Vector Instruction Selection for Digital Signal Processors using Program Synthesis

Maaz Bin Safeer Ahmad
Adobe
Seattle, USA
mahmad@adobe.com

Alexander J. Root
MIT CSAIL
Cambridge, USA
ajroot@csail.mit.edu

Andrew Adams
Adobe
San Jose, USA
anadams@adobe.com

Shoaib Kamil
Adobe
New York, USA
kamil@adobe.com

Alvin Cheung
University of California, Berkeley
Berkeley, USA
akcheung@cs.berkeley.edu

ABSTRACT

Instruction selection, whereby input code represented in an intermediate representation is translated into executable instructions from the target platform, is often the most target-dependent component in optimizing compilers. Current approaches include pattern matching, which is brittle and tedious to design, or search-based methods, which are limited by scalability of the search algorithm. In this paper, we propose a new algorithm that first abstracts the target platform instructions into high-level uber-instructions, with each uber-instruction unifying multiple concrete instructions from the target platform. Program synthesis is used to lift input code sequences into semantically equivalent sequences of uber-instructions and then to lower from uber-instructions to machine code. Using 21 real-world benchmarks, we show that our synthesis-based instruction selection algorithm can generate instruction sequences for a hardware target, with the synthesized code performing up to 2.1× faster as compared to code generated by a professionally-developed optimizing compiler for the same platform.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; *Compilers*; *Software performance.*

KEYWORDS

Instruction selection, program synthesis, compiler optimizations

ACM Reference Format:
Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 2022. Vector Instruction Selection for Digital Signal Processors using Program Synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507714

1 INTRODUCTION

We have witnessed the rise of hardware accelerators across different application domains. These accelerators, such as the Qualcomm Hexagon DSP [10] that is now found on-die in millions of Android mobile phones, offer domain-specific optimization for applications, such as performance improvement and energy savings as compared to general processors. However, to fully utilize such accelerators, applications must either be written against libraries or exotic instruction intrinsics provided by the accelerator, or in a domain-specific language (DSL) altogether.

While mapping computations from programmer expressions into a sequence of processor instructions can be formulated as *instruction selection* as part of program compilation, choosing the optimal sequence of instructions for a given computation is especially difficult when considering vector instructions, which enable fine-grained parallel computation. Modern vector instruction sets, such as Intel's AVX-512 and VNNI, ARM's Advanced SIMD, or Hexagon's HVX, offer a rich set of complex vector instructions. These include lane-parallel vector instructions such as single-instruction multiple-data (SIMD) instructions, non-isomorphic instructions that apply different operations to different lanes of the input vector in parallel, cross-lane vector instructions that implement reductions such as dot-products, and sliding window instructions that use intersecting sets of input vector lanes to compute the output values in parallel.

Fully exploiting these instructions is difficult for compilers. Most approaches (such as LLVM [16] and Halide [22]) utilize some variant of pattern-matching rewrites, which transform templatized sequences of operations into hardware-specific vector instructions. Indeed, Halide utilizes its own matching machinery, consisting of ad-hoc patterns and rewrites built by programmers experienced in writing code for each specific hardware backend, due to LLVM's inability to fully exploit complex vector instruction sets such as Hexagon's HVX. While work has been done to enhance LLVM's library of patterns [8, 23], matching-based approaches in general suffer from the limitations of the underlying greedy algorithm to match code patterns and thus can miss rewriting opportunities. Meanwhile, prior work that formulates instruction selection as dynamic programming [15] or exhaustively enumerates instruction sequences up to a fixed length guided by a cost model [17, 24] struggle to scale to large input instruction sequences, or require complex cost models to make the search efficient.

In this paper, we describe Rake, a system that leverages *program synthesis* to perform instruction selection for vectorized Halide expressions. Unlike prior work, Rake does not rely on manually crafted code patterns to match on the input code. Given an input code sequence represented in the Halide IR, Rake instead synthesizes a sequence of target platform instructions that is provably semantically equivalent to the input. Rake first uses synthesis to *lift* the input code sequence into an intermediate representation (IR). This IR, called Uber-Instruction IR, is a high-level abstracted version of the target instruction set. Once lifted, Rake then lowers the Uber-Instruction IR into the concrete syntax of the target instruction set using synthesis. Lowering is done *incrementally* by first synthesizing a combination of hardware intrinsics that performs the computation while ignoring any data movement operations (a *swizzle-free sketch*).

Once the swizzle-free sketch is synthesized, the data movement (i.e., swizzle) instructions are concretized via another synthesis query, and the finished sequence of instructions is grafted back into the Halide IR. Rake makes it possible to integrate backend-specific rewrites into a domain-specific compiler without explicitly specifying transformation rules or their priority order. Instead, compiler developers only need to specify the semantics of the target instructions, modifying the Uber-Instruction IR if necessary, and Rake will then synthesize target instruction sequences automatically. As our experiments show, Rake's synthesis-driven "lift-then-lower" approach makes instruction selection scalable, without compromising the quality of the generated code.

To summarize, our key contributions are:

- A methodology that decomposes instruction selection for mixed vector/scalar IR into a series of tractable program synthesis queries, by searching for high-level Uber-Instructions, then lowering to sketches without data movement operations, and finally synthesizing the required data movement.
- An implementation of our methodology within the Halide DSL compiler and evaluation using real-world benchmarks.[1]

We evaluate Rake by using it to generate code for the HVX accelerator and show that Rake-generated code can produce speedups of up to 2.1× over that generated by the existing Halide and LLVM HVX pipeline (which has been developed and tuned by Qualcomm, Google, and LLVM developers), as Rake finds instruction sequences that are not considered by the existing Halide pattern-matching rules and the instruction selection pass in LLVM.

Next, we give background on the Halide domain-specific compiler and describe how Rake works in three phases: lifting to the Uber-Instruction IR (§3), swizzle-free sketch synthesis (§4), and swizzle synthesis (§5). Then, in §7 we show the efficacy of Rake using 21 real-world benchmarks.

2 BACKGROUND & OVERVIEW

In this section, we give an overview of the entire compilation process illustrated in Figure 1, while providing background on the methodologies underlying Rake's synthesis-based instruction selection algorithm.

