm can generate solutions to many problems in seconds or, at most, under two minutes, which gave us orders of magnitude of speedup compared to Swizzle Inventor, the main previous work on this swizzle synthesis problem [14].

The remainder of this paper is structured as follows:

- In Section 2, we will describe the Swizzleflow language, taking transposition and convolution on GPUs as our main examples. This high-level model allows us to describe swizzling programs from multiple platforms and allows us to perform effective viability testing.
- In Section 3, we will describe our synthesis algorithm and the viability tests underlying it using transposition on GPUs as an illustrative example and discuss how we compute these tests efficiently.
- Section 4 will explain how our viability tests can be extended to handle more complex program sketches that include reductions (such as convolution) or multiple input arguments.
- Finally, Section 5 will present our empirical evaluation, showing how effective our viability tests were at pruning the search space (99.6% of the possible paths were pruned, and 45% of the benchmarks pruned optimally) and the speedups this produced over previous work.

2 The Swizzleflow language

The Swizzleflow language allows us to model swizzling computations at a high level that abstracts away details of the hardware, such as imperative loops or the fact that code is running on multiple threads. This is demonstrated in Listing 2, which is the Swizzleflow model of the GPU transpose code in Listing 1. Unlike the GPU-specific code, the Swizzleflow implementation is written in a functional style that represents various operations, such as loads from memory or inter-thread communication as transformations between arrays whose dimensions correspond to aspects of how the program's execution vary in time and space (such as the value of variables are executed on different threads or during different loop iterations). The execution of this program is visualized in Figure 1.

$\langle program \rangle$ ::= ($\langle statement \rangle$ | $\langle function\text{-}def \rangle$)*

$\langle statement \rangle$ ::= $\langle ident \rangle$ ':' $\langle shape \rangle$ '=' $\langle expr \rangle$

$\langle shape \rangle$ ::= '[' $\langle integer \rangle$+ ']'

$\langle expr \rangle$ ::= 'fold' [$\langle op\text{-}description \rangle$] $\langle ident \rangle$
| $\langle func\text{-}name \rangle$ '(' $\langle ident \rangle$+ ')'
| $\langle literal \rangle$

Figure 3: The grammar for the Swizzleflow language. Note that the structure of custom function definitions and literals has been omitted for brevity.

Swizzleflow is suitable for more than just data movement code like transpose. The code in Listing 4, which is our model of the convolution implementation from Listing 3, shows that we can model typical mathematical operations by unrolling loops and using the fold operator to model the additions the original loop performs, as illustrated in Figure 2.

Swizzleflow is a functional, single-static assignment language whose core grammar is given in Figure 3 The variables in a Swizzleflow program are all multi-dimensional arrays with a fixed shape or type, such as `[4, 2]` for a 4×2 array. During synthesis, these arrays mainly contain *terms*, which are either arbitrary symbols given by the specification or multisets of terms, though they can also contain the special values 0 (a general identity) and $\perp$ (an undefined value).

Gathers Swizzleflow's semantics are built from two types of operators: gathers and fold. Aside from the special-purpose reduction operator fold, all Swizzleflow functions are taken from the set of *gathers*: functions between arrays that cannot depend on the values of their arguments. Each array location in the output of a gather is defined to be loaded from some location in an argument or to be one of the special

```
int j = get_thread_id();
register float s1[4], s2[4], s3[4];
for (int i = 0; i < 4; i++)
  s1[i] = input[i][j - i % T];
for (int i = 0; i < 4; i++)
  s2[i] = s1[(3 * i + j) % 4];
for (int i = 0; i < 4; i++)
  s3[i] = __shfl_sync(FULL_MASK,
            s2[i], j + i % T);
return s3;
```

Listing 1: An optimized implementation of matrix transpose for a $T \times 4$ matrix on GPUs whose memory accesses provide increased performance. This is a variation on the Trove algorithm [5] that uses fewer in-register promotions.Three loops are required because `__shfl_sync`, used for inter-thread communication, requires all threads to broadcast the same variable when reading from their neighbors.

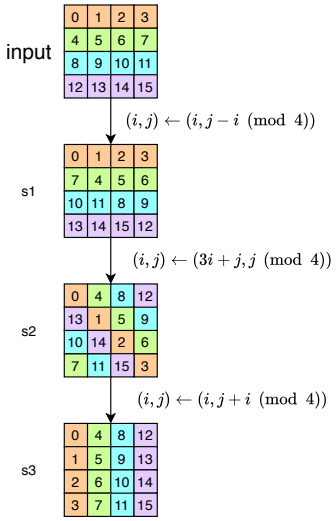

Figure 1: An illustration of our model of optimized GPU transpose, corresponding to the Swizzleflow code in Listing 2

```
memory:  [16] = [x0, ... x15]
input: [4, 4] = reshape(memory)
s1: [4, 4] = swizzle_row(\i, j. j - i % 4)(input)
s2: [4, 4] = swizzle_col(\i, j. 3i + j % 4)(s1)
s3: [4, 4] = swizzle_row(\i, j. j + i % 4)(s2)
```

Listing 2: A model of the transpose algorithm from Listing 1, where the parallelism in the original code is modelled with an additional array dimension. This code is written assuming 4 parallel threads, instead of the real value of 32, to make illustrating its execution more feasible.

```
int i = get_thread_id();
float loaded[2] = {x[i], x[W + i]};
float accum = 0;
for (int j = 0; j < K; j++)
  float to_send = loaded[i >= j ? 0 : 1];
  float received = __shfl_sync(FULL_MASK,
        to_send, i + j % W);
  accum += received;
out[t] = accum;
```

Listing 3: An optimized implementation of a width-K 1D convolution for GPUs, which uses two coalescing loads in order to load the input array onto W threads and then uses a pair of swizzles to coordinate passing the loaded values between threads in order to compute the desired result.

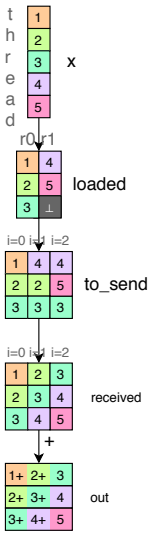

Figure 2: Illustration of our model of GPU kernel execution from Listing 4

```
x:        [34]    = [x0, ... , x34]
loaded:   [32, 2] = load_trunc(x)
to_send:  [32, 3] = select(\i, j. i >= j)(loaded)
received: [32, 3] = swizzle_col(\i, j. i + j % 3)(to_send)
out:      [32]    = fold(+) received
```

Listing 4: A Swizzleflow model of the GPU convolution from Listing 3, executed on 32 threads with a filter width $K = 3$. The usage of the `accum` variable in the native code is represented by the fold operator that produces `out`

values 0 or $\perp$. For example, the function that defines `s3` in Listing 2 and Figure 1 can be written as the gather $f(a)[i,j] = a[i, i+j \mod 4]$.

Gathers generalize permutations and can express operations such as loading data from memory, communication between threads, and many conditional statements (such as the question of in which iterations a value should be accumulated). The lack of value dependencies simplifies symbolic execution and analysis, and is justified since such expressions do not occur in most swizzling kernels.

In Swizzleflow, we restrict our variables (and thus function inputs and outputs) to having constant, bounded shapes. This means that Swizzleflow cannot reason about unbounded computations, which simplifies our analyses by making the set of locations within arrays that we need to track information about finite and easy to enumerate. This restriction does not have much impact on the usability of our model for swizzling kernels, since the program sections that are usually hand-optimized compute fixed-size subparts of larger problems.

