Swizzle Inventor: Data Movement Synthesis for GPU Kernels

Phitchaya Mangpo Phothilimthana[♡*] Archibald Samuel Elliott[†] An Wang[†]
Abhinav Jangda[‡] Bastian Hagedorn[◇] Henrik Barthels[♣] Samuel J. Kaufman[†]
Vinod Grover[♠] Emina Torlak[†] Rastislav Bodik[†]

[♡]University of California, Berkeley [†]University of Washington [‡]University of Massachusetts Amherst
[◇]University of Münster [♣]AICES, RWTH Aachen University [♠]NVIDIA

Abstract

Utilizing memory and register bandwidth in modern architectures may require *swizzles* — non-trivial mappings of data and computations onto hardware resources — such as shuffles. We develop Swizzle Inventor to help programmers implement swizzle programs, by writing program sketches that omit swizzles and delegating their creation to an automatic synthesizer. Our synthesis algorithm scales to real-world programs, allowing us to invent new GPU kernels for stencil computations, matrix transposition, and a finite field multiplication algorithm (used in cryptographic applications). The synthesized 2D convolution and finite field multiplication kernels are on average 1.5–3.2x and 1.1–1.7x faster, respectively, than expert-optimized CUDA kernels.

CCS Concepts • **Software and its engineering → Source code generation**; **Automatic programming**; • **Computer systems organization** → *Single instruction, multiple data.*

Keywords program synthesis, swizzling, GPGPU

ACM Reference Format:
Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304059

1 Introduction

Efficient execution on a GPU has always rested on a careful assignment of data onto the machine hierarchy (threads, warps, and thread blocks). Recent years have witnessed the emergence of a class of program optimizations that generalize these assignments and communication patterns into so-called *swizzles*, unlocking dramatic speedups. Informally, we define a swizzle kernel to be a program that coordinates multiple mappings of data and computations to hardware resources in order to achieve the correct result. These mappings may be complex and non-affine.

One such optimization, by Ben-Sasson *et al.*, introduces a *register cache* [3], a logical structure that distributes data across threads' private registers. The data is shared among threads in the same warp with swizzle patterns using an intra-warp shuffle instruction. An optimization by Catanzaro *et al.* [7] transpose a non-square matrix while performing only coalesced memory accesses. Their trick is to perform the transpose entirely in the registers of threads in a warp in three shuffling phases: column-wise, row-wise, and again column-wise. These papers demonstrated 1.5–45x speedup.

These results stem from the flexibility of swizzles, which gives them more power to simultaneously:

1. improve locality and/or parallelism,
2. utilize low-latency private memory such as registers,
3. satisfy hardware constraints, for example by coalescing global memory accesses, and
4. organize the computation so as to exploit special hardware support, such as an intra-warp data exchange.

Today's compilers cannot perform swizzling optimizations [2, 12, 28], and the swizzle kernels published in literature required significant human expertise [3, 7, 13, 18, 29].

This paper describes the design of Swizzle Inventor, a tool that aides in developing swizzle kernels. The programmer writes a program sketch that describes the algorithm but leaves the swizzles unspecified (Section 3); the tool then synthesizes those mappings to complete the sketch into a correct program. The algorithm with unknown mappings is easy to write because a swizzle corresponds to an array index expression. For example, index expressions used when arrays are copied determine data movement and thus also

This work was partly done when Mangpo, Sam Elliott, Abhinav, Bastian, Henrik, and Ras were visiting researchers at NVIDIA during summer 2018.
[*]Now at Google Brain.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04.
https://doi.org/10.1145/3297858.3304059

the mapping of data to memory. Similarly, index expressions used when computing on array elements determine the mapping of computations to threads. Although Swizzle Inventor is built for GPUs (Section 6), its underlying techniques can be used for other hardware targets.

Swizzle Inventor allows programmers to experiment. Imagine we want to determine whether there exists an algorithm with less computation. We sketch the algorithm, restricting it to fewer statements than the known algorithm, thus expressing the hypothesis. If the synthesizer can complete the sketch, we discover such an algorithm. If not, we have saved time by not attempting to develop the mappings ourselves. We used this process to discover many GPU kernels that are faster than expert-written ones (Section 7).

Swizzle Inventor can synthesize *input-independent* and *bounded* kernels. A kernel is input-independent if input values do not influence any control flow or array indexing. Thus, Swizzle Inventor cannot be used for restructuring sparse data at the moment. However, input-independent kernels are ubiquitous, including stencil computations, dense linear algebra, and cryptography operations. In a bounded kernel, loop bounds are constants known at compile time, and so are the dimensions of arrays accessed by the kernel. Bounded kernels are important in practice because they usually determine the overall performance; they are typically invoked by an outer loop that iterates over fixed-size tiles of an array with an unknown or dynamic size. Note that this is a typical way to handle unbounded matrices in GPU programming.

Automatic synthesis of swizzle kernels for GPUs poses new challenges. First, the SIMT execution requires the synthesized maps to be expressions over a thread identifier and loop iterations. Previously synthesized swizzle programs [1, 23, 25] used constant maps. To this end, we design a language of swizzles as a set of composable primitive swizzle operations that are (1) efficient to evaluate at runtime, (2) sufficient to compose into swizzles needed in advanced optimizations, and (3) efficient for synthesis (Section 5).

Second, to give the synthesizer more flexibility when mapping a program to hardware, it is sometimes useful to restructure the computation by reordering terms, such as terms in a summation of a stencil. We develop a representation of program states that uses canonical forms of algebraic expressions. This representation lets an SMT solver efficiently synthesize programs that reorder terms compared to how the computation is expressed in the specification (Section 4).

Using Swizzle Inventor, we discover new implementations that are better than expert-written ones: (1) a variant of a finite field multiplication that is on average 7–67% faster than the original version based on Ben-Sasson *et al.* [3]; (2) a 2D convolution kernel that is 3.2x and 2.3x faster than NVIDIA Performance Primitives (NPP) [21] and ArrayFire [32], respectively; and (3) a new algorithm for an in-place non-square matrix transposition, based on Catanzaro *et al.* [7], that does not need any intra-warp shuffles because swizzles are performed during the load and store, which remain coalesced despite performing a swizzle (Section 7).

2 Overview

In this section, we overview Swizzle Inventor by demonstrating how to implement an optimized 1D convolution GPU kernel, a weighted variant of the example in Ben-Sasson *et al.* [3]. According to Ben-Sasson *et al.*, caching input data using registers can be faster than using shared memory. Unlike shared memory, registers are private to an individual thread. Therefore, we have to explicitly use intra-warp shuffle instructions to exchange data between threads in each warp.

Program Sketch In Swizzle Inventor, programmers write an optimized implementation as a program sketch, which omits index swizzle expressions. For our running example, assume that the programmers already prefetch the input from global memory to registers *inp*[2][*warp_size*] as suggested by Ben-Sasson *et al.* Two rows of *inp* are sufficient to hold input data for each warp when $K = 3$. This part is straightforward, so the programmers handle it themselves. Therefore, the program sketch, shown in Listing 1, considers registers *inp* as the input to the program, which is distributed among all GPU threads in each warp.

Program sketches often naturally express high-level strategies. This particular sketch expresses that in each iteration, each thread (i) selects an input value it owns Ⓐ, (ii) exchanges the value with other threads Ⓑ, and (iii) multiplies the obtained value with the right weight before accumulating the result Ⓒ. Ⓐ and Ⓑ express the *Read-Publish* communication strategy, described by Ben-Sasson *et al.*

```
// These two constants are specified at compile/synthesis time
int K = 3;         // Filter size
int warp_size = 4; // Using 4 for illustration. Real warp size is 32.

float[warp_size] sketch(__register__ float inp[2][warp_size],
                        float w[K] /* in global constant memory */) {
  // Thread tid owns column tid of inp, out, and to_share.
  __register__ float to_share[warp_size], out[warp_size] = {0};
  for (int k = 0; k < K; k++) {
    // Threads in a warp run in parallel.
    parallel for (int tid = 0; tid < warp_size; tid++) {
      // Each thread chooses the value to share with other threads.
      to_share[tid] = inp[?swizzle1(tid,k)][tid]; Ⓐ
      // Solution: ?swizzle1 = (tid >= k) ? 0 : 1
    }
    parallel for (int tid = 0; tid < warp_size; tid++) {
      // Choose which thread to read data from.
      // This corresponds to a CUDA intra-warp shuffle instruction
      float tmp = to_share[?swizzle2(tid,k)]; Ⓑ
      // Solution: ?swizzle2 = (tid + k) % warp_size
      out[tid] += w[?swizzle3(tid,k)] * tmp; Ⓒ
      // Solution: ?swizzle3 = k
  } }
  return out
}
```

Listing 1. The program sketch for swizzled 1D convolution.