[1]Source code available at https://github.com/uwplse/rake/

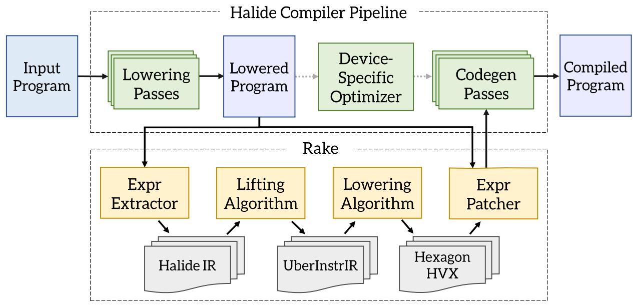

Figure 1: Compilation Overview. Rake intercepts Halide's compilation pipeline to synthesize device-specific implementations for vector expressions.

We implement Rake within Halide [22], a domain-specific language for computations on images and dense tensors. Halide enables programmers to concisely describe the algorithm they wish to implement separately from details of how that algorithm should be executed. This separation between *algorithm* and *schedule* allows the compiler to generate optimized code for CPUs, GPUs, and DSPs from the same algorithm specification. Halide enjoys wide adoption in industry, including usage by Google, Adobe, and others [21].

2.1 Motivating Example

The Sobel filter [20] is a well-known algorithm used in image processing and computer vision for approximating the gradient of an image intensity function. This approximation is particularly useful in edge-detection algorithms.

Figure 2 shows an implementation of the Sobel filter[2] expressed in Halide. To compile this algorithm for a given architecture, Halide requires a schedule that describes high-level device-specific optimizations, such as how to tile and vectorize the loops in the output program. Lines 18 to 21 specify a Halide schedule for compiling the Sobel filter to the Hexagon DSP architecture [10]. The schedule directs Halide to offload the computation to Hexagon, prefetch data into the cache two iterations before it is needed, and compute the output in tiles of 4×128 elements. The schedule also directs Halide to vectorize the inner x-loop. Since the schedule specifies no directives for any of the intermediate results, such as `sobel_x` or `sobel_y`, by default Halide will inline their computation. Thus, after all scheduling is applied, the lowered program in Halide IR is a single tiled loop-nest, where the body of the inner-most loop computes a 1×128 element tile of the output using a target-independent vector-expression, shown in Figure 3.

Once the program is lowered into Halide's IR, the next step is code-generation. Currently, Halide relies on two mechanisms to map Halide's IR to machine instructions: for most operations, Halide uses LLVM's built-in vector types and operations, but in addition, the Halide compiler uses a pattern-matching pass (the Device-Specific Optimizer in Figure 1) to rewrite sequences of Halide IR operations into calls to LLVM backend-specific intrinsics. This transformation is essential for obtaining performance on architectures

[2]This implementation obtained from the Halide repository is a modified version that does not take the square root of the gradient.

```
void sobel3x3(Buffer<uint8_t> input, Buffer<uint8_t> output) {
  Var x, y, xi, yi;
  Func in16, x_avg, y_avg, sobel_x, sobel_y;

  // The Algorithm
  in16(x, y) = cast<uint16_t>(input(x, y));

  x_avg(x, y) = in16(x-1, y) + 2 * in16(x, y) + in16(x+1, y);
  sobel_x(x, y) = absd(x_avg(x, y-1), x_avg(x, y+1));

  y_avg(x, y) = in16(x, y-1) + 2 * in16(x, y) + in16(x, y+1);
  sobel_y(x, y) = absd(y_avg(x-1, y), y_avg(x+1, y));

  output(x, y) = cast<uint8_t>(clamp(sobel_x(x, y)
      + sobel_y(x, y), 0, 255));

  // The schedule
  output.hexagon()
      .prefetch(input, y, 2)
      .tile(x, y, xi, yi, 128, 4)
      .vectorize(xi);
}
```

Figure 2: The Sobel Filter expressed in Halide.

```
// Syntax guide:
// - uint8x128(...) and uint16x128(...) are vector casts
// - x128(c) broadcasts scalar c to a 128-lane vector
// - input(x,y) denotes a 1024-bit vector load from location (x,y)
// - min, max, absd (absolute difference), + and * are vector ops

uint8x128(
  max(
    min(
      absd(
        uint16x128(input(x - 1, y - 1)) +
          uint16x128(input(x, y - 1)) * x128(2) +
          uint16x128(input(x + 1, y - 1)),
        uint16x128(input(x - 1, y + 1)) +
          uint16x128(input(x, y + 1)) * x128(2) +
          uint16x128(input(x + 1,  y + 1))
      )
      +
      absd(
        uint16x128(input(x - 1, y - 1)) +
          uint16x128(input(x - 1, y)) * x128(2) +
          uint16x128(input(x - 1, y + 1)),
        uint16x128(input(x + 1, y - 1)) +
          uint16x128(input(x + 1, y)) * x128(2) +
          uint16x128(input(x + 1, y + 1)),
      ),
    x128(0)),
  x128(255)))
```

Figure 3: Target-independent Halide IR vector expression produced by the Sobel filter algorithm.

like Hexagon, where LLVM fails to automatically discover mappings from generic LLVM IR to semantically-rich backend vector instructions such as those in the HVX ISA.

While Halide's pattern-matching approach is fast, it is also brittle. If a program fragment could map to an HVX-specific instruction, but doesn't conform to the pre-written patterns, Halide will use lower-performing generic vector instructions. This leaves valuable performance on the table. Figure 4 highlights three instances in the Sobel filter benchmark where Halide's pattern-matching approach fails to discover the best instruction sequence. In (a), Halide uses the more general `vmpa` (sum of two widening multiplies) and `vadd` instructions to implement the 3-point horizontal convolution. This computation can be implemented more efficiently as a single `vtmpy` instruction (sliding-window-sum of two widening multiplies with an additional accumulation). In (b), although the `vtmpy` instruction is not applicable as this expression does not implement a sliding-window reduction, we can replace the `vmpa` and `vadd` instructions with a single `vmpa.acc` instruction (a variant of `vmpa` that accumulates into the target register). Finally, in expression (c), Halide fails to infer that the *min* and *cast* operations on an unsigned input can be replaced by a single *saturate* operation. In contrast, Rake can discover all three optimizations without the need for any re-write rules, resulting in a 27% runtime performance improvement over Halide's existing optimizer.