Reductions and fold Many swizzling programs, such as convolutions and other stencils, ultimately combine their inputs using associative and commutative operators such as addition, multiplication, or max. Such reductions are also the only loop-carried dependencies seen in most swizzling kernels, which enables us to unroll loops into array dimensions, use gathers to describe the relationships between variables in the loop body, and then represent the only inter-iteration dependencies (reductions) with a special operator. To represent these reductions, the Swizzleflow languages includes the fold operator in addition to gathers, which packs the last dimension of an array into multisets, like so

$$\text{fold}\left(\begin{bmatrix} a & a \\ d & c \end{bmatrix}\right) = \begin{bmatrix} \text{fold}\{a,a\} \\ \text{fold}\{c,d\} \end{bmatrix}$$

For clarity, folds are usually annotated with the operation they are performing, so that, for example, $\text{fold}_+\{a,b\}$ can instead be written as $a+b$. Folds also have the following semantic equivalences that allow 0 to be the identity element:

$$\text{fold}\{a_1, a_2, \ldots a_k, 0\} = \text{fold}\{a_1, a_2 \ldots a_k\}$$
$$\text{fold}\{\} = 0$$

In all cases, nested folds are held to represent different arithmetic operations. The expression $\text{fold}_\times\{\text{fold}_+\{a,b\},c\}$ represents $c(a+b)$, and, with the annotations removed, it is $\text{fold}\{\text{fold}\{a,b\},c\}$. This means that folds cannot be merged in Swizzleflow, and so that $\text{fold}_+\{\text{fold}_+\{a,b\},c\} \neq \text{fold}_+\{a,b,c\}$ (because nested folds might represent different operations). This illustrates one limitation of our model: we cannot synthesize the precise manner in which a reduction is performed, such as by choosing between arithmetic instructions on the HVX processor.

We found that we can use our model to synthesize algorithms takes from multiple domains, including image processing, graphics, and cryptography, targeting both GPUs and Qualcomm's HVX processor. However, our model does not support data-dependent indexing or other indirection commonly seen in sparse kernels because the problem of synthesizing such code is difficult and outside the scope of this work. We also do not perform bit-accurate reasoning that would allow us to validate operators such as a fast inverse square root, whose correctness relies on the precise semantics of numerical operators. Despite these limitations on our scope (and our restriction to fixed-size subproblems), we have still developed a useful model for many swizzling programs.

3 Swizzleflow synthesis for transpose

Now that we have defined a model of dense swizzling programs, we can use it to perform program synthesis, allowing programmers to prototype their swizzling code. To illustrate our synthesis problem initially, we will use the GPU transposition code shown in Listings 1 and 2 (and illustrated in Figure 1). Since

```
for (int i = 0; i < 4; i++)
  s1[i] = input[i][?gpu_swizzle(j, i, T, 4];
for (int i = 0; i < 4; i++)
  s2[i] = s1[?gpu_swizzle(i, j, 4, T)];
for (int i = 0; i < 4; i++)
  s3[i] = __shfl_sync(FULL_MASK,
            s2[i], ?gpu_swizzle(j, i, T, 4));
```

Listing 5: The main body of the sketch needed to synthesize the GPU transpose from Listing 1.

```
memory:  [16] = [x0, ... x15]
input: [4, 4] = reshape(memory)
s1: [4, 4] = ?gpu_swizzle_row(input)
s2: [4, 4] = ?gpu_swizzle_col(s1)
s3: [4, 4] = ?gpu_swizzle_row(s2)
goal s3 == [[x0, x4, x8, x12], ...]
```

Listing 6: The synthesis problem for GPU transpose modelled in Swizzleflow, whole solution is in Listing 2. The specification has been shortened for clarity. Each `?hole` represents a set of functions for the synthesizer much choose from to satisfy the specified goal.

it does not feature reductions, this example will allow us demonstrate the core of how we make enumerative search viable for swizzle synthesis while deferring some details for Section 4.

One way to synthesize parts code like the fast transpose seen in Listing 1 would be to sketch the swizzling expressions by replacing them with holes like `?gpu_swizzle(i, j, 4, T)`, as shown in Listing 5 and using a solver to instantiate the holes with expressions (which are functions of relevant variables and their bounds) so that the resulting code computes a transpose, that is, for all i and j, $s_3[i][j] = input[j][i]$. This approach, used by Swizzle Inventor [14], the main previous work on swizzle synthesis, has difficulties scaling with the size of the program inputs and outputs and the complexity of the expression grammar due to limitations on how effectively SMT solvers can discover invariants about this search space.

Using Swizzleflow, however, the problem is phrased differently. The specification (or goal) of our synthesis problem is an assertion about the final output of a program being equal to some array of terms. The sketch is a program where the holes are sets of gathers of the same type, and the synthesis problem is to choose one gather from each set so that the program as a whole satisfies the specification. An example of such a synthesis problem can be found in Listing 6. While such a problem can be solved by many techniques, such as machine-learning guided search, we have found that a simple enumerative search, pruned by our viability tests, is sufficient to obtain scaleable synthesis performance.

Our search procedure is given in Algorithm 1

One of the main ways in which we made our synthesis effective was the *pair-reachability test*, or simply reachability test, which uses dataflow analysis to discover partial programs that cannot be completed to solve the synthesis problem. At a high level (as shown in Figure 4 for one term), when a candidate intermediate state is computed, this test allows us to check if each pair of terms in the output can be found in locations within that candidate those terms could come from (given the remaining swizzles). This test allowed us to, on average, avoid 99.56% of the paths in our search space and caused 45% of our benchmarks to explore the search tree as if they had an oracle that indicated if a partially-synthesized candidate was a partial solution.

3.1 Reachability for scalars in SSA programs

To simplify the discussion of this viability test, we will first present the core dataflow analysis for scalar program synthesis tasks in single static assignment (SSA) form. Because all Swizzleflow arrays have statically-known sizes, we can initially generalize from this scalar case by noting that each *location* in an array, that is, the variable v^i where array element $v[i]$ is held, is a scalar.

Consider the partially-synthesized program in Listing 7. The assertions in this program will fail because there is no way for the value of d to come from c, the only variable that has the value 2. We can prove that (partial) programs where variables can only take their values from other variables or constants cannot be correct in two steps: first by computing which variables the output of interested can come from, and then by checking if any of those variables contains the desired value. (The applicability of this type of analysis to Swizzleflow is discussed in the next subsection and shown in Figure 4).

In general, suppose we want to prove the assertion $F[g] = c$ (for some *goal variable* g and constant c) cannot be true for any final state F of a program that follows from executing statement s to produce a candidate state σ. The method we use for this bears some similarity to those used in taint analysis. We begin by defining $R_g(s)$, the set of variables v such that executing every statement after s could cause

Algorithm 1 Swizzleflow search procedure

function SEARCH(s, σ, g, c)
Require: s is a statement, σ is a program state (map of variables to values), g is the goal variable, and c is the value g must have have in a correct program
  for all choices $\hat{s}$ for how to execute s **do** ▷ Statements with no hole have one available choice so $\hat{s} = s$
    $\sigma' \leftarrow \hat{s}(\sigma)$
    if $\sigma'[g] = c$ **then** succeed
    else if $g \in \sigma'$ **then** fail ▷ Fail if the goal has a different value than expected
    else if VIABLE_PAIRS$(s, \sigma', g, c) \wedge$ VIABLE_COPY_COUNT(s, σ', g, c) **then** ▷ Viability tests, presented in Algorithms 3 and 4
      SEARCH(succ$(s), \sigma', g, c)$
    else continue
    end if
  end for
end function

```
a = 0 // R_d(a) = {a, b}, U(a) = {0}
b = 1 // R_d(b) = {a, b}, U(b) = {0, 1}
c = 2 // R_d(c) = {a, b}, U(c) = {0, 1, 2}
d = ?choose(a, b) // R_d(d) = {d}, U(d) = {0, 1, 2}
assert d == 2
```

Listing 7: An SSA program, annotated with the analyses we perform. $R_d(s)$ is the set of variables the value of d can come from when the program starting after the statement s is executed, and $U(s)$, the *universe* of s, is the set of constants that appear at or before s. `?choose` is an unmade choice of which variable to read from in the synthesis, and statements are named after the variable they define.

$F[v] = F[g]$, irrespective of the earlier state of the program. To solve this problem, we observe that, at the definition of g, only the variable of g can reach g, so $R_g(g) = \{g\}$.