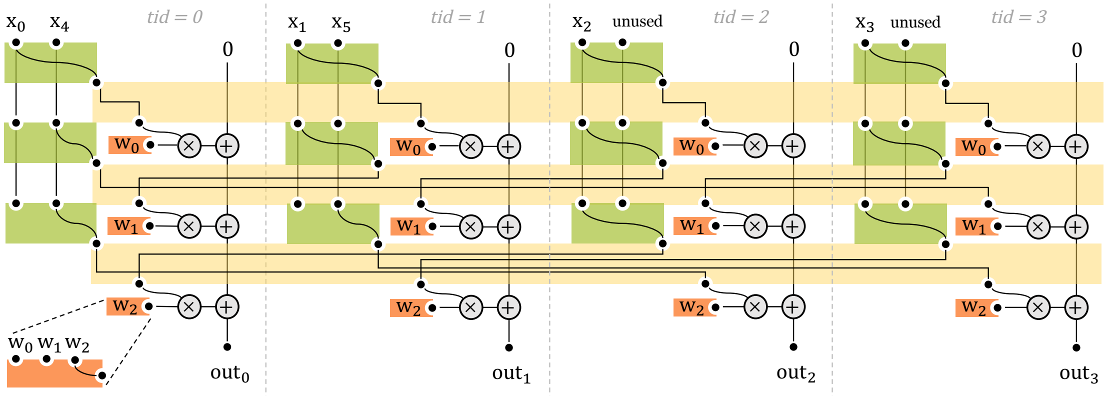


Figure 1. The execution of the synthesized convolution from Listing 1. Each column presents an execution of a thread *tid*. Values flow from top to bottom. The green, yellow, and orange boxes represent swizzles that (i) select a local input value, (ii) select a remote thread to receive data from, and (iii) select a weight, respectively.

precondition:	$inp =$	$[[x_0, x_1, x_2, x_3], [x_4, x_5, 0, 0]]$
	$w =$	$[w_0, w_1, w_2]$
postcondition:	$out =$	$[(w_0 * x_0) + (w_1 * x_1) + (w_2 * x_2),$
		$(w_0 * x_1) + (w_1 * x_2) + (w_2 * x_3),$
		$(w_0 * x_2) + (w_1 * x_3) + (w_2 * x_4),$
		$(w_0 * x_3) + (w_1 * x_4) + (w_2 * x_5)]$

Figure 2. Pre- and post-condition for the sketch in Listing 1

```
float[] spec(float inp[], float w[K], int n) {
  float out[n] = {0};
  for(int i = 0; i < n; i++)
    for(int k = 0; k < K; k++)
      out[i] += w[k] * inp[i+k];
  return out;
}

// Ensure that the implementation behaves like the spec.
float inp_spec[warp_size+K-1] = symbolic_matrix([warp_size+K-1]);
// inp_spec = [x0, x1, x2, x3, x4, x5]
float inp_impl[2][warp_size]  = reshape(inp_spec, [2, 4]);
// inp_impl = [[x0, x1, x2, x3], [x4, x5, 0, 0]]
float w[K] = [w0, w1, w2];
assert(spec(inp_spec, w, warp_size) == sketch(inp_impl, w));
```

Listing 2. Program specification for 1D Convolution

Swizzle Synthesis The difficult part of implementing this program is selecting (i) which local cached value to share Ⓐ, (ii) from which remote thread each thread receives data Ⓑ, and (iii) the right weight to multiply Ⓒ. These decisions are delegated to our synthesizer, which replaces ?*swizzle* with concrete swizzle expressions such that the program meets the specification.

Specification A specification to our synthesis problem is in the form of a program's pre- and post-conditions. A precondition is an input matrix with each of its elements initialized with a distinct *algebraic variable* or *algebraic expression* that captures a precondition of that specific matrix element. A post-condition is an output matrix whose elements are algebraic expressions of the input variables.

Writing a specification as pre- and post-conditions can be troublesome. Therefore, we let programmers write a sequential program as a specification. Programmers initialize input matrices with algebraic variables or expressions, and then symbolically interpret both the sequential program and the program sketch on these input matrices. Our correctness condition is simply the equivalence of the output matrices from the sequential program and the synthesized program. This process implicitly extracts pre- and post-conditions used by the synthesis problem.

The specification of 1D convolution is shown in Figure 2, which is derived from a sequential program in Listing 2.

Solution Our framework synthesizes the solution shown in the comments below lines Ⓐ, Ⓑ, and Ⓒ in Listing 1, which implements non-trivial swizzles illustrated in Figure 1, in eight seconds. Even though this is a simple 1D convolution, the synthesized implementation is 5% faster on a Kepler GPU compared to a shared memory implementation (for the filter of size 9; see details in Section 7.1).

Alternative Solution Another solution to this problem is:

```
?swizzle1 = (tid >= (K-1) - k) ? 0 : 1
?swizzle2 = (tid + (K-1) - k) % warp_size
?swizzle3 = (K-1) - k
```

In iteration k, it multiplies an input value by w$[2 - k]$. The output $out[0]$ from this solution is $(w_2 * x_2) + (w_1 * x_1) + (w_0 * x_0)$, equivalent to the expected output, despite the order of the summation. This reordering is necessary for performance improvement in some programs, such as a finite field multiplication (Section 7.2) and reducing values in an array of structures (Section 7.3).

3 Programming Abstractions

This section explains the key programming abstractions used in Swizzle Inventor in a hardware-independent setting. In this setting, Swizzle Inventor helps programmers implement *matrix programs* (Section 3.1) by completing unknown *swizzle expressions* in *program sketches* (Section 3.2).

3.1 Matrix Programs

Matrix programs can be used to represent various kinds of parallel programming models. Listing 1 is a matrix program that has inner parallel loops to imitate the implicit parallel thread execution in SIMT, and uses each column of a matrix to represent a local array belonged to one thread.

Index Swizzle Array access is where a swizzle happens. Therefore, we call an array index expression an *index swizzle*. An index swizzle maps integers to an integer. Index swizzles in a matrix program can be used to express patterns of various kinds of data movements, including local data access, data transfer from one type of memory to another, and data exchange between computing elements. Swizzle Inventor supports an index swizzle that is an expression of loop variables and constants, so it is input-independent. This restriction is required for our efficient synthesis algorithm. Note that an index swizzle does not have to be used directly at an array indexing location. It can be assigned to a variable that is later used as an array index for indirect array access. It can be used as a return value of a function that computes an array index.

Condition Swizzle Data movement can be determined by control flow, which we control with a *condition swizzle*. A condition swizzle maps integers to a boolean and can be used in branching or predication. Like an index swizzle, a condition swizzle must be input-independent.

Matrices and Loops All matrix sizes must be known at compile time. Values of matrix elements are values computed by expressions over input matrix elements and constants. Loops must be statically bounded. To handle an unbounded or dynamically-sized matrix, we can execute a bounded matrix program on each bounded tile of an unbounded matrix.

3.2 Program Sketches

Program sketches are matrix programs with some swizzles replaced with *unknown swizzles* or *holes* ($?swizzle$). We provide three kinds of holes.

Condition Swizzle $?sw_{cond}(v, \ldots)$ is a hole for a condition swizzle. The arguments to $?sw_{cond}$ are integer variables that may be used in the substituted expression.

Transformation Index Swizzle $?sw_{xform}(i, n, k)$ is a hole for an index swizzle such that the overall array access is a permutation or broadcast of the original data when considering all n elements. $?sw_{xform}$ takes three arguments:

i original index before applying the swizzle (primary index).

n number of elements to be swizzled, where $i \in [0, n)$, and n is statically known.

k runtime parameter (secondary index, e.g., another loop variable).

For the 1D convolution example in Listing 1, $?swizzle2$ is $?sw_{xform}(tid, n, k)$, and $?swizzle3$ is $?sw_{xform}(k, K, tid)$.

Partition Index Swizzle $?sw_{part}(n, i, \ldots)$ is a hole for an index swizzle that divides different values of $i, \ldots$ into a few partitions and produces the output value based on the partition. It can be used when the number of elements available is smaller than the number of accesses required. The $?sw_{part}$ hole takes two or more arguments: n the number of elements available (an array bound), and the rest of the arguments are integer variables that may be used in the expression. For the 1D convolution in Listing 1, we use $?sw_{part}(K, k, tid)$ for $?swizzle1$ because the total number of loop iterations K is more than the number of data values own by each thread's $inp[2]$.

4 Swizzle Synthesis

4.1 General Approach: Canonicalization

Synthesis Program sketches include holes that need to be instantiated with concrete swizzle expressions such that the resulting matrix program satisfies the specification.

Verification A matrix program meets a specification if all post-conditions hold. An output matrix X is equal to a specification matrix X' if they have the same shape and equivalent values at each index. Therefore, the key of our verification problem is to determine if value x is equivalent to value x', when x and x' are algebraic expressions such as $(a * b) + (c * d) + (e * f)$ and $(d * c) + (e * f) + (a * b)$, where a to f are algebraic variables.

Our approach is to represent a value in a canonical form so that values can be compared for equivalence by checking if they have the same form. Typically, to prove that two programs are equivalent, we use a constraint solver to check that they produce the same outputs for all possible concrete inputs. Using canonicalization, we do not need to use a constraint solver to check program equivalence.

This approach to verification-by-canonicalization is sound but incomplete; the verifier will never incorrectly conclude that two nonequivalent expressions are equivalent, but it may fail to discover some equivalences, such as that $a + a$ is equivalent to $2 * a$, and $a * (b + c)$ is equivalent to $(a * b) + (a * c)$. Swizzle Inventor treats these values as nonequivalent to simplify the synthesis problem and because optimizing algebraic expressions is not our primary goal.