2.2 Instruction Selection using Rake

As illustrated in Figure 1, Rake intercepts Halide's compilation pipeline after the input program has been lowered to Halide's IR and all scheduling optimizations, including vectorization, have been applied. Rake then extracts the set of vectorized expressions found in the lowered program and uses program synthesis to discover mappings from Halide IR to backend-specific intrinsics using program synthesis. Rake decomposes the instruction selection problem into multi-step program synthesis by first lifting the input expressions to high-level Uber-Instructions before lowering. Rake lowers uber-instruction sequences into executable instructions by synthesizing the computational instructions followed by data movement guided by a simple, explainable cost model. Together, these strategies enable Rake to scale up to large loop bodies, rather than the scale of a few instructions as in prior superoptimizers [19, 24]. Finally, Rake patches the lowered program, replacing target-independent Halide IR vector expressions with optimized target-aware instruction sequences.

In the remainder of this section, we first provide a primer on program synthesis and then explain Rake's instruction selection algorithm.

2.2.1 Program Synthesis & Verification. Rake relies on syntax-guided program synthesis (SyGuS) [4] to map expressions from Halide IR to the target ISA. Syntax-guided synthesis is a search-based technique for constructing programs. As input, it takes a set of semantic constraints known as the *specification* as well as a set of syntactic constraints, which we refer to as the *grammar*. As output, it generates programs or expressions from the grammar that satisfy the semantic constraints. We can formulate the instruction-selection problem as a SyGuS problem as follows:

$$\exists\, e_t \in \mathcal{G}.\quad interpret(e_t) = interpret(e_o)$$

The synthesizer must find an expression e_t in the target ISA expressed using the grammar $\mathcal{G}$, such that the output of e_t is equivalent to the output of the original Halide IR expression e_o. In Rake, the search for equivalent expressions is performed via inductive program synthesis [4, 26, 27], which utilizes satisfiability modulo theories (SMT) to guide the search. In this technique, a synthesizer queries the SMT solver to find a candidate equivalent program that holds over a small number of example input/output pairs. Then, the SMT engine is re-queried to either prove the equivalence holds over all inputs, or produce a counterexample input, in which case this input is added to input-output pairs and the SMT solver is once

	Halide IR Expr	Halide Codegen (HVX)	Rake Codegen (HVX)

(a)	uint16x128(input(x - 1, y + 1)) +
	uint16x128(input(x, y + 1)) * x128(2) +
	uint16x128(input(x + 1, y + 1))
		/* Latency: 4, Loads: 3 */
		vmpa(// 2-multiply-add
		  input(x, y + 1),
		  input(x + 1, y + 1),
		  0x2, 0x1) +
		vzxt(input(x - 1, y + 1))
			/* Latency: 2, Loads: 2 */
			vtmpy(// Sliding window 3-point reduction
			  input(x - 1, y + 1),
			  input(x, y + 1),
			  0x1, 0x2)

(b)	uint16x128(input(x - 1, y - 1)) +
	uint16x128(input(x - 1, y)) * x128(2) +
	uint16x128(input(x - 1, y + 1))
		/* Latency: 4 */
		vmpa(// 2-multiply-add
		  input(x - 1, y),
		  input(x - 1, y + 1),
		  0x2, 0x1) +
		vzxt(input(x - 1, y - 1)))
			/* Latency: 3 */
			vmpa.acc(// 2-multiply-add accumulate
			  vzxt(input(x - 1, y - 1)),
			  input(x - 1, y),
			  input(x - 1, y + 1),
			  0x2, 0x1)

(c)	uint8x128(
	  max(
	    min(
	      absd(...) +
	      absd(...),
	      x128(0)),
	    x128(255)))
		/* Latency: 11 */
		vshuffeb(// Extract lower byte
		  vmax(
		    vabsdiff(...) + vabsdiff(...),
		    vsplat(255)),
		  vmax(
		    vabsdiff(...) + vabsdiff(...),
		    vsplat(255)))
			/* Latency: 9 */
			vsat(// Saturate
			  vabsdiff(...) + vabsdiff(...),
			  vabsdiff(...) + vabsdiff(...))

Figure 4: An illustration of key differences in the HVX code generated by Halide's current optimizer and Rake for the Sobel filter. We do not count broadcasts of loop-invaraint expressions towards latency, as they will be moved outside the loop by LLVM.

```
(narrow
  (vs-mpy-add
    '((abs-diff
       (vs-mpy-add (load-data) '(2 1 1) #f uint16)
       (vs-mpy-add (load-data) '(2 1 1) #f uint16))
      (abs-diff
       (vs-mpy-add (load-data) '(2 1 1) #f uint16)
       (vs-mpy-add (load-data) '(2 1 1) #f uint16)))
    '(1 1) #f uint16)
  #t #f uint8)
```

Figure 5: The Sobel filter expression lifted to HVX uber-instructions.

```
(define (narrow vec saturate? round? outT)
  (let a (if round? (round vec) vec))
  (if saturate? (sat_cast<outT> a) (cast<outT> a)))

(define (abs-diff vec0 vec1)
  (- (max vec0 vec1) (min vec0 vec1)))

(define (vs-mpy-add vec weights saturate? outT)
  (let a (if saturate? (cast<int64> vec) (cast<outT> vec)))
  (let b (convolve a weights))
  (if saturate? (sat_cast<outT> b) b))
```

Figure 6: A sample of HVX uber-instructions.

again re-queried, continuing the search. In order to prove observational equivalence between expressions, SMT engines require an interpreter for the target ISA instructions.

The primary hurdle to using program synthesis for instruction selection is scalability. To effectively translate large expressions, often requiring dozens of vector-intrinsics to implement, Rake breaks down the instruction selection process into three stages.

2.2.2 *Lifting to Uber-Instruction IR.*

Rake begins the instruction selection process by lifting the input Halide IR expressions (such as the one shown in Figure 3) into a target-specific IR of uber-instructions, which we call the Uber-Instruction IR.