Additionally, to ensure we do not reject valid programs simply because we have not executed a statement that defines a constant yet (as we would if we tested the viability of Listing 7 before defining `c`), we must also define the *universe* U of each statement to be the set of constants that have appeared in statements at or before s, starting with $U(\langle\text{init}\rangle) = \{\}$.

Then, as we work through the program (from the back for the reachability analysis and from the front for the universe one), whose statements have the general form

$$v = \text{choose}(a_1, \ldots a_k, c_1, \ldots c_l)$$

, where the a_I are variables and c_i are constants, we define

$$R_g(\text{pred}(v)) = \begin{cases} (R_g(v) \cup \{a_1, \ldots a_k\}) - \{v\} & v \in R_g(v) \\ R_g(v) & \text{otherwise} \end{cases} \quad (1)$$
$$U(v) = U(\text{pred}(v)) \cup \{c_1, \ldots, c_l\}$$

where pred(v) is the statement before the one that defines v. Note that, when we encounter the statement that defines a variable the goal can come from, we replace that variable with its arguments. This means that $R_g(s)$ is always a subset of the variables that are live at s (using a variable adds it to the set and defining it removes it).

With this information, we can prune an enumerative search as shown in Algorithm 2[1] For example, we can prune the partial program in Listing 7 by noting that, after executing statement c, we have $\sigma_c = \{a \mapsto 0, b \mapsto 1, c \mapsto 2\}$ and $R_d(c) = \{a, b\}$, but none of the variables that reach d (a or b) are equal to 2, which means the program cannot be correct. We could not invalidate the program at statements a or b, since no 2 had been encountered yet.

When generalized to Swizzleflow, this reachability analysis allows us to detect mistakes during synthesis, such as missing terms or terms that are in the wrong part of the array.

The reachability problem we defined has several useful properties: the solution at each statement is

[1]This algorithm's soundness relies on constants only appearing once within a program, but this condition can be imposed by rewriting subsequent appearances to variable references, like by rewriting $a = 1; b = 1$ to $a = 1; b = a$

Algorithm 2 Viability test based on reachability for scalar variables

```
function VIABLE_SCALAR(s, σ, assertions)
Require: σ is a state produced by executing some branch/choice of the statement s
    for all assertions g = c do
        if c ∈ U(s) and there is no v ∈ R_g(s) such that σ[v] = c then
            return false
        end if
        return true
    end for
end function
```

a subset of a finite set (the set of program variables V) and the inference rules for the dataflow analysis (Equation 1) distribute over union. These properties, shared by many dataflow analysis problems that can be expressed as tracking a bit-vector (of one bit per variable) will allow us to express the solution to problem as graph reachability using the IFDS framework developed by Reps [15]. To summarize the relevant part of the IFDS result, we can define a graph with nodes (s, v) for each statement s and variable V, and adding the edge $(s, v) \to (\text{pred}(s), v')$ if $v \in R_g(s)$ implies $v' \in R_g(\text{pred}(s))$. Then, the solution to our dataflow problem is the set of nodes reachable from (g, g) (the node representing the variable g at the statement g, which defines it) in this exploded graph.

The universe analysis is not a dataflow problem, but can be easily computed from the program. In addition to preventing certain classes of unsoundness in the case of multiple constants (or, in Swizzleflow's case, input arguments) it also allows us to use a definition of $R_g(s)$ only includes variables that are live at s. This reduces the amount of storage space needed to track this reachability data significantly.

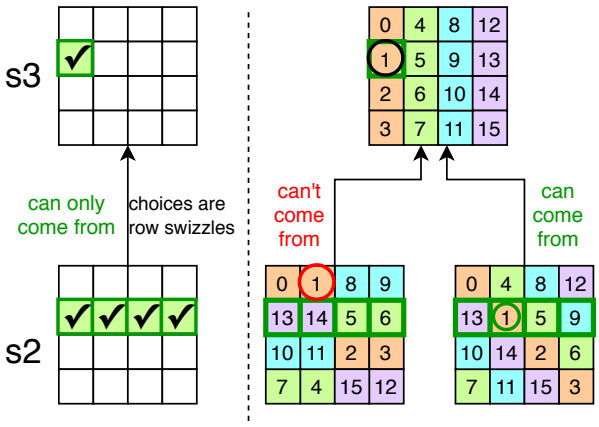

Figure 4: An illustration of how reachability information, in this case to the location $s3^{1,0}$ can be used to reject incorrect program states. $s3$ is produced by applying row swizzles to $s2$, which means that the value in $s3^{1,0}$ can only come from the locations $s2^{1,i}$ for each i. The colored, numbered boxes represent symbolic constants x_i

3.2 Reachability analysis in Swizzleflow

The algorithm presented above can be applied to Swizzleflow programs, as illustrated in Figure 4. Returning to our transpose synthesis example, part of our specification asserts that $s3[1, 0] = x_1$. If we treat the array location $s3[1, 0]$ as the variable $s3^{1,0}$, our reachability analysis shows that, because $s3$ produced by swizzling within the rows of $s2$,

$$R_{s3^{1,0}}(s2) = \{s2^{1,0}, s2^{1,1}, s2^{1,2}, s2^{1,3}\}$$

as shown in Figure 4. This then allows us to conclude that when none of the $s2^{1,j}$ in a candidate value of $s2$ is equal to x_1, like in the state on the left of Figure 4, cannot be correct.

In general, we can convert the assertion $g = a$ for some array a to the set of assertions $\{g^0 = a[0], g^1 =$

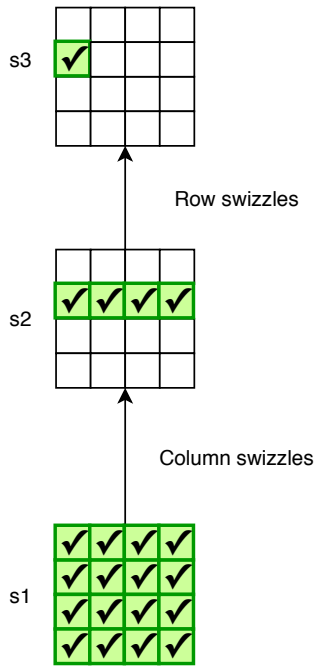

Figure 5: A demonstration of the fact that, because $s3$ comes from swizzles of the rows of $s2$ (that include rotations) and $s2$ is produced by swizzling $s1$'s columns, the value from any location in a value of $s1$ encountered while synthesizing Listing 6 can reach $s3^{1,0}$.

$a[1], \ldots\}$ and then use our analysis of R_{g^i} for each index i to prune our search during synthesis with appropriate invocations of VIABLE_SCALAR. This approach has one major problem, however, which is illustrated in Figure 5: tracking reachability to one location does not provide a specific enough abstraction to produce meaningful pruning. In practice, when we set our code to use reachability to one location, all of our benchmarks timed out.

To resolve this precision problem, we track which pairs of locations can reach which pairs of locations in a target array g, creating the pair-reachability problem R_{g^i,g^j}. Our ability to define this problem rests on the one key difference between Swizzleflow programs and the scalar ones we considered earlier: the execution cannot mix and match behaviors from different gathers for different variables. That is, if the statement defining a variable x' from $x = [1, 2]$ using uses the two functions

$$f([a, b]) = [a, b]$$
$$g([a, b]) = [b, a]$$

(which are the functions used in Figure 6) the value $x' = [1, 1]$ could not be constructed. However, if we

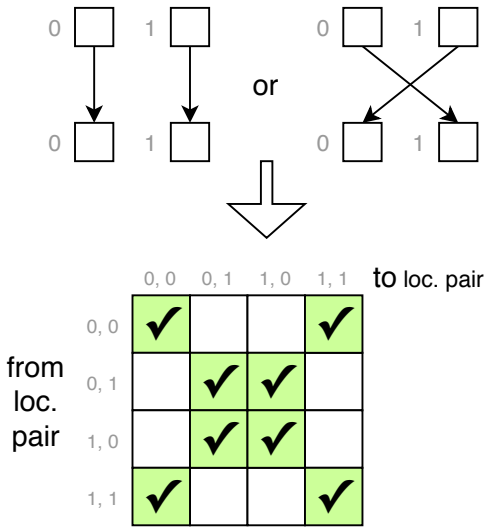

Figure 6: Construction of a statement matrix for the set of functions $f([a, b]) = [a, b]$ and $g([a, b]) = [b, a]$, showing which pairs of output locations can come from which pair of input locations.

had the abstracted assignment statements

$$x'[0] = ?\text{choose}(x[0], x[1])$$
$$x'[1] = ?\text{choose}(x[0], x[1])$$

which the expansion of f and g into scalar operations per our initial approach would imply, $x' = [1, 1]$ would be a valid outcome.