4.2 Canonical Representation

For our prototype framework, we choose to handle only output values that are computed from a particular class of computations, namely reduction. While this choice restricts the kinds of programs that our tool can support, they already cover many interesting optimizations, as shown in Section 7.

Programming Abstraction We represent reduction using an *accumulator*. Users can create, update, and evaluate accumulators as follows, where → denotes operational semantics:

$$\begin{aligned} \texttt{create:}\;\; & o := \mathit{create\_accumulator}(\mathit{init}, \mathit{finalize}, \oplus[, \odot]) \\ & \quad \longrightarrow o := \mathit{init} \\ \texttt{update:}\;\; & \mathit{accumulate}(o, v, c) \longrightarrow \mathsf{if}(c)\;\; o := o \oplus v \\ & \mathit{accumulate}(o, [v_1, v_2, \ldots], c) \\ & \longrightarrow \mathsf{if}(c)\;\; o := o \oplus (v_1 \odot v_2 \odot \ldots) \\ \texttt{evaluate:}\;\; & \mathit{eval}(o) \longrightarrow \mathit{finalize}(o) \end{aligned}$$

An operator $\oplus$ is an outer reduction operator, and $\odot$ is an optional inner reduction operator. Reduction operators must be commutative and associative.

To modify Listing 1 to use accumulators, we replace the definition of *out* and how it is updated at ⓒ by:

```
// Definition
acc out[warp_size] = { create_accumulator(0, identity, +, *) };
// Update
accumulate(out[tid], [tmp[tid], w[k]], true);
```

Encoding We encode an output value v from an accumulator as an accumulation $a = (s, \mathit{init}, \mathit{finalize}, \oplus[, \odot])$, where s is a multiset (unordered list) of algebraic variables from input matrices. For example, we encode the expected $\mathit{out}[0]$ from our running example as $(\{\{w_0, x_0\}, \{w_1, x_1\}, \{w_2, x_2\}\}, 0, \mathit{identity}, +, *)$.

The encoding process happens while Swizzle Inventor interprets a program. We define the interpretation ($\rightsquigarrow$) of accumulator's create, update, and evaluate as follows:

$$\begin{aligned} \texttt{create:}\;\; & s := \mathit{create\_accumulator}(\mathit{init}, \mathit{finalize}, \oplus[, \odot]) \\ & \quad \rightsquigarrow s := \{\} \\ \texttt{update:}\;\; & \mathit{accumulate}(s, l, c) \rightsquigarrow \mathsf{if}(c)\;\; o := o \uplus \{l\} \\ \texttt{evaluate:}\;\; & \mathit{eval}(s) \rightsquigarrow (s, \mathit{init}, \mathit{finalize}, \oplus[, \odot]) \end{aligned}$$

Where $\uplus$ denotes multiset sum. Accumulators $a = (s, \mathit{init}, \mathit{finalize}, \oplus[, \odot])$ and $a' = (s', \mathit{init}', \mathit{finalize}', \oplus'[, \odot'])$ are equivalent if $s \equiv_{multiset} s'$, and all other fields are pairwise equal. Nested multisets s and s' are equivalent if they contain the same elements regardless of their order, in a recursive manner. A multiset is essentially a canonical representation of a reduction expression because a multiset ignores the order of elements but preserves their count.

Given that all other fields in a and a' are pairwise equal, $s \equiv_{multiset} s' \Rightarrow a \equiv a'$ because $\oplus$ and $\odot$ are both commutative and associative. For example, consider $\mathit{out}[0]$ from the specification and the alternative solution of our running example:

$$\{\{w_0, x_0\}, \{w_1, x_1\}, \{w_2, x_2\}\} \equiv_{multiset} \{\{w_2, x_2\}, \{w_1, x_1\}, \{w_0, x_0\}\}$$
$$(w_0 * x_0) + (w_1 * x_1) + (w_2 * x_2) = (w_2 * x_2) + (w_1 * x_1) + (w_0 * x_0)$$

4.3 Synthesis Algorithm

The synthesis problem is to select a swizzle expression for each swizzle hole in a program sketch such that the resulting matrix program meets the specification. Each hole describes a space of swizzle expressions to choose from. In this section, we assume that expressions in the space have unique id's. Our goal is to find the expression id for each hole; we call it the *control value* of the hole. The actual encoding of expressions in the search space of a swizzle hole is defined in Section 5.

While a matrix program has a single execution, a program sketch defines multiple executions, one for each combination of control values. Rather than systematically enumerating all possible executions, we represent all executions in a compact predicated program state. Recall that the state of a value of a matrix element in a matrix program is a canonical algebraic expression. In a program sketch, the state of an element's value will be a set of alternative canonical expressions. These alternatives arise from the choices of control values. To keep track of these choices, these alternatives are predicated by boolean expressions of control values. For example, x is $(\mathit{hole}_0 == 0)?\, a * b : (\mathit{hole}_0 == 1)? : a * c : a * d$.

We use Rosette [26, 27], a solver-aided language, to compute program states of a program sketch and find a solution for control values. Rosette, in turn, translates our query into an SMT formula and asks an SMT solver for a solution.

Objective Function In addition to satisfying the specification, our synthesizer searches for a solution that minimizes the number of accumulate statements because the fewer accumulate statements the less computation a program requires. If the predicate of an accumulate statement can be evaluated to false statically (e.g., $i < i$), such a statement is not counted. Swizzle Inventor does not yet model other hardware performance costs such as stalls from bank conflicts.

5 Synthesis Search Space

This section presents our encoding of the synthesis search space for the swizzle holes provided by Swizzle Inventor.

5.1 Search Space of Condition Swizzle

The following grammar defines the search space of a condition swizzle hole:

$$\begin{aligned} ?sw_{cond}(v, \ldots) :=&\;\; (v \mid \ldots) \odot_{cmp} (I \odot_{bin} (v \mid \ldots)) \\ \odot_{cmp} :=&\;\; = \mid \neq \mid \geq \mid > \mid \leq \mid < \\ \odot_{bin} :=&\;\; + \mid - \\ I :=&\;\; \text{integer constant} \end{aligned}$$

5.2 Search Space of Partition Index Swizzle

This index swizzle hole builds upon condition swizzle holes:

$$\begin{aligned} ?sw_{part}(n, v, \ldots) :=\;\; & \mathsf{if}\ ?sw_{cond}(v, \ldots)\ \mathsf{then}\ 0 \\ & \mathsf{elif}\ ?sw_{cond}(v, \ldots)\ \mathsf{then}\ 1 \\ & \ldots \\ & \mathsf{else}\ n - 1 \end{aligned}$$

Example The solution to the swizzle hole at Ⓐ in Listing 1 is if $(tid >= (0 + k))$ then 0 else 1.

5.3 Search Space of Transformation Index Swizzle

By examining swizzle expressions required to perform the in-place matrix transposition designed by Catanzaro *et al.* [7], we observe that their permutation expressions require operations $+$, $-$, $*$, $/$, and mod.

Naive Template Naively, we might create a grammar for search space as follows:

$$\begin{aligned} ?sw^{d}_{xform}(v,\ldots) &:= I \mid v \mid \ \ldots \ \mid \\ &\quad ?sw^{d-1}_{xform}(v,\ldots) \odot_{linear} ?sw^{d-1}_{xform}(v,\ldots) \\ &\quad ?sw^{d-1}_{xform}(v,\ldots) \odot_{bin} I \\ ?sw^{0}_{xform}(v,\ldots) &:= I \mid v \mid \ \ldots \\ \odot_{linear} &:= + \mid - \\ \odot_{bin} &:= * \mid / \mid \mathrm{mod} \end{aligned} \quad (1)$$

However, this grammar is unnecessarily general, and some example swizzles require a large depth d, making the search space extremely large and synthesis slow (Section 7.4).

Instead, we develop a template that can express all the kinds of permutations and broadcasts we have encountered, but rules out as many other expressions as possible.

Swizzle Primitives Our template will be composed of the following three permutation primitives, where $\widetilde{x}$ denotes a named integer constant that will be synthesized.

Equation (2) defines a *rotation permutation*, which rotates an array of elements by a constant R, as depicted in Figure 3.

$$\begin{aligned} rot(n)(k)(i) &= (i + R) \bmod n \\ \text{where} \quad R &= k * \widetilde{c_r} + \lfloor k/\widetilde{d_r} \rfloor + \widetilde{c} \end{aligned} \quad (2)$$

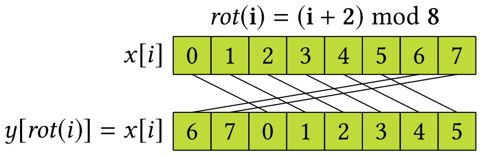
A *fanning permutation*, as defined in Equation (3), spreads out adjacent elements by putting them $\widetilde{c_f}$ distance apart.

$$\begin{aligned} fan(n)(i) &= (i * \widetilde{c_f} + \lfloor i/\widetilde{d_f} \rfloor) \bmod n \\ \text{where} \quad \widetilde{d_f} &= n/gcd, \quad gcd = GCD(\widetilde{c_f}, n) \end{aligned} \quad (3)$$

Imagine filling out an output array y by placing an element from the input array x one by one from the lowest to highest index. If $x[i]$ is placed at $y[j]$, $x[i+1]$ will be placed at $y[(j + \widetilde{c_f}) \bmod n]$. If $gcd = 1$, this insertion pattern has no collisions (Figure 4). Otherwise, collisions will occur. $\lfloor i/\widetilde{d_f} \rfloor$ handles these collisions, by shifting the element over if there already exists an element in the target location (Figure 5). Equation (3) provides this desired behavior because after $\widetilde{d_f} = n/gcd$ placements, a collision will happen, so it increments the output index by 1 after every $\widetilde{d_f}$ placements.