The Uber-Instruction IR is a condensed version of the target ISA, where each uber-instruction unifies a set of related intrinsics in the target ISA by implementing the common higher-level compute pattern. For example, Figure 5 shows the expression encountered in the Sobel filter (Figure 3) expressed using the uber-instructions derived from the HVX ISA. The `vs-mpy-add` uber-instruction unifies all available HVX intrinsics that implement vector-scalar multiply-add patterns, such as `vadd` (addition), `vmpy` (widening multiply) or `vmpa` (sum of two widening multiples). Similarly, the set of HVX intrinsics that downcast integer values in the input vector to a narrower integer type are consolidated into an uber-instruction called `narrow`. Figure 6 shows pseudo-code describing the semantics of HVX uber-instructions that appear in Figure 5.

The translation from Halide IR to Uber-Instruction IR is performed using a bottom-up enumerative synthesis algorithm (described in §3) that *lifts* the lower-level Halide IR to the higher-level Uber-Instruction IR. The lifting algorithm attempts to rewrite the input expressions using the fewest number of uber-instructions possible. This, in effect, clusters operations in the input expression that implement a single high-level compute pattern by rewriting them into an uber-instruction. Rake then synthesizes target ISA implementations for the lifted expression by lowering the sequence of uber-instructions into a sequence of target ISA instructions. Lifting expressions to Uber-Instruction IR makes synthesis-based instruction selection scalable in two ways. First, it breaks the synthesis problem down to easier sub-problems since larger expressions require many uber-instructions to implement, and the algorithm builds the sequence of uber-instructions bottom-up. Second, for each uber-instruction only a subset of the target ISA is relevant, so we can specialize the grammar to just those instructions.

2.2.3 *Lowering to the Target ISA.*

In the second stage of instruction selection, Rake uses a recursive backtracking algorithm, listed in Algorithm 2, to lower the lifted expressions to the target ISA. Given

```
(??swizzle
  (vtmpy (??load [vec-pair? #t]), 1, 2)
  [vec-pair? #t])
```

Figure 7: A swizzle-free sketch of an HVX expression. Data-movement is abstracted away using `??load` and `??swizzle`.

```
(let [x (vtmpy
          (vcombine
            (vread (- x 1) (- y 1))
            (vread (+ x 1) (- y 1)))
          1, 2)]
    (vshuffvdd (hi x) (lo x) -2))
```

Figure 8: Synthesized data movement replaces `??load` and `??swizzle` to yield complete HVX implementations.

an expression e in the Uber-Instruction IR, Rake first recursively lowers each sub-expression to the the target ISA (Lines 5 to 7). Then, Rake uses the lowered sub-expressions $\mathcal{S}$ as building-blocks to synthesize $\mathcal{I}$, the lowered implementation for e. We explain Rake's lowering algorithm in more detail in §4 and §5.

Despite the incremental approach outlined above, synthesizing efficient vector implementations remains challenging. The best implementations often involve an interweaving of data-movement (i.e., loading/storing data across memory hierarchies or re-arrangement of vector lanes into different permutations) and computation (operations that produce new values) to exploit intrinsics that offer the greatest throughput. Directly synthesizing such implementations is expensive due to the sheer number of candidates that can be enumerated. Therefore, Rake first synthesizes a swizzle-free sketch τ of the output expression. A swizzle-free sketch is a partial implementation of the input expression in the target ISA that specifies the computation concretely using intrinsics from the target ISA but abstracts away the necessary data movement using special placeholder terms. For example, consider the lifted multiply-add expression from line 4 of Figure 5. Figure 7 shows a valid swizzle-free sketch for this expression. The computations performed by the sketch (multiplications, additions and widening-casts) are implemented using intrinsics from the HVX ISA, such as `vtmpy` (3-point fused widening-multiply-add), but the loading and swizzling of data are expressed abstractly using special constructs `??load` and `??swizzle`. In §4, we define the semantics of the constructs used by Rake to represent data movement in swizzle-free sketches and explain how Rake uses them to verify partial implementations for correctness during synthesis. At a high level, these allow instructions in a swizzle-free sketch to load data from any memory location or vector lane in a register without needing to reason about the cost or sequence of data movement operations required.

2.2.4 Synthesizing Data Movement. Once a valid swizzle-free sketch is synthesized, Rake attempts to complete the implementation by synthesizing the missing data movement. Figure 8 shows a complete implementation synthesized using the sketch from Figure 7. Each `??load` and `??swizzle` term has been replaced by a sequence of vector-reads and HVX shuffling instructions (such as `vcombine` and `vshuffvdd`) to yield a fully lowered HVX expression.

3 DYNAMIC EXPRESSION DECOMPOSITION

Synthesizing low-level device-specific expressions from IR code is an expensive task. For example, the relatively simple IR expression

```
Input: An expression in the Halide IR
Output: An equivalent expression in the Uber-Instruction IR
1 Function Lift($e$)
2     $\mathcal{S} \leftarrow \{$ Lift($se$) $\mid se \in$ Subexprs($e$) $\}$
3     for $s \in \mathcal{S}$ do
4         if $\ell \leftarrow$ UpdateInstr($s, e$) $\neq$ unsat then return $\ell$
5     for $s \in \mathcal{S}$ do
6         if $\ell \leftarrow$ ReplaceInstr($s, e$) $\neq$ unsat then return $\ell$
7     return ExtendExpr($\mathcal{S}, e$)
```

Algorithm 1: Lifting expressions from Halide IR to the Uber-Instruction IR.

found in the Sobel filter (Figure 3) requires a sequence of 38 HVX intrinsics to implement. On top of that, the HVX ISA offers hundreds of intrinsics to choose from at every step when building the sequence. Prior work attempted to scale synthesis to large instruction sets by constraining the length of the input instruction sequence considered at each step [5, 19]. Other prior work attempted to scale the input instruction sequence length by carefully extracting a directed acyclic graph (DAG) of operations to consider with a single output [23] while restricting the scope of the target instructions to middle-end IR rather than concrete instructions. While these strategies are effective, they have mostly been applied to scalar code (or, in the case of [23], directly to LLVM IR); vectorized code introduces not just a larger variety of potential instructions but additional complications due to potentially needing swizzling between operations. The goal of Rake is to scale synthesis to input sequences sufficiently large enough to optimize real-world code, as well as scaling reasoning to handle a large, complex vector instruction set.