Instead of using this abstraction, we track reachability between pairs, as demonstrated in Figure 6 for the above example. Tracking the behavior of pairs of outputs does not eliminate this mix-and-match precision problem, but it reduces its presence enough that pruning becomes useful.

Algorithm 3 gives our main viability test, which uses this pair-reachability to check if state σ (arising from executing statement s) can lead to satisfying the assertion $g = c$

3.3 Computing pair reachability

To compute this reachability data, we first create a *statement matrix* for each statement in the sketch, as shown in Figure 6. This matrix records where each pair of locations in every live variable after a statement executes can come from in the variables live before that statement. This construction is analogous to the one used for single-variable reachability.

We can then multiply these matrices together, starting from the statement matrix for the final program output (which encodes that (g^i, g^j) can come

Algorithm 3 Viability test for the assertion $g = c$ based on reachability between pairs of locations

```
function VIABLE_PAIRS(s, σ, g, c)
    for all (a, b) ∈ U(s)^2 do
        for all i, j such that c[i] = a and c[j] = b do
            if there are no (l1, l2) in R_{g^i,g^j}(s) such that σ[l1] = a and σ[l2] = b then
                return false
            end if
        end for
    end for
    return true
end function
```

from (g^i, g^j) for all i and j) in order to create the reachability matrix for each statement s. Each viability matrix contains the solutions to the graph reachability problems arising from $R_{g^i,g^j}(s)$, since each statement matrix is the adjacency matrix for a layer of nodes in the exploded graph for all of those problems. The statement matrices can be used for all of the R_{g^i,g^j} dataflow problems because Swizzleflow's restriction to gathers means that the structure of the exploded graph corresponding to the problem remains the same no matter which pair of output locations we are computing information about — only the start node of the graph changes.

In addition to allowing us to compute all the reachability data needed for VIABILITY_PAIRS at once, this statement matrix approach has several implementation benefits. One is that, since multiple different programs may use the same set of gathers (especially if they are a set that was built in to Swizzleflow), the statement matrix for those functions can be cached, allowing Swizzleflow to avoid a potentially expensive recomputaiton. More interestingly, these statement matrices, in addition to being Boolean-valued, are typically quite sparse: for our GPU benchmarks, only two of them had more than 5% of their bits set, while 82% had a density less than 1%. While the resulting chained products did not share this low density (swizzling within rows composed with swizzling within columns continues to greatly increase the amount of available data movement options in the sketch, even if the complete loss of precision in Figure 5 is avoided), Boolean matrix multiply with one sparse input can be implemented much more efficiently than a general-purpose matrix multiplication. Using matrix multiply algorithms adapted from O'Neil [12], we reduced the multiplication times for computing each viability matrix needed for synthesizing a 32×5 transpose (which had dimensions 25600×25600) to between 0.08 seconds and 7 seconds from the 120 seconds each needed using a general floating-point matrix multiplication implementation.

The dimensions of the matrices mentioned above also indicate why we do not use triples of locations or even larger pairs to obtain more precision. While $(32 \cdot 5)^{2^2} \approx 2^{29}$ Boolean matrix entries per multiplication output can be managed, $(32 \cdot 5)^{3^2} \approx 2^{43}$ Booleans, even if they can be stared in a sparse format, is not a matrix product that can be computed in a time feasible for program synthesis. We did not elect to explore augmenting our test with selective refinement strategies, like those used by Wang *et al.* [21] and Guo *et al.* [8] because it had proved highly effective as is.

4 Viability testing with arithmetic

The pair reachability test developed in Section 3 can be extended to account for the fold operator present in Swizzleflow. The first problem in this extension can be seen in the problem of synthesizing the Swizzleflow convolution from Listing 4 using the sketch and specification shown in Listing 8. The specification for this problem is, as shown in Figure 2, $[x_0 + x_1 + x_2, x_1 + x_2 + x_3, \ldots]$. However, earlier variables, such as `received`, only contain terms like x_1, x_2, or x_3. Applying the VIABLE_PAIRS procedure as written would cause us to prune valid programs because, for example, $x_1 + x_2 + x_3$ is not present in any term in `received`.

Our solution to this problem, illustrated in Figure 7, is to modify VIABLE_PAIRS to instead investigate where subterms of a location in the goal can come from. That is, we take all the term equality tests and replace them with $v \in^* l$ (or, "v is a subterm of l in σ"). For example, in the loop over the i and j such that that $g[i] = a$ and $g[j] = b$, we instead want the i and j to satisfy $a \in^* g[i]$ and $b \in^* g[j]$.

To accompany this extension of the viability test, we define the abstract behavior of a fold to put mul-

```
x:        [34]    = [x0, ... , x34] // U(x) = {x_0, x_1, ... x_34}
loaded:   [32, 2] = load_trunc(x)
to_send:  [32, 3] = ?select(loaded)
received: [32, 3] = ?gpu_swizzle_col(to_send)
out:      [32]    = fold(+) received // U(out) = {x_0 + x_1 + x_2, ...}
goal out == fold(+) [[x0, x1, x2], [x1, x2, x3], ...]
```

Listing 8: A synthesis problem for the GPU convolution from Listing 3, whose Swizzleflow model (one of the solutions to this problem) is in Listing 4 and which is illustrated in Figure 2. Note that the specification includes compound terms like $x_0 + x_1 + x_2$, complicating the viability testing process.

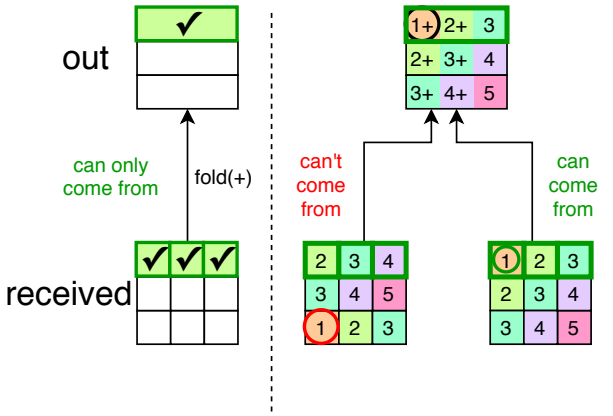

Figure 7: Left: A depiction of how the fold operator is abstracted for pruning analysis based on the sketch in Listing 8. Right: How this abstracted fold can be used to prune a state on account of how subterms of a desired output are not in places they can come from.