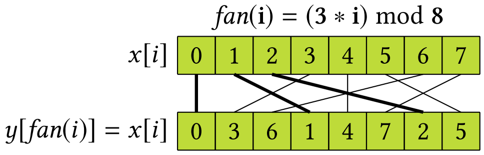
Any permutation may be *grouped*, using Equation (4).

$$group(gs, p)(i) = \lfloor i/gs \rfloor * gs + p(i \bmod gs) \quad (4)$$



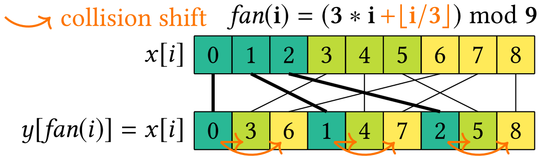
$rot(\mathbf{i}) = (\mathbf{i} + \mathbf{2}) \bmod \mathbf{8}$
$x[i]$
0 1 2 3 4 5 6 7
$y[rot(i)] = x[i]$
6 7 0 1 2 3 4 5

Figure 3. Rotation permutation. $n = 8, R = 2$



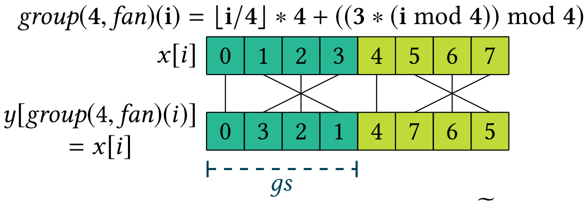
$fan(\mathbf{i}) = (\mathbf{3} * \mathbf{i}) \bmod \mathbf{8}$
$x[i]$
0 1 2 3 4 5 6 7
$y[fan(i)] = x[i]$
0 3 6 1 4 7 2 5

Figure 4. Co-prime fanning permutation. $\widetilde{c_f} = 3, \widetilde{d_f} = n = 8$



collision shift
$fan(\mathbf{i}) = (\mathbf{3} * \mathbf{i} + \lfloor \mathbf{i}/\mathbf{3} \rfloor) \bmod \mathbf{9}$
$x[i]$
0 1 2 3 4 5 6 7 8
$y[fan(i)] = x[i]$
0 3 6 1 4 7 2 5 8

Figure 5. Non-co-prime fanning permutation. $n = 9, \widetilde{c_f} = 3$, $\widetilde{d_f} = n/gcd = 3$



$group(\mathbf{4}, fan)(\mathbf{i}) = \lfloor \mathbf{i}/\mathbf{4} \rfloor * \mathbf{4} + ((\mathbf{3} * (\mathbf{i} \bmod \mathbf{4})) \bmod \mathbf{4})$
$x[i]$
0 1 2 3 4 5 6 7
$y[group(4, fan)(i)]$
$= x[i]$
0 3 2 1 4 7 6 5
gs

Figure 6. Grouping permutation. $n = 8, gs = \widetilde{d_f} = 4, \widetilde{c_f} = 3$

This divides the elements of the original array into multiple groups of gs consecutive elements, and permutes elements locally within each group based on the given permutation p (Figure 6). $\lfloor i/gs \rfloor * gs$ determines the starting location of each group, while $p(i \bmod gs)$ determines the permuted element offset within each group.

Our Template Our swizzle template, $?sw_{xform}$, is composed of these permutation primitives as follows. We use i for the index to be swizzled, n for the total number of indices (statically known), and k for an additional runtime parameter.

$$\begin{aligned} ?sw_{xform}(i, n, k) &= group(\widetilde{gs}, fan\_rot(\widetilde{gs}, k))(i) && (5) \\ \text{where} \quad & \widetilde{gs} \ \text{divides} \ n \\ fan\_rot(n, k) &= rotate(n, k) \circ fan(n) && (6) \\ rotate(n, k) &= \text{if } \widetilde{wrap} \text{ then } group(gcd, rot(gcd)(k)) \\ &\quad \text{else } rot(n)(k) && (7) \\ \text{where} \quad & gcd = GCD(\widetilde{c_f}, \widetilde{gs}) \end{aligned}$$

$?sw_{xform}$ is a grouped permutation whose subgroup is in turn permuted by fan_rot. If $\widetilde{gs} = n$, we essentially apply fan_rot

$$\mathbf{z}[(group(\mathbf{3}, rot) \circ fan)(\mathbf{i})] = \mathbf{x}[\mathbf{i}]$$
$$fan(\mathbf{i}) = (\mathbf{3} * \mathbf{i} + \lfloor \mathbf{i}/\mathbf{3} \rfloor) \bmod \mathbf{9}$$
$$group(3, rot)(\mathbf{i}) = \lfloor \mathbf{i}/\mathbf{3} \rfloor * \mathbf{3} + (((\mathbf{i} \bmod \mathbf{3}) + \mathbf{1}) \bmod \mathbf{3})$$

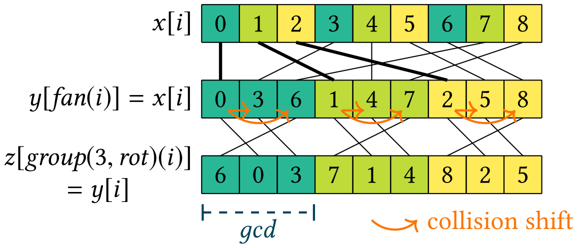


Figure 7. When swizzle expression is fanning followed by grouped rotation. $n = \widetilde{gs} = 9, \widetilde{c_f} = 3, \widetilde{d_f} = n/gcd = 3, R = 1,$ $\widetilde{wrap} = \text{true}$

on the entire array. Using function composition $\circ$, we define *fan_rot* as applying *fan* followed by *rotate*. The function *rotate* is either a rotation or a grouped rotation, depending on the synthesized boolean $\widetilde{wrap}$. The group size of the grouped rotation in Equation (7) is the GCD of $\widetilde{c_f}$ from *fan* in Equation (6) and $\widetilde{gs}$ from Equation (5). Figure 7 illustrates a complete permutation when *fan_rot* is *fan* followed by a grouped rotation, and $\widetilde{gs} = n$.

Overall, we synthesize $\widetilde{gs}, \widetilde{c_f}, \widetilde{d_f}, \widetilde{c_r}, \widetilde{d_r}, \widetilde{c}$, and $\widetilde{wrap}$ for each swizzle hole. In theory, we do not have to synthesize $\widetilde{d_f}$ because it can be computed in Equation (3). However, in practice, computing GCD requires recursion, which needs to be unrolled in order to generate constraints. The unrolling of the *GCD* function leads to inefficient constraints. Instead, we treat $\widetilde{d_f}$ as an unknown and synthesize it. We include an additional constraint that $\widetilde{d_f}$ must divide $\widetilde{gs}$, and $gcd = \widetilde{gs}/\widetilde{d_f}$. Note that this index template can express non-affine mappings because the divisors $\widetilde{d_r}$ and $\widetilde{gs}$ are not constant.

This swizzle template also lets us move values from a source array of size n_s to a destination array of size n_d, where $n_d < n_s$. Additionally, the template can express broadcasts ($y[i] = x[?sw_{xform}(i, n, k)]$) when $\widetilde{c_f} = 0$, $\widetilde{d_f} = n$, $\widetilde{wrap} =$ false, and the broadcasted element's index is R.

Example The solution to the swizzle hole at Ⓑ in Listing 1 is a rotation permutation with $R = k$ ($\widetilde{c_r} = 1$, $\widetilde{d_r} = warp\_size$, $\widetilde{c} = 0$), no grouping ($\widetilde{gs} = warp\_size$), no fanning ($\widetilde{c_f} = 1$, $\widetilde{d_f} = warp\_size$), and $\widetilde{wrap} = \text{false}$.

5.4 Search Space Restriction

Even our restricted swizzle template and grammars are more flexible than required for most programs. Since exploring the full search space defined by them takes a significant amount of time, we progress our synthesis by first exploring a very restricted search space and keep expanding it until we find a solution, similar to metasketches [4].

Most programs require only simple permutations with rotation or co-prime fanning and without grouping. Most condition expressions do not require any constants. Thus, we explore the search space in the following order:

Space 1: We prohibit grouping and non-co-prime fanning in $?sw_{xform}$ by fixing $\widetilde{d_f} = \widetilde{gs} = n$ and $\widetilde{wrap} = \text{false}$. We additionally fix $\widetilde{d_r} = n$ and restrict the constant in $?sw_{cond}$ to be zero.
Space 2: Same as Space 1 but with the full $?sw_{cond}$ grammar.
Space 3: The full search space.