3.1 Lifting to Uber-Instruction IR

In modern ISAs, the number of high-level compute patterns implemented is typically much smaller than the number of intrinsics offered; many intrinsics can be viewed as specializations of a more general compute pattern. An uber-instruction is a function that implements one such high-level compute pattern and therefore consolidates the semantics of many related intrinsics in the target ISA. The Uber-Instruction IR is simply a collection of all uber-instructions manually derived from the target ISA. To make synthesis scalable, Rake re-writes large input expressions as a sequence of the derived uber-instructions, before lowering each uber-instruction incrementally. In §6, we discuss the key design concerns when designing the Uber-Instruction IR for a given target ISA, and how we derived the set of uber-instructions for the HVX ISA.

3.2 Lifting Algorithm

Rake uses a bottom-up enumerative search algorithm, listed as Algorithm 1, to lift expressions from the Halide IR to the Uber-Instruction IR. Given a Halide IR expression e, Rake first recursively lifts each sub-expression of e to the Uber-Instruction IR (Line 2). Then, Rake applies a sequence of *update*, *replace* and *extend* steps to the set of lifted sub-expressions ($\mathcal{S}$) to construct the lifted representation of e.

3.2.1 Update. In the update step, Rake tries to lift the input expression by updating the inputs to an instruction in one of the lifted sub-expressions. Suppose e is the input expression in Halide IR and

Step	Halide Expr	Lifted Sub-Exprs	Rule	Lifted Expr
1	`cast<uint16_t>(input(x-1, y-1))`	∅	Extend	`(widen (load-data) int16)`
2	`cast<uint16_t>(input(x, y-1))`	∅	Extend	`(widen (load-data) int16)`
3	`2`	∅	Extend	`(broadcast 2)`
4	`cast<uint16_t>(input(x+1, y-1))`	∅	Extend	`(widen (load-data) int16)`

5	`cast<uint16_t>(input(x, y-1)) * 2`	`[(widen (load-data) int16),`
		`broadcast(2)]`
			Replace	`(vs-mpy-add (load-data)`
				`[kernel: '(2)]`
				`[saturating: #f]`
				`[output-type: int16])`

6	`cast<uint16_t>(input(x-1, y-1)) +`
	`cast<uint16_t>(input(x, y-1)) * 2`
		`[(widen (load-data) int16),`
		`(vs-mpy-add (load-data)`
		`[kernel: '(2)] [saturating: #f]`
		`[output-type: int16])]`
			Update	`(vs-mpy-add (load-data)`
				`[kernel: '(2 1)]`
				`[saturating: #f]`
				`[output-type: int16])`

7	`cast<uint16_t>(input(x-1, y-1)) +`
	`cast<uint16_t>(input(x, y-1)) * 2 +`
	`cast<uint16_t>(input(x+1, y-1))`
		`[(widen (load-data) int16),`
		`(vs-mpy-add (load-data)`
		`[kernel: '(2 1)] [saturating: #f]`
		`[output-type: int16])]`
			Update	`(vs-mpy-add (load-data)`
				`[kernel: '(2 1 1)]`
				`[saturating: #f]`
				`[output-type: int16])`

...	...	...	...	...

Figure 9: An illustration of how Rake uses bottom-up program synthesis to lift the Sobel filter to the Uber-Instruction IR.

$\mathcal{S}$ is the set of sub-expressions of e lifted to the Uber-Instruction IR. Then, for each $s \in \mathcal{S}$, We formulate the following query to the SMT solver:

$$\exists\, i \in s.\quad interpret(e) = interpret(s[i \rightarrow i'])$$

We want to check if there exists an uber-instruction i in the sub-expression s, such that updating the parameters to i makes the output of s and the output of e equal. For instance, if i is the `narrow` uber-instruction, we may update the `saturate?` flag to also perform saturation while narrowing.

3.2.2 *Replace.*

The replace step is similar to the update step, except that instead of updating an instruction, Rake attempts to replace an instruction with a different uber-instruction:

$$\exists\, i \in s.\ j \in \text{UberIR}.\quad interpret(e) = interpret(s[i \rightarrow j])$$

We want to check if there exists an uber-instruction i in the sub-expression s, such that when we replace i with another uber-instruction j, the output of s and e are equal.

3.2.3 *Extend.*

Finally, if neither update or replace steps are successful, Rake lifts e by extending the lifted sub-expressions with a new uber-instruction:

$$\exists\, i \in \text{UberIR}.\ s_0, ..., s_n \in \mathcal{S}.\quad interpret(e) = interpret(i(s0, ..., s_n))$$

3.2.4 *Demonstrative Example.*

Figure 9 illustrates the lifting process for the Sobel filter expression in Figure 3. For brevity, we do not show the numerous synthesis queries that are *unsat* and instead focus on the successful queries to demonstrate how the lifted expression evolves. Steps 1, 2, 3 and 4 show the base case for the recursive algorithm. Since there are no vector sub-expressions for leaf nodes, Rake extends the expression by adding a new uber-instruction. Step 5 shows an application of the *replace* step: by replacing the uber-instruction `widen` with the uber-instruction `vs-mpy-add`, Rake is able to construct an equivalent expression without increasing the total number of uber-instructions. Finally, steps 6 and 7 demonstrate applications of the update rule: additional sum operations can simply be folded into the existing `vs-mpy-add` instruction by updating the weight matrix. The weight matrix for the `vs-mpy-add` instruction specifies both the length of the multiply-add pattern, as well as the scalar weights.

Rake's lifting algorithm greedily folds each new Halide IR operation encountered during the bottom-up traversal into the existing Uber-Instruction IR expression. As a result, it may not always discover the Uber-Instruction IR representations with the fewest number of instructions. However, the greedy approach is scalable as each synthesis query attempts to add or modify at most a single uber-instruction.