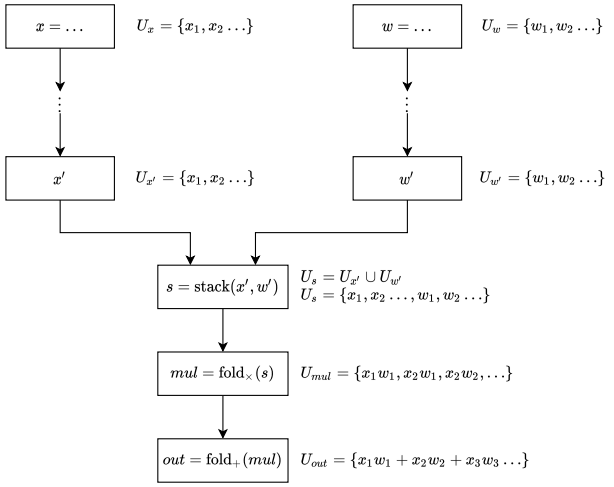

Figure 8: The computation of universes for each variable in a two-argument convolution

tiple values in a box as shown in Figure 7.. That is, if we have $v = \text{fold}\, w$ as a program statement, then the resulting statement matrix is built from pairs of edges of the form $v[i] \leftarrow w[i, k]$ for all relevant k. For example, in the convolution sketch, we define edges indicating that that out^0 can come from $received^{0,0}$, $received^{0,1}$, or $received^{0,2}$. This does mean that, before any reductions are executed during our search, our abstraction treats both $ad + bc$ and $ab + cd$ as $\{a, b, c, d\}$, which reduces the effectiveness of pruning.

One additional change we need to make to our viability test is the definition of $U(s)$, the universe at s. In the scalar and arithmetic-free cases, we could define $U(s)$ to be the set of constants that had appeared at or before s. To obtain more precise pruning (for example, the ability to notice that the term $x_1 w_1$ was not constructed when multiplying data with weights during a convolution), we extend our definition and have $U(s)$ be the set of terms that can appear within a variable that is live at s which are subterms of the goal.

Defining this set begins by defining U_v, the universe of every variable v as follows:

- If $v = [c_1, c_2, \ldots c_k]$ or some other literal, then $U_v = \{c_1, c_2, \ldots, c_k\}$
- If $v = f(a_1, a_2, \ldots, a_k)$ for some gather or set of gathers f, $U_v = \bigcup_i U_{a_i}$, which reflects that a gather can read from anywhere in any of its arguments.
- If $v = \text{fold}\, w$, then U_v is the set of all $\{t_1, t_2, \ldots t_k\}$ that are subterms of the specification such that each $t_i \in U_w$. In other words, the result of a fold can include all the ways to combine together the terms that might be in the input which appear in the specification.

A concrete example of these computations for a convolution that takes an array of weights as a second parameter is shown in Figure 8.

One caveat to this universe analysis is that it can still lead to correct programs being rejected in the

case where multiple independent literals use the same constants. For example, having the program

$$\begin{aligned} x &= [a, b]; \\ y &= f(x); \\ z &= [a, c]; \\ &\dots \end{aligned}$$

could lead to falsely rejecting a candidate if f replaces a with 0. Fortunately, this case can be detected and programs can be transformed to avoid it by removing the offending constants from one literal and adding a multi-argument gather, this issue does not fundamentally impact the correctness of our algorithm. We can, for example, rewrite the program from earlier to

$$\begin{aligned} x &= [a, b]; \\ y &= f(x); \\ z_0 &= [\bot, c]; \\ z &= [x[0], z_0[1]]; \\ &\dots \end{aligned}$$

which keeps x live until z is defined, preventing us from unsoundly rejecting programs (at the cost of somewhat lower pruning accuracy).

4.1 Copy count viability

One limitation of our pair-reachability test is that it cannot detect when there are too many copies of a value, as shown in Listing 9. This limitation arose in practice during larger polynomial multiplication benchmarks, where the synthesis task required choosing 16 conditionals that control when the values $a_1b_1, a_1b_2, a_2b_1, a_2b_2$ would be added to four accumulators. In the typical correct solutions, 8 of the synthesized conditionals would be `false`. However, using solely VIABLE_PAIRS, we would accept any choice for those 8 holes, since they were a superset of the correct program, causing the benchmark to time out.

To address this problem (taking the polynomial multiplication search time to 3 seconds), we added Algorithm 4, the copy count viability test. This copy count test allows us to detect when it must be the case that too many copies of a value will appear in the entirety of the final output. It rests on a computation of $\min(g \leftarrow v^i)$, the lower bound on how many times the value in v^i will be copied into the output. This can be computed by working backwards through a program sketch, beginning with $\min(g^i \leftarrow g^i)(g) = 1$ and, for any gather, noting that the minimum copies of each input location is the sum of the minimum copies of each of the output locations that read from it (plus any copies made by other statements). Taking the minimum over all gathers in a hole (and using the same "folds place multiple values in a box" abstraction seen in Figure 7) produces the solution to $\min(g \leftarrow v^i)(s)$ for the entire program.

5 Results

Our empirical evaluation focused on three main questions

1. How scalable is our algorithm? What causes it to stop being effective?
2. How effective are our viability tests?
3. How do we compare to Swizzle Inventor [14], the previous work on this problem?

5.1 Benchmarks

The Swizzleflow synthesis model can be applied to multiple platforms by changing what array dimensions (which can represent variation in time and space) mean and through changing the contents of the holes in the sketch. For example, on the HVX processor, we represented native swizzling instructions such as `valign`, which joins together portions of two registers, as gathers, and then have a `?hvx_recombine_vectors` hole that selects between instances of these instructions with various arguments. On GPUs, however, we used gathers that modelled the expressions generally found in swizzling code on the GPU so that experts could use Swizzleflow to discover them. This template, originally developed for Swizzle Inventor, consists of compositions of a fan operations with a rotation, with optional grouping in between each steps. We were able to generate gathers from all the expressions in this grammar and use those sets of functions for synthesis.

Our primary benchmark set, the *GPU benchmarks*, which were taken from Swizzle Inventor and represent computations from multiple computational domains. We varied the input sizes in these benchmarks to evaluate scalability. These benchmarks are

Trove (CRC) A $32 \times s$ matrix transposition using the algorithmic strategy from Ben-Sasson *at al.* [5]

Trove (Sum) A variant of CRC Trove that sums the values on each thread, eliminating the final in-register permutation.

```
x = ?choose(a, 0) // min(o ← x)(x) = 1
y = ?choose(a, 0) // min(o ← y)(y) = 1
z = ?choose(b, 0) // min(o ← z)(z) = 1
o = x + y + z // min(o ← o) = 1
assert o == a + b
```

(a) Program sketch

```
x = a
y = a
z = ?choose(b, 0)
o = x + y + z
assert o == a + b
```

(b) Partially-synthesized candidate

Listing 9: An example scalar program showing how the reachability test cannot detect extra copies of values. The candidate program on the right will not be rejected by the reachability test since the a could come from `x` or `y`. After defining y in the candidate, we can observe that two copies of a must reach o, which lets us reject the program since o needs to be reachable by at most one copy of a (computing $a + a$ cannot create $a + b$). The $\min(o \leftarrow v)(s)$ values count the minimum number of times v can appear in o (if execution starts at s).

Algorithm 4 Viability test using minimum copy counts

function VIABLE_COPY_COUNT(s, σ, g, c)
  for all $a \in U(s)$ **do**
    $t \leftarrow |c[i] = a|$ ▷ the expected number of copies of a
    $b \leftarrow \sum_{\substack{\sigma[v^i]=a \\ v \text{ live at } s}} \min(g \leftarrow v^i)(s)$ ▷ the lower bound on how many times a could appear in g, computed from bounds on each location.
    if $b > t$ **then return** false
    end if
  end for
  return true
end function

Trove (RCR) A variant on Trove (CRC), an example of which is shown in Listing 1, where the data is permuted while it is loaded from memory, allowing the final in-register permutation to be eliminated

1D Convolution Application of a length k filter that is passed as a parameter $w_1, \ldots w_k$ to an array of length $32 + k - 1$. This version, like the one used in Swizzle Inventor, does not swizzle the weight array, though we successfully synthesized versions of this benchmark that did.

2D Stencil Application of some fixed $k \times k$ filter to a $(4 + k - 1) \times (4 + k - 1)$ array loaded onto 16 threads

FFM (registers) Finite field multiplication. Algorithms for polynomials of degree 32 and 64 were simulated with degree 4 and 8, respectively, on Swizzle Inventor. The data was loaded into registers on each thread.