The best solution in a larger search space may have a lower cost than a solution found in a smaller search space. Therefore, if we want an optimal solution, we will have to explore the full space. However, we show in Section 7 that the first solution found is often optimal and already faster than existing manual implementations.

6 Specialization for GPU Kernels

Thus far, we have described a general framework for programming swizzles. In this section, we discuss how we specialize this framework for GPU programming.

6.1 SIMT Programs

SIMT programs are a subset of matrix programs. The function `sketch` in Listing 1 can be written as lines 3–14 in the full GPU kernel sketch in Listing 3. Note that in SIMT, there are implicit parallel inner loops over threads in a warp, and local variables have an extra dimension for threads in a warp. Thus, the `sketch` function in Listing 3 is identical to that in Listing 1, including the shapes of *inp* and *out*. Notice that swizzle holes can be used as before. Listing 2 is then a specification of the function `sketch` in Listing 3 as well. In practice, we let programmers write program sketches in SIMT style.

```
 1  int K = 3; __constant__ float w[3];
 2
 3  __device__ inline float sketch(float inp[2], float w[K]) {
 4    int tid  = threadIdx.x % warp_size;
 5    float out = 0;
 6    for(int k = 0; k < K; k++) {
 7      float to_send = inp[?sw_part(tid, k)];
 8      // Intrinsic for exchanging values between threads in each warp
 9      float tmp = __shfl_sync(FULL_MASK, to_send,
10                              ?sw_xform(tid, warp_size, k));
11      out += w[?sw_xform(k, K, tid)] * tmp;
12    }
13    return out;
14  }
15
16  __global__ void conv1d(const float *x, float *y, int n) {
17    int global_id = blockIdx.x * blockDim.x + threadIdx.x;
18    int tid  = threadIdx.x % warp_size;
19    int warp_offset = global_id - tid;
20    float inp[2];
21    // Distribute X across registers of a warp of threads
22    global_to_local_1d<float>(x, inp, warp_offset, 1, 2); Ⓓ
23    float out = sketch(inp, w);
24    if(globalId < n) y[globalId] = out;
25  }
```

Listing 3. Full sketch of CUDA Kernel for 1D Convolution

From the perspective of a warp of 32 threads, the code in Listing 3 only produces 32 output values. However, a thread block contains multiple warps, and there is a virtually unlimited number of thread blocks to execute a kernel on different parts of a potentially unbounded input matrix.

Swizzle Inventor is implemented in Rosette [26, 27], a solver-aided language embedded in Racket. Therefore, users write a program sketch as a CUDA-style Racket (CuRacket) program. Ideally, we would like to support program sketches written in CUDA, but our prototype implementation currently supports only CuRacket programs. CuRacket restricts the bodies of if-then-else to be single expressions in order to help prevent thread divergence.

6.2 Load-Store Utility Functions

Since programmers still have to handle loading and storing data from and to global memory, our framework provides convenience functions to load and store a contiguous chunk of memory per thread warp or block.

The function below makes each warp collaboratively load a contiguous 1D chunk of memory from global memory A to local memory or registers rA as depicted in Figure 8.

$$global\_to\_local\_1d(A, rA, offset_x, slice_x, round_x, [shfl_x])$$

Each warp executes $round_x$ rounds of loading. In round $r_x \in [0, round_x)$, each thread $tid = (threadIdx.x\%warp\_size)$ loads consecutive $slice_x$ elements starting at position:

$$offset_x + r_x * warp\_size * slice_x + shfl_x(tid) * slice_x$$

The function takes optional shuffle functions $shfl_x$ to shuffle data while loading it (swizzling data distribution); the default shuffle function is $shfl_x(i) = i$ (no shuffle).

Listing 3 uses this function at Ⓓ to load data from global memory to registers, with $slice_x = 1, round_x = 2$ and no shuffle, just like Figure 8. Figure 9 depicts how this function works when $shfl_x(i) = (3 * i + 3) \bmod 4$. Note that threads in a warp access consecutive memory in each round if $shfl_x$ is a permutation, so this function has the desired property that the global memory accesses are fully-coalesced (when $slice_x = 1, 2,$ or 4).

We also provide a similar function $global\_to\_shared\_1d$ for loading from global to shared memory, as well as equivalent functions for storing to global memory, and 2D loads and stores. Swizzle Inventor encourages programmers to write a kernel with coalesced memory accesses by providing these load-store functions.

These functions essentially provide an assignment of data to threads similar to existing frameworks that support decomposing arrays and processing them using different levels of the compute hierarchy. These include Chapel [8], Hierarchically Tiled Arrays (HTA) [9] and Lift [10]. However, programmers still have to supply the *shfl* argument manually in these frameworks.

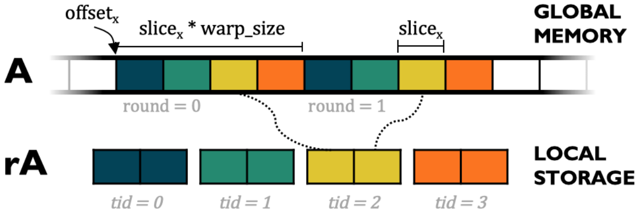


Figure 8. In function $global\_to\_local\_1d$, threads of a warp collaboratively load a consecutive 1D chunk of data from global to local memory or registers. This figure illustrates the simple case when $shfl_x(i) = i$.

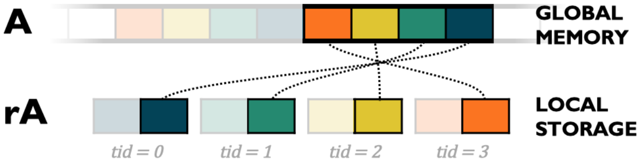


Figure 9. Function $global\_to\_local\_1d$ allows a shuffle of data while loading to local storage. This figure illustrates the behavior when $shfl_x(i) = (3 * i + i) \bmod warp\_size$. Thread tid gets slice $shfl_x(tid)$ in each round.

In Swizzle Inventor, programmers can ask the synthesizer to generate the right permutation by using *?swizzle* in the *shfl* argument. If *?swizzle* is used during load or store, the load or store must be inside a sketch function that is checked against a specification. For example, if we pass *?swizzle* as the *shfl* argument to the $global\_to\_local\_1d$ load function in Listing 3, then we must move the load function call inside the body of the `sketch` function, which has a specification. In our case study, we do not need to swizzle during loading or storing for 1D convolution, but this feature is useful for matrix transposition (Section 7.3).

6.3 Code Generation

Once our tool finds a solution to the synthesis problem, it can translate the solution into CUDA. For most CuRacket constructs, there is a straightforward translation from CuRacket to CUDA. Our framework provides the load-store utility functions in both CuRacket and CUDA.

Below, we discuss a few optimizations that are crucial for generating GPU kernels that use registers instead of the slower thread's private storage called *local memory*. Since CUDA will put arrays accessed with dynamic indices into local memory rather than registers, these optimizations avoid the use of local memory by turning accesses with dynamic indices into accesses with static indices.

Distributing Array References Consider Listing 3, the solution `inp[(tid >= k) ? 0 : 1]` has a dynamic index. If we modify this code by replacing it with `(tid >= k) ? inp[0] : inp[1]`, array `inp` is now referenced by a static index. Our code generator looks for dynamically referenced arrays as the result of conditionals and attempts to distribute static array references inside conditional expressions.

Barrel Array Rotation Some GPU programs require rotating a thread's local array by a dynamic amount. We follow the barrel rotation implementation as suggested by Catanzaro *et al.* [7]. The rotation can be performed in $\lceil \log_2 n \rceil$ steps, by statically iterating over the bits of the rotation amount R, and conditionally rotating each thread's array by distance 2^s at each step s if that bit of R is set.

7 Evaluation

We use Swizzle Inventor to implement GPU kernels that may benefit from advanced data movement. We measure the execution time of the kernels compiled with CUDA 9.2 on (i) NVIDIA Quadro GV100 Volta GPU (from 2017) and (ii) NVIDIA TITAN GK110 Kepler GPU (from 2013).[1] For each execution time data point, we report the median over three runs. Swizzle Inventor is available at http://github.com/mangpo/swizzle-inventor. The evaluation aims to answer three main questions.

Question 1 *Expressiveness: can Swizzle Inventor synthesize the kernels with swizzling optimizations in the literature?*

Swizzle Inventor is able to synthesize swizzling optimizations for stencil computation [3] (Section 7.1), finite field multiplication [3] (Section 7.2), and matrix transposition [7] (Section 7.3).

Question 2 *Inventiveness: can Swizzle Inventor invent new optimizations?*

Swizzle Inventor synthesizes a 2D convolution kernel that is on average 3.2x, 2.3x, and 1.5x faster than NVIDIA Performance Primitives (NPP), ArrayFire [32], and Iandola *et al.* [16] respectively. For finite field multiplication of degree 64 and higher, a synthesized kernel is on average 6% and 67% faster than the hand-written kernel from Ben-Sasson *et al.* [3] on Volta and Kepler, respectively. For non-square matrix transposition, we are able to invent a new algorithm that performs permutations during load and store, whose performance is on par with the state-of-the-art algorithm [7].

Question 3 *How long does Swizzle Inventor take to synthesize swizzling expressions?*

Swizzle Inventor can synthesize an optimal solution within 17 minutes for every kernel, and the median synthesis time is five seconds (Section 7.4).