4 ABSTRACTING DATA MOVEMENT

When lowering an expression from Uber-Instruction IR to the target ISA, Rake first synthesizes a *swizzle-free sketch* that expresses the computation using target ISA intrinsics, while abstracting away all data movement. The goal of synthesizing this sketch is to simplify the synthesis problem by identifying sequences of compute intrinsics that produce the correct output while assuming all required data is present in registers in the correct layout required for each instruction. To prove the correctness of a swizzle-free sketch, it is sufficient to show that there exists a sequence of loads and swizzles for which the sketch produces the correct output. Identifying the most optimal set of data movement instructions in the target ISA can be deferred until a correct sketch is found. Therefore, when synthesizing a swizzle-free sketch, we introduce two additional constructs to the search grammar that are used to represent data movement: `??load` and `??swizzle`.

Figure 10 shows the definitions of `??load` and `??swizzle` operations in pseudo-code. The `??load` operation implements the initial loading of data from memory into a vector register or vector register pair. There are three inputs to the `??load` operation: (1) the set of data values read from memory by the input expression (`live-data`), (2) a boolean flag indicating whether the operation loads a vector or a vector-pair (`vec-pair?`) and (3) the required type for elements in the output vector (`elemT`). The `??load` operation then returns a *symbolic* vector (or vector-pair), where each

```
(define (??load live-data elemT vec-pair?)
  (λ (i) (choose* ;; The synthesizer can pick any value from the
                  ;; live-data set after filtering
    (filter (λ (v) (eq? (type v) elemT))  live-data))))

(define (??swizzle exprs elemT vec-pair?)
  (λ (i) (choose* ;; The synthesizer can pick any value from exprs
                  ;; after filtering
    (filter (λ (v) (eq? (type v) elemT)) (get-vals exprs)))))
```

Figure 10: Definition of `??load` and `??swizzle`.

lane of the vector holds one of the live data values of the requested element type. A symbolic vector encodes the set of all possible vectors (of the requested type) that can be constructed from swizzling live data. This allows the synthesizer to concretize the symbolic vector into any one of its possible values, allowing us to represent all possible swizzling patterns. The `??swizzle` operation similarly implements the re-arrangement of data produced by one or more sub-expressions. Instead of returning a symbolic vector of data read from memory, it returns a symbolic vector populated with data produced by sub-expressions.

4.1 Verifying Sketches

In order to prove the validity of a candidate swizzle-free sketch, Rake must select a concrete instantiation for each symbolic vector produced by the `??load` and `??swizzle` operations, such that the output of the overall expression matches the output of the input expression we are trying to lower. The verification problem can be formally specified as follows:

$$\exists\, v_0, \ldots, v_n.\ \ \forall i \in lanes.\ \ input[i] = sketch[i]$$

where $v_0, \ldots, v_1$ represent the concrete instantiations for each symbolic vector.

The amount of work a synthesizer must do to find concrete instantiations for each vector is proportional to the number of lanes in that vector since the synthesizer must pick a value to populate each lane. As vectors grow larger (HVX vectors have up to 128 lanes), this becomes expensive. Additionally, more lanes in the output vector generally mean the set of live-data values to choose from is also larger. Fortunately, both of these challenges can be addressed by verifying incrementally for each lane of the output vector. For instance, we can simplify the verification query to only verify for the first lane of the vector:

$$\exists\, v_0, \ldots, v_n.\ \ input[0] = sketch[0]$$

The synthesizer must now only instantiate lanes of the symbolic vectors $v_0, \ldots, v_n$ that are required to compute the first lane of the overall output. While such a query does not guarantee the sketch is correct, it allows Rake to quickly reject obviously incorrect sketches, since if the two expressions produce unequal outputs for the first lane of the vector, the sketch cannot be correct. The more expensive verification query is then reserved for swizzle-free sketches that pass this initial pruning step.

5 SYNTHESIZING SWIZZLES

Once a valid swizzle-free sketch is found, Rake synthesizes an implementation to replace each `??load` or `??swizzle` operation in

```
Input: An expression in the uber-instruction IR
Output: An equivalent expression in the target ISA
1  Function Lower($e$, $\ell$)
2      $\mathcal{I} \leftarrow null$                              ▷ Best lowered implementation
3      $\beta \leftarrow \infty$                                   ▷ Expression cost upper-bound
4      for $sl \in$ SubexprLayouts($e$, $\ell$) do
5          $\mathcal{S} \leftarrow \emptyset$                          ▷ Set of lowered sub-exprs
6          for $se \in$ Subexprs($e$) do
7              $\mathcal{S} \leftarrow \mathcal{S} \cup$ Lower($se$, $sl$)
8          while $\tau \leftarrow$ SynthesizeSketch($e$, $\mathcal{S}$, $\beta$) $\neq unsat$ do
9              $v \leftarrow$ InferCost($\tau$)
10             if $\varepsilon \leftarrow$ SynthesizeSwizzles($e$, $\tau$, $\ell$, $\beta - v$) $\neq unsat$ then
11                 $\mathcal{I} \leftarrow \varepsilon$
12                 $\beta \leftarrow$ InferCost($\varepsilon$)
13     return $\mathcal{I}$
```

Algorithm 2: Lowering expressions from the Uber-Instruction IR to the target ISA.

the sketch. The synthesis problem can be formulated as follows:

$$\forall\, v_0, \ldots, v_n \in \tau.\ \ \exists\, e_0, \ldots, e_n \in \mathcal{G}_{sw}.$$
$$interpret(\tau[v_i \rightarrow e_i]) = interpret(e)$$

For all symbolic vectors in the sketch $v_0, \ldots, v_n$ (each representing an abstract swizzle), we search for swizzle expressions $e_0, \ldots, e_n$ constructed using the grammar of swizzle intrinsics G_{sw} in the target ISA, such that replacing the symbolic vectors with the swizzle expression still produces the correct output. In practice, Rake will not try to replace all abstract swizzles at once, but do so one at a time.

5.1 Backtracking and Intermediate Data Layouts

The incremental approach described thus far for lowering expressions from Uber-Instruction IR to the target ISA, although scalable, runs the risk of introducing inefficiencies in the final implementation.