The one Swizzle Inventor benchmark we were not able to synthesize in less than 30 minutes was a variant on FFM that placed the data into shared memory for reasons we will discuss below.

In all of these benchmarks, we used the Level 3 grammar from Swizzle Inventor (all compositions of fans, rotations, and grouping) to construct our sets of swizzles, which includes with their full grammar of Boolean conditionals.

To show that our system can be used for multiple platforms, we also defined a set of swizzles representing several HVX instructions (`v[l]align`, `vshuff[o/e]`, and `vmux`, and a move) to any (two, if needed) registers in the synthesized input to construct a new register. Six of these holes were used to synthesize the data movement shown in Figure 9, which was taken from the HVX SDK's Gaussian filter [10] to create the **HVX Gaussian** benchmark. This benchmark used registers of length 8 to represent the length 128 registers seen on real processors, taking advantage of the length-invariance required by HVX programs.

All benchmarks, including the Swizzle Inventor comparisons, were run on a laptop with an Intel i7-8565U CPU and 16 GB of RAM. We synthesized all solutions, both to ensure results were deterministic (so that randomness in the hash function used to deduplicate gathers would not impact search time) and to provide a fair comparison against Swizzle Inventor, which used a simple cost model to guide Z3 on some benchmarks.

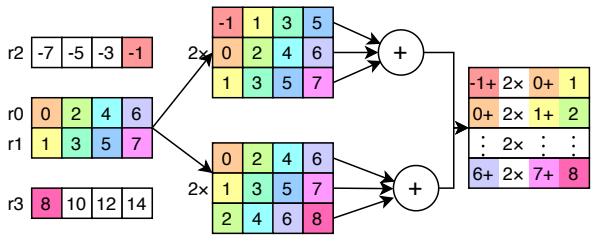

Figure 9: The data movement used in a handwritten Gaussian filter for the HVX platform which gives a 2x speedup over code generated by Halide. Synthesizing this code was our HVX Gaussian benchmark.

5.2 Scalability

Our main scalability results are shown in Figure 10.

These results show that we could synthesize smaller problems in seconds and that, when we moved to larger problem sizes, we only needed a minute or two to complete the synthesis of, say, a 9×9 2D stencil. Overall, they demonstrate that our algorithm scales effectively to larger instances of multiple problems. (The HVX Gaussian benchmark, not shown in the main plot, was synthesized in 0.6 seconds.)

The results show the polynomial scaling expected from the fact that matrix multiplication is a large portion of our algorithm. However, we obtain improved performance on some benchmarks, like the 1D convolution of size 11, by caching statement matrices between multiple experiments.

For most GPU benchmarks, the limit to our scalability was a lack of RAM. 32×11 transposes and length 15 1D convolutions (and transposes with sum) could not be computed due to memory limits. For example, the 32×11 transposes required us to keep in memory five matrices, each with $(32 \cdot 11)^2 \approx 2^{33}$ bits of required storage space. Since these matrices had significant variance in their densities and structure, we did not attempt to store them in a sparse format and instead accepted RAM as the current limit to our scalability.

In the 2D stencil benchmark, however, using an 11×11 filter required synthesizing there conditionals simultaneously in order to create code equivalent to `cond0 ? 0 : cond1 ? 1 : cond2 ? 2 : 3`, while previous benchmarks only required one or two conditionals. Enumerating these triples of expressions and constructing their corresponding statement matrix caused us to exceed our timeout of 30 minutes.

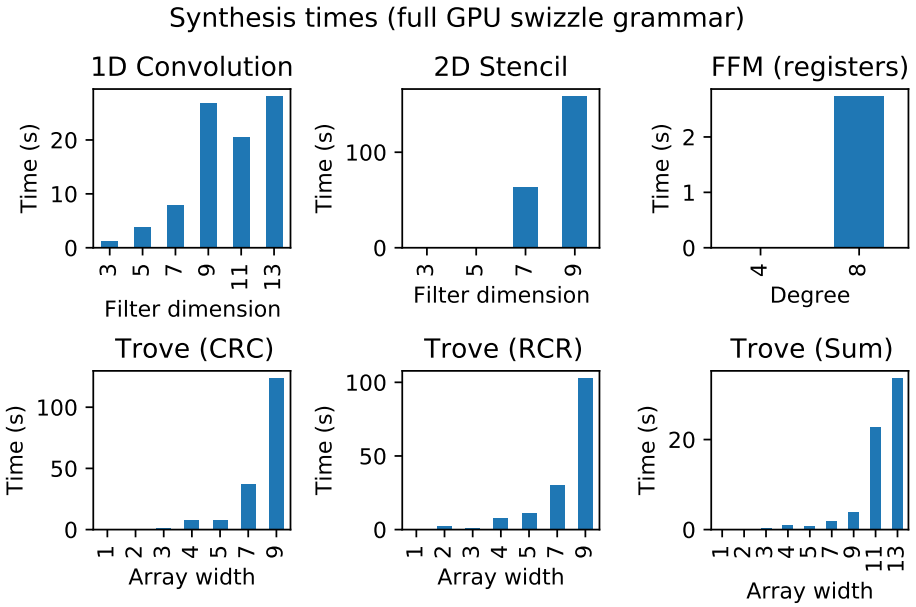

Figure 10: Swizzleflow performance on the GPU benchmarks vs problem size (as defined for each benchmark) using the full grammar of GPU swizzles. Times include both computation of reachability data and search for all solutions.

5.3 Pruning effectiveness

Our main metric for evaluating the effectiveness of our pruning method was comparing the number of states we constructed to how many were visited when using an oracle that had information on whether a given candidate state was part of a solution. 45% of our GPU benchmarks had this *ineffectiveness factor* of 1.0, meaning that the pruning algorithm visited no more states than the oracle did. Of the remainder, many, like the 7×7 stencil, had ineffectiveness factors close to 1, such as 1.01. The full results are shown in Figure 11.

One main cause for higher ineffectiveness factors, such as those seen in polynomial multiplication (where width 8 has an ineffectiveness of 27.6) came from the difficulty of combining swizzles for multiple arguments. In polynomial multiplication, we much choose elements of the inputs a and b that each thread will multiply on each loop iteration. When searching, many choices of a and b are valid individually but not compatible with each other, a circumstance our search procedure rarely detects until after the multiplications are performed. An oracle, however, would be aware of the impending incompatibility and terminate the search. A milder version of this effect appears in a variant of the 1D convolution benchmark that also swizzles the weights array, where the convolution can either occur left to right or right to left, but the incompatibles between when the data and weights are swizzled in the opposite way are not detectable until the $\text{fold}_\times$ operation is performed.

This multi-argument filtering problem is the cause of our one inability to replicate the width-8 shared memory polynomial multiplication benchmark from Swizzle Inventor within our 30 minute timeout (though we did complete it in an hour). Most of the time in that benchmark was spent filtering the $2240 \cdot 2240$ states that arose from swizzling a and b. Compared to the register benchmark, this cross product of possibilities was much larger due to the greater freedom afforded by loading directly from all of a (or b) during the computation.

Another source of ineffectiveness came from the limited precision of our pruning algorithm. This occurred to some extent in the row-column-row Trove benchmarks, where the expressiveness of the swizzle grammar led us to be unable to reject some states after the first swizzle (how data should be loaded from memory) was chosen because they were similar to correct ones. Applying the column swizzles (which were a smaller set of functions) allowed us to eliminate these mistakes, preventing them from escalating exponentially.