7.1 Stencil

We used Swizzle Inventor to synthesize the following stencil computation kernels:

1D stencil (moving average), the running example in Ben-Sasson *et al.* [3];

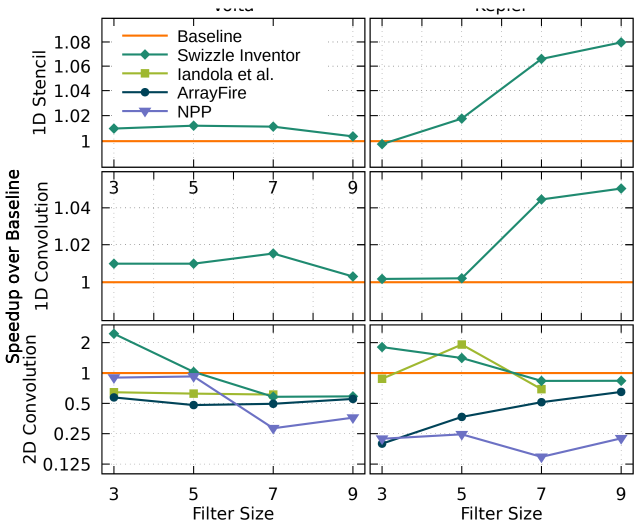


Figure 10. Stencil benchmarks showing speedup over shared memory version while varying the filter size

1D convolution similar to moving average with weights;

2D convolution (non-separable), a common kernel in image processing pipelines.

The program sketches for these kernels are similar to Listing 3 but each thread produces a tunable number of output values. The sketches prefetch input data from global memory to registers but leave out the details on how threads later obtain the correct values from these registers. The sketch for 2D convolution is slightly more complex as each thread owns a 2D *inp* array.

We were able to synthesize kernels that use a register cache similar to the solution by Ben-Sasson *et al.* [3]. Figure 10 displays the speedup of the synthesized register cache implementations over our hand-written shared memory kernels, based on the baseline in Ben-Sasson *et al.* The figure shows the speedup calculated from the execution time of the fastest configuration, considering the number of threads per thread block, the number of outputs per thread, and the shape of the 2D *inp* array.

For 2D convolution, we also compared against expert-optimized CUDA kernels from NPP [21], ArrayFire [32] (a high performance library for parallel computing), and Iandola *et al.*[15, 16] (which can run up to filter size of 7). On average, our kernel is 3.2x, 2.3x, and 1.5x faster than these three respective algorithms. ArrayFire either computes convolution directly using shared memory or applies convolution FFT. Iandola *et al.* provide an implementation with tunable parameters to control an amount of work done by each thread and where to store an input image (i.e., global, texture, shared, and registers), but do not exploit intra-warp shuffles. We tuned Iandola *et al.* for the best configuration for each filter size; NPP and ArrayFire do not expose tuning parameters. Our synthesized kernel is faster than NPP and ArrayFire on all filter sizes running on both GPUs. It is also

[1] To serve as a direct comparison to the evaluation in Ben-Sasson *et al.* [3].

faster than Iandola *et al.*, except on filter size of 5 on Kepler, for which the best configuration is to put the input image in the texture memory. To synthesize this implementation, we would need a different program sketch.

7.2 Finite Field Multiplication

Multiplication of two finite field elements is a building block and a major computational bottleneck of many cryptographic applications. Finite field multiplication (FFM) can be reduced to a small number of polynomial multiplications, as shown by Ben-Sasson *et al.* [3]. In this case study, we synthesized polynomial multiplication kernels of different degrees:

Mult-32 (32-degree) used in FFM of degree 32;
Mult-64 (64-degree) used in FFM of degree 64 or higher.

We obtained two baselines from the implementation [11] of [3] (i) using shared memory and (ii) using register cache. We wrote two program sketches, (i) using shared memory and (ii) using register cache, based on the algorithm presented by Ben-Sasson *et al.* and let the synthesizer fill in how the cached data is accessed. Figure 11 displays the speedup of FFM of degree 32 and 64 of the different implementations over shared memory version from Ben-Sasson *et al.*, when varying the number of multiplications. Figure 12 displays the speedup of FFM of degrees 64–384, which use Mult-64 as its sub-kernel, with a fixed number of multiplications.

For Mult-32, our synthesized kernels for both shared memory and register cache versions are logically identical to the baseline implementations. However, our implementations and the baselines differ on the usage of `if-else` statements and the order of thread index computation and shuffle instructions. These small differences result in an 8% speedup on average, when comparing our register cache version and the register cache baseline in degree-32 FFM, as shown in the first row of Figure 11.

The synthesized Mult-64 kernels, on the other hand, offer significant speedup over the baseline kernels as evidenced in the second row of Figure 11. Our synthesized shared memory implementation offers on average 9% and 40% speedup over the shared memory baseline on Volta and Kepler respectively. Our synthesized register cached implementation is on average 7% slower than the register cache baseline on Volta, but on average 15% faster on Kepler. Overall a synthesized kernel is the fastest configuration on both GPUs: shared memory on Volta and register cache on Kepler.

When the degree is equal to or greater than 64, the synthesized shared memory implementation is the best version on both Volta and Kepler as shown in Figure 12. It offers, on average, 6% and 67% speedup over the manual shared memory kernel on Volta and Kepler, respectively.

To explain why the synthesized Mult-64 kernels are faster than the manual baselines, we must take a look at our program sketch. Below displays the main loop of the sketch that uses register caching.

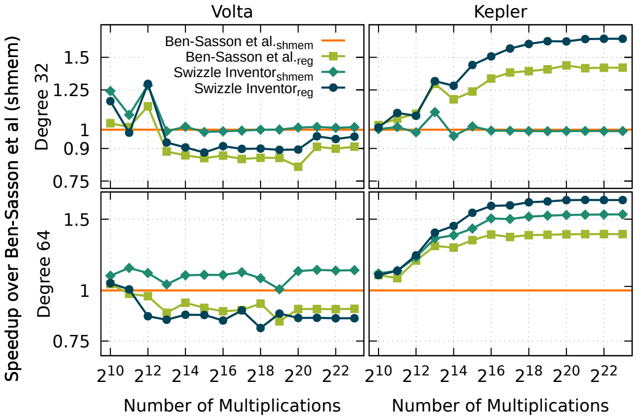


Figure 11. FFM speedup when varying number of multiplications

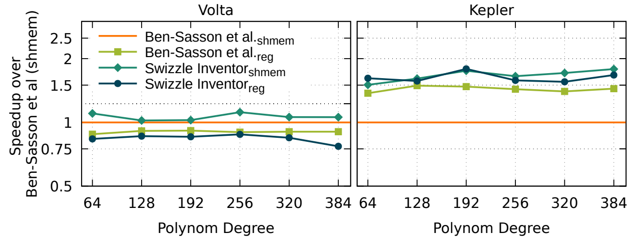


Figure 12. FFM speedup when varying polynomial degree

```
// A warp computes 128 values for one polynomial multiplication, so
// each thread computes 128/32 = 4 outputs (ans0, ans1, ans2, ans3).
acc ans0 = create_accumulator(0, identity, ^, &);
...
for(int k = 0; k < 32; k++){ // Compute all outputs in 32 iterations.
  // Use intra-warp shuffles to obtain elements cached in rA and rB.
  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 with all combinations of a0/a1 & b0/b1.
  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, and ans3, so there are 4x4 total accumulates.
  ...
}
```

There are 16 accumulate statements total in this program sketch. Our synthesizer found a solution with six accumulate statements (the other ten have false predicates), instead of eight as in the baseline program; note that accumulate statements with false predicates are eliminated away by our code generation. Our synthesizer is able to discover a better implementation because it considers more complex swizzles to obtain $a0$, $a1$, $b0$, and $b1$, compared to the baseline program's. Similarly for shared memory, we let the synthesizer make the same decision, but instead of using intra-warp shuffles, the synthesizer selects which row and column of *sA* and *sB* in shared memory each thread reads from.

7.3 Matrix transposition

Catanzaro *et al.* show how transposition of a non-square matrix can be used to accelerate loading from or storing to arrays of structures (AoS) in GPU global memory [7]. One can load from global memory in a coalescing fashion and then transfer data to the target threads by transposing data among the threads in the same warp via registers or shared memory. In this case study, we synthesized matrix transposition using registers for the following use cases:

AoS-load-store loads an array of structures and stores it back in a structure of arrays format.

AoS-load-sum loads an array of structures and computes a sum of the values in each structure.

The first baseline is our manual implementation using shared memory without data swizzling. It transposes the matrix during the load to shared memory, and both load and store are coalesced. Additionally we obtained two more implementations from the implementation [6] of [7]. Catanzaro *et al.*$_{direct}$ performs a direct load from global memory to the threads without matrix transposition, which may incur non-coalescing loads. Catanzaro *et al.*$_{shfl}$ implements the transpose algorithm using registers that is composed of three operations: column shuffle, row shuffle, and column shuffle. In the GPU setting, a column shuffle corresponds to a local array permutation, and a row shuffle corresponds to data exchange via an intra-warp shuffle. The paper presents an elaborate process to derive the correct shuffling formulas.