Sub-optimal Sketches. The most efficient swizzle-free sketches do not necessarily yield the most performant implementations. The cost overhead of the required data movement may outweigh the benefit of using fewer compute instructions. To address this, we introduce backtracking to our lowering algorithm, shown in Algorithm 2. Whenever a lowered implementation ϵ is synthesized for the input expression, Rake updates the expression cost upper bound β (initially set to infinite) and then backtracks to synthesize another implementation. With each new implementation, the cost upper bound is tightened until a better implementation cannot be found.

Intermediate Data Layouts. Since Rake builds the output expressions bottom-up by lowering one uber-instruction at a time, the lowered sub-expressions are verified against sub-expressions in the input lifted expressions. This is problematic since it forces lowered sub-expressions to produce their output in the same layout as the input lifted sub-expressions. For example, since the HVX `vtmpy` intrinsic produces deinterleaved output, the implementation in Figure 8 interleaves the output produced by the intrinsic to undo the implicit deinterleaving. However, if the expression being lowered is producing an intermediate output, then it may be beneficial to not interleave the output for two reasons: 1) just as how `vtmpy` produces deinterleaved output, other instructions down the pipeline may interleave the output, thus eliminating the need to do the swizzle at all, 2) even if the swizzle is required, it may be less expensive to do it later. For example, it is much cheaper to swizzle the output of a large reduction than to swizzle all of its inputs. To address this concern, we parameterize our lowering algorithm over the data layout ℓ of the output we wish to lower to. This allows Rake to synthesize a variety of lowered implementations for each sub-expressions, each producing different permutations of the same intermediate output. For HVX, we consider interleaved and deinterleaved layouts since HVX instructions only perform these permutations implicitly. Simpler ISAs that do not have implicit data movement in compute instructions may not require this step.

6 IMPLEMENTATION

Rake's core synthesis algorithm is implemented in Racket, using the Rosette 4.0 framework [27] with z3 [11] as the back-end solver. We've also added C++ code inside Halide to extract qualifying Halide IR expressions and compile them to Racket syntax. We have also implemented within Halide an S-Expression parser to convert the S-Expressions synthesized by Rake back to Halide IR. Finally, we manually implemented an interpreter (in Racket) for both the HVX intrinsics provided by LLVM as well as the derived uber-instructions.

Deriving Uber-Instruction IR. The quality of code generated by our lift-then-lower approach is sensitive to the design of the Uber-Instruction IR used. For instance, if the uber-instructions are too general (coarse Uber-Instruction IR), the set of ISA instructions that need to be enumerated when lowering an uber-instruction can get large. This in turn makes synthesis slow or at times intractable. On the other hand, if the uber-instructions are too low-level, there can be many different ways to express the input expression in the Uber-Instruction IR and very few ways to lower the lifted representation down to the target ISA. Our instruction-selection algorithm then effectively becomes a greedy instruction selector, which can easily generate sub-optimal implementations since it explores a very small subspace of all possible implementations. The goal is to design an Uber-Instruction IR coarse enough for the greedy lifting algorithm to find the correct representation, yet not so coarse that lowering becomes intractable. To derive the set of uber-instructions for HVX, we identified clusters of intrinsics that had related or overlapping semantics. This was fairly straightforward since the HVX documentation already lists similar instructions together. Then, we manually defined an uber-instruction that implemented the common higher-level compute pattern. The uber-instructions were designed such that each intrinsic in the target ISA was expressible by at least one uber-instruction. For example, we can express the HVX vector-addition intrinsic *vadd* using the `vs-mpy-add` uber-instruction, since addition is simply a multiply-add with multiplicative weights of `(1 1)` (that is, a multiply-add where each input is first multiplied by 1 then added together).

Cost Model. Our implementation uses a variation of instruction-count to estimate the cost of an HVX expression. Since HVX has multiple hardware resources (such as multiply, shift or permute) and different instructions can execute on different hardware resources within the same cycle, we count the number of instructions per resource and take the maximum of the computed values. This biases our cost model towards implementations that distribute the computation across resources.

Extending to other ISAs. At a high-level, extending Rake to a new target ISA requires implementing an interpreter for the available intrinsics in Racket, along with designing and implementing an appropriate Uber-Instruction IR. Preliminary results from our efforts to extend Rake to ARM Neon instructions suggest that the set of uber-instructions derived for HVX can be re-used for ARM with only slight modifications. This is rather unsurprising since both ARM and HVX target the same high-level compute patterns for fixed-point arithmetic. Furthermore, we were able to automatically generate a large portion of the required ARM Neon interpreter by leveraging Halide's ARM-specific code optimization rules. These re-write rules, originally intended to map Halide IR expressions to ARM instructions, can be used to map each ARM instruction to an equivalent Halide IR expression, which Rake can already interpret. For some backends, interpreters can be automatically generated from the pseudo-code provided in the documentation, as demonstrated by Vegen [8] for Intel x86 SIMD instructions.

In general, we believe our algorithm is suitable for target backends that, similar to HVX, have a large number of instructions containing many variants of relatively few compute patterns. In contrast, backends that expose only a small number of very distinct instructions would not benefit much from our approach.

7 EVALUATION

We evaluate Rake using a suite of 21 benchmarks, listed in Table 1. The benchmarks consist of open-source applications taken from the Halide repository as well as sample Halide programs provided in the Hexagon Software Development Kit (SDK) v3.5.2. These benchmarks, summarized below, span a range of image processing, computational photography, computer vision and machine learning workloads.

- *Image Processing.* Our test suite includes image processing operations, implementing fundamental operations such as blurs (box blur, gaussian blur, median filter), edge detection (Sobel filter), image dilation and general 3×3 convolutions. For Gaussian blur, we include implementations for three different radii: 3, 5 and 7. For general convolutions, we include implementations for both 16-bit and 32-bit accumulators.
- *Machine Learning.* This subset contains Halide implementations of core Tensorflow operations, including normalization layers (l2norm, softmax), elementwise layers (add, mul), pooling layers (average_pool, max_pool), reduction (mean), fully connected layers, and convolutional layers (conv, depthwise_conv).
- *Camera Pipeline.* This is the Frankencamera pipeline [1] for processing raw data from an image sensor into a color image. The pipeline performs hot-pixel suppression, demosaicking, color correction, gamma correction, and contrast.
- *Matrix Multiplication.* This benchmark implements quantized matrix-multiplication of two unsigned 8-bit matrices.