A more extreme example is the HVX Gaussian benchmark, whose ineffectiveness factor was 51.3. This arose from the fact that six of the HVX instruc-

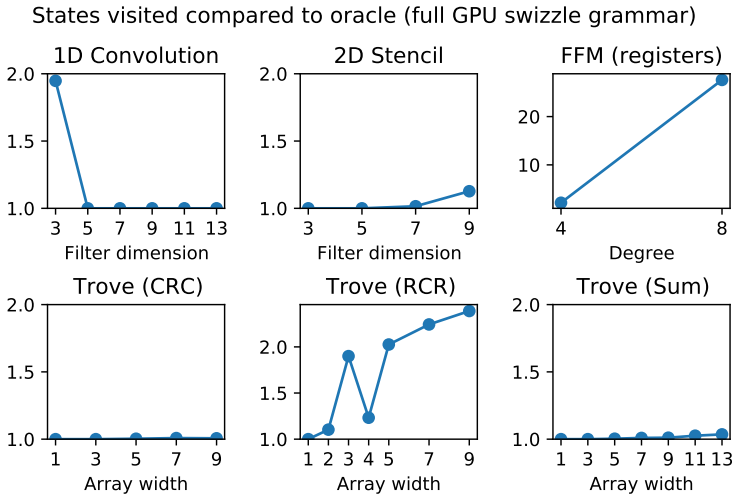

Figure 11: Plots of the *ineffectiveness factor* for Swizzleflow — how many states its search visited compared to an optimal pruning oracle for various GPU benchmarks.

tions could, from the perspective of the abstraction, move a pair of values from the initial input (which was available in each synthesis step as a source of arguments for the remaining swizzles) to any pair of locations in the output. This saturation was resolved after the first register was defined for each half of the computation (the computation of the odd or even results). This shows the continued necessity of controlling the expressivity of the synthesis grammar when using Swizzleflow, and demonstrates further avenues for research such as applying abstraction refinement in the Swizzleflow context.

Another piece of evidence for the overall effectiveness of our pruning techniques can be seen in Figure 12, which shows how Swizzleflow spent its time during selected benchmarks. This figure shows that our search procedure was rarely a significant portion of our run time (it often took much less than a second), which demonstrates the utility of our pruning methods and the importance of the viability matrices to quickly executing them. (A notable exception to this negligible search time is polynomial multiply with width 8, which took 3 seconds to complete its search process.)

This time breakdown also shows the advantages to caching statement matrices. This can be seen in the fact that CRC trove of size 7 spends some time creating statement matrices, while RCR trove with size 7, which uses the same holes in a different order, spends most of its time performing matrix multiplication.

5.4 Comparisons to Swizzle Inventor

As shown in Figure 13, our pruning method allowed us obtain dramatic speedups (frequently at least 100x!) over Swizzle Inventor on their benchmarks. In many cases, we solved benchmark sizes, such as a 7×7 2D stencil, that Swizzle Inventor could not.

Swizzle Inventor imposed grammar restrictions, such as limiting the constants considered for various operations, in order to successfully synthesize many of these benchmarks quickly. When we imposed similar restrictions on our swizzles, our stencil synthesis speedups remained, while the Trove benchmarks were only up to 5x faster in most cases. In some CRC Trove instances, Swizzle Inventor was faster with the restricted grammar (though, for both tools, synthesis took only a few seconds) due to their solver (Z3)'s ability to detect symmetries in and invariants about their grammar and sketch.

However, those limited speedups also show the advantage of Swizzleflow's pruning method. One key reason for our performance increase was that precomputing the abstract viability data takes significantly less time than Z3 or other solvers need to discover (or rediscover) invariants about their search space. Another cause of these speedups was that, in Swizzleflow, we synthesized over sets of gathers, not expressions, which allowed us to only explore many expressions in the GPU swizzle grammar that had different syntax but the same semantics only once, while Z3 had to find these equivalences or try incorrect programs multiple times.

These results show the great benefits that can come

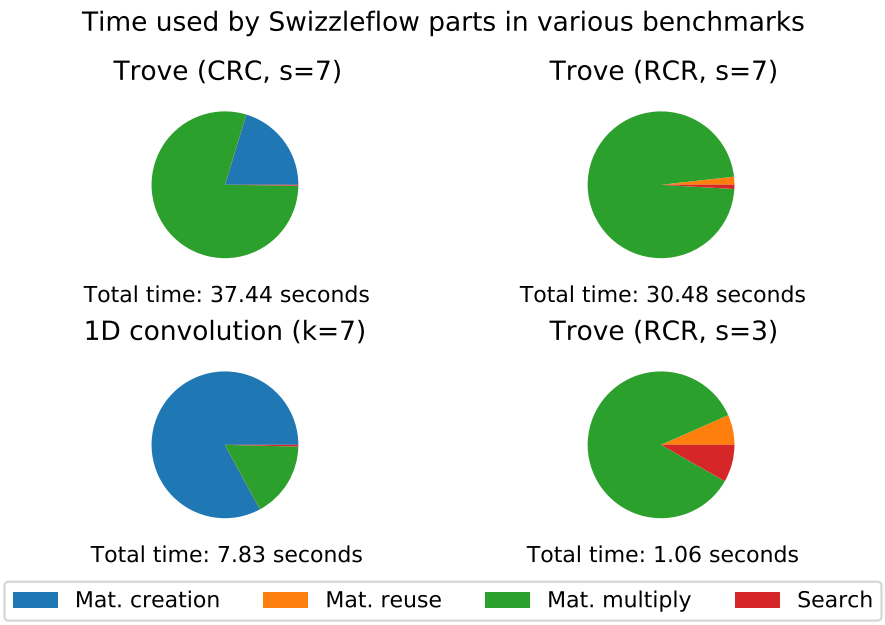

Figure 12: Time spent in different phases of Swizzleflow execution during selected benchmarks.

from using properties of the application domain to guide a synthesis problem.

6 Related Work

Accelerator performance improvement Many tools have been developed that attempt to use search or synthesis to create performant accelerator programs. Tools such as AutoTVM [6], Lift [18], and Tiramisu [3] primarily aim to solve the scheduling problem and find an efficient way to decompose a computation, such as a neural network, into smaller parts that can be executed optimally on an accelerator. While these systems do explore optimizations such as operator fusion and sometimes aim to generate efficient kernels, they do not explore the full space of swizzles.

Another class of tools, such as FFTW [7] and SLinGen [17] generate highly efficient code for particular operations (such as the Fourier transform or matrix algebra) and can sometimes target multiple platforms. This places them in a middle ground between the swizzle synthesis problem we solve and superoptimizers, such as Lens [13] or Souper [11], which aim to generate the most efficient code for any operation and often use similar pruning techniques to those seen is swizzle synthesis.

Swizzle synthesis The most directly related work on this problem is Swizzle Inventor [14], which defined our swizzle grammar and could synthesize complex data movement on GPUs. Similar work was performed by Cowan *et al.* [2], who aimed to synthesize efficient code for quantized versions of many mathematical operators (those that use integers instead of floating point numbers). SynthCL [19] also aimed to generate GPU swizzles, though did not search the same space of non-affine expressions as Swizzle Inventor. Work by Barthe *et al.* [4] and that seen in BitStream [16] aimed to synthesize swizzles in the context of x86's SIMD instructions, with BitStream focusing on cryptographic operations. In general, this prior work focused on synthesis tasks that targeted particular hardware and did not develop platform-independent models or abstractions.