Using Swizzle Inventor, we wrote a program sketch that performs the three-step shuffle operation, as shown in the pseudocode in Listing 4. The framework then synthesizes the shuffling formulas. For AoS-load-sum, the last column shuffle is unnecessary because the order of elements in a column does not matter as we only care about the sum of the column, so our program sketch only includes a two-step shuffle. This requires the program verification to treat summations of the same list of elements as equivalent despite their orders, as supported by our encoding (Section 4.2). Our framework can synthesize and generate code (Swizzle Inventor$_{crc}$) that performs as well as Catanzaro *et al.*$_{shfl}$, as shown in Figure 13.

Additionally, we used Swizzle Inventor to devise a new transpose algorithm for AoS-load-store that applies row-column-row shuffles instead of the original column-row-column shuffles. The benefit of this new strategy is that we can perform the row shuffles together with load and store (via *global_to_local* and *local_to_global* functions) without using any intra-warp shuffle instructions. The synthesizer was able to synthesize shuffling expressions for all struct sizes. The performance of our new algorithm (Swizzle Inventor$_{rcr}$) is the same as the original algorithm, as shown in Figure 13.

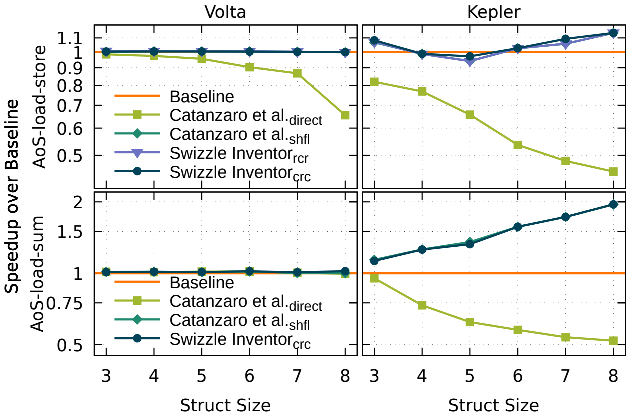


Figure 13. AoS-load-store and AoS-load-sum speedup when varying struct sizes.[2] Note that Swizzle Inventor$_{crc}$ is on top of Swizzle Inventor$_{rcr}$ and Catanzaro *et al.*$_{shfl}$.

```
for i, j in [0...m), [0...n)
    t1[i][j]  = inp[?swizzle1(i,j)][j] // Step 1: column shuffle
for i, j in [0...m), [0...n)
    t2[i][j]  = t1[i][?swizzle2(j,i)]  // Step 2: row shuffle
for i, j in [0...m), [0...n)
    out[i][j] = t2[?swizzle3(i,j)][j]  // Step 3: column shuffle
```

Listing 4. Pseudocode sketch of matrix transposition, composed of column-row-column shuffles

7.4 Synthesis Time

We synthesized all kernels in the three case studies on MacBook Pro 3.1GHz Intel Core i7 with Rosette 2.2. The synthesis time is reported in Table 1. Recall from Section 5.4 that our framework performs the search in three search spaces from the most restricted space to the complete search space (Space 1–3). Table 1 reports time to find a solution in each search space. The "TTS" column reports Time To find the first optimal Solution when the synthesizer starts searching from Space 1 to Space 3. For example, if the optimal solution is found in Space 2, and not in Space 1, then the TTS is equal to the sum of time in Space 1 and Space 2, excluding time in Space 3. For all kernels except Mult-64 with register cache, the first solution found is optimal. For Mult-64 with register cache, the first solution found in Space 1 is the same implementation as Ben-Sasson *et al.*$_{reg}$, but the solution found in Space 2 has a lower cost. Overall, our synthesizer can find an optimal solution within 17 minutes for every kernel, but the median TTS is five seconds.

Comparison to Naive Template Our transformation index swizzle template is key to our fast synthesis. To show its effect on the synthesis time, we synthesized the easier benchmarks (Mult-32 and AoS-load-sum) by replacing the

[2] The solution for AoS struct size = 3 can be used for struct size = 6 by always moving two elements as a unit (using `int2` datatype). Similarly, the solution for struct size = 1 and 2 can be used for struct size = 4 and 8, respectively, by always moving four elements as a unit (using `int4` datatype).

Benchmark	Version	Synthesis Time (s)			
	(Struct Size)	Space 1	Space 2	Space 3	TTS
1D stencil	reg	10	11	16	10
1D conv	reg	8	8	15	8
2D conv	reg	39	155	154	39
Mult-32	shmem	1	1	3	1
	reg	1	1	3	1
Mult-64	shmem	479	865	–	479
	reg	((118))	867	–	985
AoS-Sum	crc (1)^2	1	1	3	1
	crc (2)^2	(1)	(1)	4	6
	crc (3)^2	1	1	9	1
	crc (5)	3	3	15	3
	crc (7)	4	4	29	4
AoS-Store	crc (1)^2	1	1	3	1
	crc (2)^2	(1)	(1)	8	9
	crc (3)^2	1	1	12	1
	crc (5)	3	3	21	3
	crc (7)	10	11	51	10
	rcr (1)^2	1	1	6	1
	rcr (2)^2	(2)	(2)	159	163
	rcr (3)^2	6	6	–	6
	rcr (5)	13	13	–	13
	rcr (7)	23	29	1132	23

Table 1. Median synthesis time in seconds from three runs. "TTS" column reports accumulated time to find the first optimal solution from Spaces 1–3. The parenthesized value indicates time to prove that no solution exists in that space. Double-parenthesized value indicates time to find a solution, but the solution is not optimal when considering candidates in larger space. "–" indicates timeout (> 30 minutes).

Benchmark	Version	Mean Synthesis Time (s)			
	(Struct Size)	SW	Naive 1	Naive 2	No Acc
Mult-32	shmem	1	1	11	3
	reg	1	2	11	3
AoS-Sum	crc (1)^2	1	1	6	1
	crc (2)^2	6	(3)	(680)	18
	crc (3)^2	1	(9)	637	–
	crc (5)	3	(15)	–	749
	crc (7)	4	(24)	–	–

Table 2. Comparison of different synthesizer versions. Table reports mean synthesis time in seconds from three runs. "SW" column reports "TTS" column from Table 1. "Naive 1" and "Naive 2" use the naive transformation index swizzle template with depth 1 and 2 respectively. "No Acc" synthesizes without accumulators (without canonicalization). The parenthesized value indicates time to prove that no solution exists within this depth. "–" indicates timeout (> 30 minutes).

Matrix dimensions $m \times n$	Swizzle Compositions for Step 1–3
$m = n$	1. $\mathrm{rot}(m)(k = j)(i)$ 2. $\mathrm{rot}(n)(k = i)(j)$ 3. i
$\gcd(m, n) = 1$	1. $(\mathrm{rot}(m)(k = j) \circ \mathrm{fan}(m))(i)$ 2. $(\mathrm{rot}(n)(k = i) \circ \mathrm{fan}(n))(j)$ 3. i
$m \neq n \wedge n\%m = 0$	1. $(\mathrm{rot}(m)(k = j) \circ \mathrm{fan}(m))(i)$ 2. $(\mathrm{rot}(n)(k = i) \circ \mathrm{fan}(n))(j)$ 3. i
$m \neq n \wedge m\%n = 0$	1. $(\mathrm{rot}(m)(k = j) \circ \mathrm{fan}(m) \circ\ \mathrm{rot}(m)(k = j))(i)$ 2. $\mathrm{rot}(n)(k = i)(j)$ 3. $\mathrm{rot}(m)(k = j)(i)$
$1 < \gcd(m, n) < m, n$	1. Either $(\mathrm{fan} \circ \mathrm{fan} \circ \mathrm{rot})(i)$ or $(\mathrm{group}(gs, \mathrm{fan}) \circ \mathrm{fan} \circ \mathrm{fan} \circ \mathrm{rot})(i)$ 2. $(\mathrm{cm2linear}(m, n)(k = j) \circ \mathrm{rot}(m)(k = j))(i)$ 3. $\mathrm{rot}(m)(k = j)(i)$

Table 3. The compositions of swizzle primitives that achieve the same effect as Catanzaro *et al.*'s column-row-column shuffles, when m and $n \in [1, 64]$.

template in Equation (5) with the naive version in Equation (1). Table 2 compares the synthesis time of our original synthesizer (Swizzle Inventor) with the modified version that uses the naive template with depth 1 (Naive 1) and depth 2 (Naive 2). The table shows that the synthesizer that uses the naive template is much slower than our original version.

Comparison to No Canonicalization Another key to our fast synthesis is canonicalization using accumulators. In this experiment, we synthesized the same benchmarks without canonicalization, by letting the SMT solver reason about actual algebraic operations directly. As a result, to verify that a matrix program satisfies a specification, we have to check that the post-condition holds on all concrete input values (requiring for-all quantifier in SMT formulas). Table 2 shows that synthesizing without accumulators (No Acc) exceeds 30 minutes on many easy benchmarks. Therefore, our accumulator encoding drastically reduces the synthesis time.