Pruning enumerate synthesis Our dataflow analysis-based pruning approach is not entirely novel, having also been used by Mukherjee *et al.* [11] to prune away partially-instantiated candidates during superoptimization. Their work uses several different abstractions to analyze how expressions (both concrete ones and those with uninstantiated holes) manipulate individual bits and use this to reject programs. For example, their analyses observe that the expression `x << Constant` cannot be an optimization of `3 * x | 1`, since the latter's final bit must be 1, but the former's rightmost bit cannot be guaranteed to be 1. Their use of these techniques improved the performance of their enumerative search significantly,

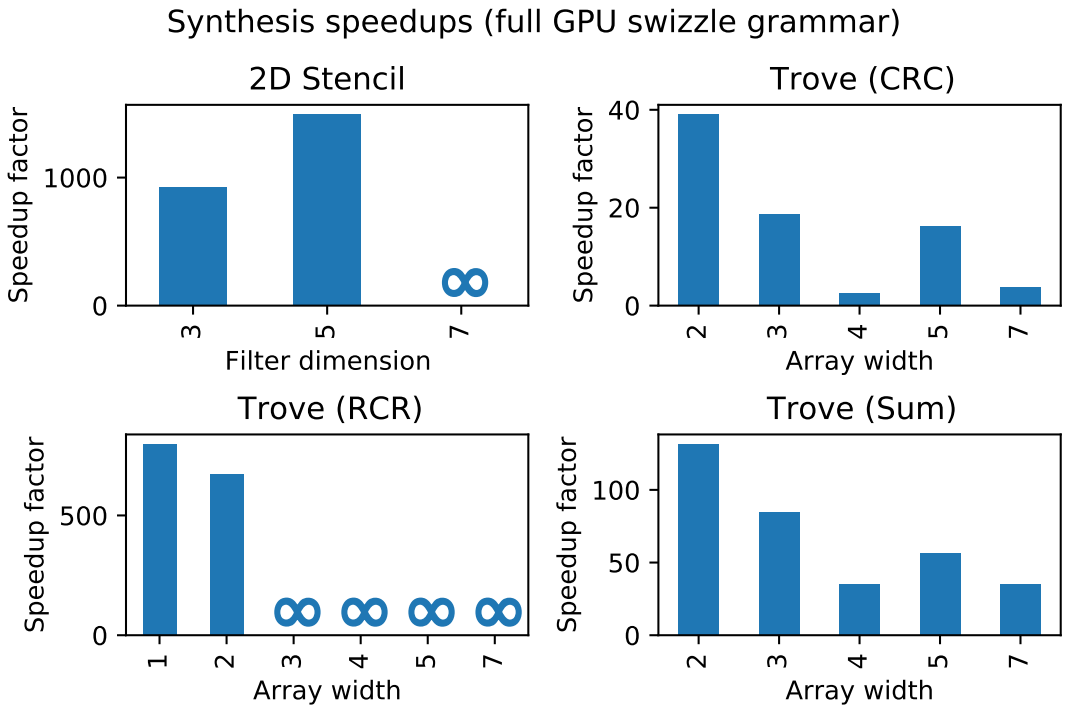

Figure 13: Speedup for Swizzleflow compared to Swizzle Inventor when using the full grammar of GPU swizzles defined by Swizzle Inventor for various GPU code synthesis benchmarks. Infinite speedups represent benchmarks we could solve that Swizzle Inventor could not solve in 30 minutes.

just as in Swizzleflow.

More generally, domain-specific abstractions that are used to detect that continuing a search would not be useful are seen throughout program synthesis literature, from Scythe [20], which uses a multi-phase process to more efficiently synthesize SQL queries, to the work of Guo *et al.* [8], which successively refines abstract representations in order to find a Haskell function with a given type signature. These abstractions are common enough that Wang *et al.* have developed a system [21] for learning such abstractions automatically to speed up synthesis problems.

7 Conclusions

We have presented Swizzleflow, a language that we have used to express the irregular mappings of data to memories and compute elements needed to obtain performance on mathematical accelerators such as GPUs and vector processors. Using this model, we have been able to synthesize these mappings more scalably than previous work using pruning methods based on dataflow analysis.

Acknowledgements I would like to thank my advisor, Rastislav Bodik, for his extensive advice and feedback on this project. This work would not be where it is without him. I would also like to thank Sam Kaufman for his assistance in developing many of the ideas presented here and Maaz Ahmad for help creating the HVX benchmarks and obtaining concrete performance numbers.

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Giancarlo Alfonsi et al. “Performances of Navier-Stokes Solver on a Hybrid CPU/GPU Computing System”. In: *Parallel Computing Technologies*. Ed. by Victor Malyshkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 404–416. ISBN: 978-3-642-23178-0.

[2] “Automatic generation of high-performance quantized machine learning kernels”. In: 2020,

pp. 305–316. ISBN: 9781450370479. DOI: 10.1145/3368826.3377912.

[3] Riyadh Baghdadi et al. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code". In: *CGO 2019 - Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (2019), pp. 193–205. DOI: 10.1109/CGO.2019.8661197. arXiv: 1804.10694.

[4] Gilles Barthe et al. "From relational verification to SIMD loop synthesis". In: *Principles and Practice of Parallel Programming (PPoP)*. Vol. 48. 8. 2013, pp. 123–133. ISBN: 9781450319225. DOI: 10.1145/2517327.2442529.

[5] Bryan Catanzaro, Alexander Keller, and Michael Garland. "A decomposition for in-place matrix transposition". In: *Principles and Practice of Parallel Programming, PPoPP*. 2014, pp. 193–206. DOI: 10.1145/2555243.2555253. URL: https://doi.org/10.1145/2555243.2555253.

[6] Tianqi Chen et al. "Learning to Optimize Tensor Programs". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 3389–3400. URL: http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs.pdf.

[7] Matteo Frigo. "A fast fourier transform compiler". In: *ACM SIGPLAN Notices* 39.4 (2004), pp. 644–655. ISSN: 03621340. DOI: 10.1145/301618.301661.

[8] Zheng Guo et al. "Program Synthesis by Type-Guided Abstraction Refinement". In: *POPL*. Vol. 1. January. 2020.

[9] Bastian Hagedorn et al. "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs". In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. PACT '20. Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 71–82. ISBN: 9781450380751. DOI: 10.1145/3410463.3414632. URL: https://doi.org/10.1145/3410463.3414632.

[10] Qualcomm Inc. *Hexagon DSP SDK Tools & Resources*. URL: https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools.

[11] Manasij Mukherjee et al. "Dataflow-based Pruning for Speeding up Superoptimization". In: vol. 4. November. 2020. DOI: 10.1145/3428245.

[12] Patrick E O'Neil and Elizabeth J O'Neil. "A fast expected time algorithm for boolean matrix multiplication and transitive closure". In: *Information and Control* 22.2 (1973), pp. 132–138.

[13] Phitchaya Mangpo Phothilimthana et al. "Scaling up Superoptimization". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16*. Atlanta, GA: ACM, Apr. 2016, pp. 297–310. ISBN: 9781450340915. DOI: 10.1145/2872362.2872387. URL: http://dl.acm.org/citation.cfm?doid=2872362.2872387.

[14] Phitchaya Mangpo Phothilimthana et al. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 65–78. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304059. URL: http://doi.acm.org/10.1145/3297858.3304059.

[15] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 49–61. ISBN: 0897916921. DOI: 10.1145/199448.199462. URL: https://doi.org/10.1145/199448.199462.

[16] Armando Solar-Lezama et al. "Programming by sketching for bit-streaming programs". In: *Programming Language Design and Implementation (PLDI)*. Vol. 40. 6. Chicago, IL: ACM, 2005, pp. 281–294. ISBN: 1595930566. DOI: 10.1145/1064978.1065045.

[17] Daniele G. Spampinato et al. "Program Generation for Small-Scale Linear Algebra Applications". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 327–339. ISBN: 9781450356176. DOI: 10.1145/3168812. URL: https://doi.org/10.1145/3168812.

[18] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "LIFT: A functional data-parallel IR for high-performance GPU code generation". In: *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE, 2017, pp. 74–85. ISBN: 9781509049318. DOI: 10.1109/CGO.2017.7863730.

[19] Emina Torlak and Rastislav Bodik. "A lightweight symbolic virtual machine for solver-aided host languages". In: *Programming Language Design and Implementation (PLDI)*. Vol. 49. 6. 2014, pp. 530–541. ISBN: 9781450327848. DOI: 10.1145/2594291.2594340.

[20] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. "Synthesizing Highly Expressive SQL Queries". In: *Programming Language Design and Implementation*. 2017. ISBN: 9781450349888. DOI: 10.1145/3062341.3062365.

[21] Xinyu Wang et al. "Learning abstractions for program synthesis". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10981 LNCS (2018), pp. 407–426. ISSN: 16113349. DOI: 10.1007/978-3-319-96145-3_22.