8 Further Analysis of Swizzle Template

The transformation swizzle template presented in this paper covers many permutations but not all. We discover that we cannot synthesize matrix transposition (AoS-load-store) for some larger struct sizes. Therefore, we run an additional analysis to characterize the expressiveness of our template for the matrix transposition sketch in Listing 4, for all m and $n \in [1, 64]$. We summarize our findings in Table 3. For each case, we describe compositions of our swizzle primitives that achieve the same effect as Catanzaro *et al.* Swizzle Inventor can synthesize solutions for the first three cases as the compositions are covered by $?sw_{xform}$, defined in Equation (5).

New Swizzle Primitives However, we find that the solution for Step 2 of the last case cannot be expressed by any combination of our three primitives. We observe that it requires *column-major linearization*:

$$cm2linear(n, m)(k)(i) = (k * n + i) \bmod m \qquad (8)$$

This primitive is handy for switching primary and secondary indices. Observe in Table 3 that typically we apply swizzle primitives on the primary index (i for Step 1 and 3, and j for Step 2); except in Step 2 of the last case, we apply *rot* on the secondary index i, followed by *cm2linear*.

Another kind of permutations that our current template does not cover is moving u consecutive elements together as a unit when u divides n. To cover this kind of permutations, we can introduce a *unit* primitive:

$$unit(n, u, p)(i) = p(\lfloor n/u \rfloor)(\lfloor i/u \rfloor) * u + (i \bmod u) \quad (9)$$

Similar to *group*, it takes a permutation p as an argument.

Template Extension To cover more swizzles, we can generalize our template to compose the swizzle primitives in any arbitrary order with limited depth to define a finite search space. Unfortunately, our synthesizer times out with this new template because there are more unknown constants to synthesize. We leave the problem of scaling up synthesis for future work.

9 Related Work

Register Caching and Intra-Warp Shuffles Swizzle Inventor can help implement many algorithms that exploit registers and intra-warp shuffles to accelerate GPU kernels, including matrix transposition [7], finite field multiplication [3], SHA-3 [18], sorting algorithms [13], and sequence alignment [29]. Similar in goals to Swizzle Inventor, Hou *et al.* propose a framework, called GPU-UniCache, to automatically generate code that accesses data buffered in registers and shared memory [14]. Unlike Swizzle Inventor, GPU-UniCache is limited to stencil computations. GPU-UniCache also requires more intra-warp data exchanges to fetch the desired value from a register cache than does our approach.

Program Synthesis Program synthesis has been used to synthesize parallel programs with swizzles. Barthe *et al.* [1] synthesize SIMD code (including SSE shuffle instructions). BitStream [25] can synthesize permutations for CPU cryptography primitives. SynthCL [27] can verify and synthesize OpenCL programs. Sketching Stencils [24] synthesizes high-performance stencil implementations for CPUs. Raabe and Bodik [23] adapt sketch-based synthesis to hardware design. MSL [31] is a sketching language for developing SPMD programs, with a focus on bulk-synchronous parallelism. These frameworks can be used to synthesize swizzles. Some of these systems synthesize only constant mappings [1, 23, 25]. Others can (potentially) synthesize swizzles that are functions of time and space (iterations and thread identifiers), but do not address complex swizzling expressions needed for advanced GPU kernels [24, 27, 31]. Our canonicalization encoding is similar to techniques used in some prior program synthesizers [5, 20, 23].

Polyhedral Model Swizzle Inventor is complementary to polyhedral compilation [17, 19, 22, 30]. First, our framework does not attempt to automatically find a combination of loop optimizations created by developers of a polyhedral compiler. Instead, we let programmers sketch target programs, so new optimizations can be invented by the programmers even if such optimizations cannot be expressed as a composition of existing loop transformations. Second, Swizzle Inventor can synthesize mappings that are more flexible than those of polyhedral techniques because we allow non-affine mappings and a more permissive correctness condition (equality of values rather than preservation of dependencies among statements). These advantages come at the cost of restricting ourselves to bounded kernels. In contrast, polyhedral compilers transform loops with affine bounds and index accesses.

Here are a few samples of polyhedral techniques focusing on generating efficient GPU kernels. Baskaran *et al.* [2] design an automatic source-to-source polyhedral compiler for GPUs, performing several GPU-specific optimizations including coalescing global memory accesses; avoiding shared memory bank conflicts; and optimizing copies from global to shared memory. PPCG [28] generates CUDA code for a given static control loop nest. Similar to [2], PPCG uses a polyhedral model to select schedules that generate a maximum number of parallel dimensions for each statement, and maps these parallel dimensions to CUDA threads. Holewinski *et al.* [12] generate CUDA code for stencils using overlapped tiling.

10 Conclusion

Swizzle Inventor is a framework that helps developers implement swizzle kernels. Developers write a program sketch that expresses their high-level algorithm but leaves out the data access patterns that determine how threads collaboratively access data and/or how computations are mapped onto loop iterations. The framework then synthesizes index expressions in the program such that the program behaves correctly with respect to a specification. We demonstrated that Swizzle Inventor can help create kernels that are faster than manually-written CUDA kernels.

Acknowledgments

This work is supported in part by the NSF Grant ACI OAC–1535191; by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF-1723352); the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01; grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032; gifts from Google, Intel, Mozilla, Nokia, and Qualcomm; as well as hardware donations from NVIDIA. We thank the authors of Ben-Sasson *et al.* [3] for helping us to reproduce their results, and Uday Bondhugula and Albert Cohen for a discussion of polyhedral compilation.

References

[1] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. 2013. From relational verification to SIMD loop synthesis. In *Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2442516.2442529

[2] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/1375527.1375562

[3] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. 2016. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/2925426.2926259

[4] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837666

[5] Eric Butler, Emina Torlak, and Zoran Popović. 2018. A Framework for Computer-Aided Design of Educational Domain Models. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Isil Dillig and Jens Palsberg (Eds.). https://doi.org/10.1007/978-3-319-73721-8_7

[6] Bryan Catanzaro. 2018. Trove. https://github.com/bryancatanzaro/trove.

[7] Bryan Catanzaro, Alexander Keller, and Michael Garland. 2014. A Decomposition for In-place Matrix Transposition. In *Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/2555243.2555253

[8] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442

[9] Jia Guo, Ganesh Bikshandi, Daniel Hoeflinger, Gheorghe Almási, Basilio B. Fraguela, María Jesús Garzarán, David A. Padua, and Christoph von Praun. 2006. Hierarchically tiled arrays for parallelism and locality. In *International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2006.1639573

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/3168824

[11] Matan Hamilis. 2018. https://github.com/HamilM/GpuBinFieldMult.

[12] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance Code Generation for Stencil Computations on GPU Architectures. In *International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/2304576.2304619

[13] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. 2017. Fast segmented sort on GPUs. In *International Conference on Supercomputing, (ICS)*. https://doi.org/10.1145/3079079.3079105

[14] Kaixi Hou, Hao Wang, and Wu-chun Feng. 2017. GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs. In *Computing Frontiers Conference (CF)*. https://doi.org/10.1145/3075564.3075583

[15] Forrest Iandola. 2018. https://github.com/forresti/convolution.

[16] Forrest N. Iandola, David Sheffield, Michael J. Anderson, Phitchaya Mangpo Phothilimthana, and Kurt Keutzer. 2013. Communication-minimizing 2D convolution in GPU registers. In *International Conference on Image Processing (ICIP)*. https://doi.org/10.1109/ICIP.2013.6738436

[17] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega Calculator and Library, Version 1.1.0. (1996). http://www.cs.utah.edu/~mhall/cs6963s09/lectures/omega.ps

[18] Wai-Kong Lee, Xian-Fu Wong, Bok-Min Goi, and Raphael C.-W. Phan. 2017. CUDA-SSL: SSL/TLS accelerated by GPU. In *International Carnahan Conference on Security Technology (ICCST)*. https://doi.org/10.1109/CCST.2017.8167848

[19] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *International Conference on Supercomputing (ICS)*. http://doi.acm.org/10.1145/305138.305197

[20] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast Synthesis of Fast Collections. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2908080.2908122

[21] NVIDIA. 2019. NVIDIA Performance Primitives. https://developer.nvidia.com/npp. Accessed 15 January 2019.

[22] William Pugh. 1991. Uniform Techniques for Loop Optimization. In *International Conference on Supercomputing (ICS)*.

[23] Andreas Raabe and Rastislav Bodik. 2009. Synthesizing Hardware from Sketches. In *Annual Design Automation Conference (DAC)*. ACM, New York, NY, USA, 623–624. https://doi.org/10.1145/1629911.1630074

[24] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1250734.1250754

[25] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-streaming Programs. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1065010.1065045

[26] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Symp. on New Ideas in Programming and Reflections on Software (Onward!)*.

[27] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2594291.2594340

[28] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions Architecture and Code Optimization (TACO)* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[29] Jie Wang, Xinfeng Xie, and Jason Cong. 2017. Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms. In *International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2017.79

[30] Doran K. Wilde. 1993. *A Library for Doing Polyhedral Operations*. Technical Report 785. IRISA.

[31] Zhilei Xu, Shoaib Kamil, and Armando Solar-Lezama. 2014. MSL: A Synthesis Enabled Language for Distributed Implementations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/SC.2014.31

[32] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. 2015. ArrayFire: A High Performance Software Library for Parallel Computing with an Easy-To-Use API. https://github.com/arrayfire/arrayfire.