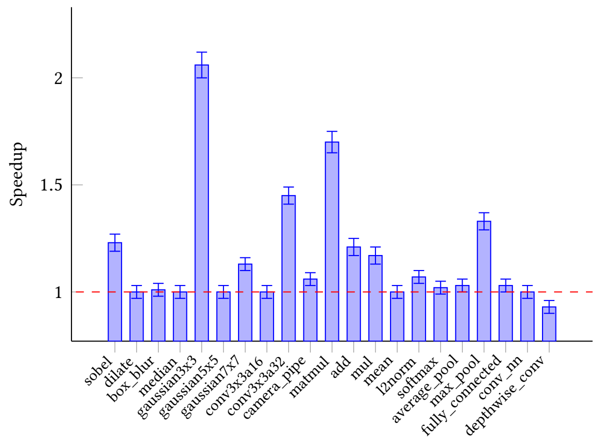

Figure 11: Speedups for Rake over the default Halide HVX backend. Across the 22 benchmarks, Rake improves performance by an average of 18% over the existing highly-optimized backend, which has been developed over years by Qualcomm and Google engineers.

For all benchmarks, we use the existing Halide schedules found in the implementations; Rake only changes instruction selection and does not alter the overall structure of the generated code. The set of benchmarks outlined above generates a total of 450 qualifying vector expressions that Rake attempts to optimize. Rake currently ignores all scalar expressions as well as all trivial vector expressions, such as a single variable, non-strided vector-loads, or scalar broadcasts. Rake assumes these are handled correctly by LLVM.

The benchmarks were compiled on a Windows 10 desktop machine with an AMD Ryzen Threadripper 2950X 3.5GHz 16-core CPU and 128GB of RAM. All HVX runtime performance numbers were computed using the reported cycle counts from Qualcomm's Hexagon Simulator v8.3.07.

7.1 Runtime Performance

To evaluate the effectiveness of Rake's instruction selection algorithm, we compare the runtime performance of all benchmarks for the Rake HVX backend, using the existing Halide 12.0 HVX backend as the baseline. The existing HVX backend is developed by Qualcomm and Google engineers, representing years of developer effort, and has been used for compiling production Android code distributed to millions of devices [21, 29].

Figure 11 graphs the results of our experiment. On average, Rake improved the overall runtime performance by 18%, with a maximum observed speedup of 2.1× in the `gaussian3x3` benchmark and the lowest observed speedup of 0.93× in `depthwise_conv`. 10 of the 21 benchmarks demonstrated a performance improvement beyond the 3% margin of error introduced by the simulator, with another 10 benchmarks showing identical performance. Upon manual inspection of the generated code, we discovered that Rake did improve instruction selection in some of these benchmarks. However, these optimizations did not result in overall performance improvement either because the benchmarks were memory-bound or because the optimizations were not in the critical code path. Rake performed worse than Halide's optimizer on only a single benchmark.

To illustrate the breadth of optimizations discovered by Rake, we now discuss a handful of representative examples, shown in Figure 12. The figure shows the Halide IR expression, as well as the optimized code generated by Halide and by Rake.

7.1.1 *Missing Optimization Patterns.*

The first class of improvements made by Rake over Halide's existing Hexagon optimizer relate to identifying missing optimization patterns. Through search, Rake considers a much larger space of implementations and discovers optimizations not handled by any of Halide's existing rewrites. We already highlighted three such instances from the Sobel filter in §2.1. In Figure 12, we provide three more examples from other benchmarks in our test suite.

- **`average_pool`:** The input code implements an addition between vectors of type `uint16` and `uint8`. The Halide implementation first zero-extends the `uint8` vector and then performs vector addition to complete the implementation. Rake, by contrast, uses a single widening multiply-add instruction with a multiplicative weight of 1.
- **`camera_pipe`:** In this example, Rake removes the `vmax` instruction since the instruction `vpackub` already saturates the value to an unsigned byte, making the max with zero operation redundant.
- **`add`:** Rake is able to fold the shift operation into a widening multiply-add, implemented using the single `vmpy-acc` intrinsic. Halide instead zero-extends the input before implementing the shift-left and addition using a non-widening multiply-add using `vmpyi-acc`. Since `vmpy-acc` generates a vector-pair as output, two `vmpyi-acc` instructions are needed to compute the equivalent tile.

7.1.2 *Semantic Reasoning.*

In addition to discovering optimization patterns missing in Halide's rule-set, Rake can also discover *context-specific* optimizations that require semantic reasoning about the expression, such as inferring the range of values possible for an intermediate output. Figure 12 shows two examples from the benchmarks `l2norm` and `gaussian3x3`.

- **`l2norm`:** The input IR multiplies a vector of words with a vector of halfwords. Halide generates the results in two steps: first, it multiplies all odd halfwords with the vector of words using the `vmpyio` instruction, and then it uses the shift-left instruction `vaslw` to move the even halfwords into the odd indices and repeats the first step. Rake, on the other hand, avoids the shift-left operation and instead uses the `vmpyie` instruction to directly multiply the even halfwords with the vector of words. Interestingly, HVX only offers the `vmpyie` instruction for unsigned halfwords. Therefore, to use this instruction safely, Rake must prove that the sub-expression producing the input to this pattern will never produce negative values (i.e, the most significant bit is always 0).
- **`gaussian3x3`:** Rake uses a fused instruction to implement the rounding-shift-right as well as the lowering cast. This transformation is only safe if the upper-most 8-bits of the input values are always 0. In other words, performing a truncating cast or a saturating cast produces the same outputs.

	Benchmark	Code Pattern (Halide IR)	Halide Codegen (HVX)	Rake Codegen (HVX)
Missing Patterns	average_pool	`wild_u16x + uint16x128(wild_u8x)`	`/* Latency: 3 */ wild_u16x + vzxt(wild_u8x)`	`/* Latency: 2 */ vmpy-acc(wild_u16x, wild_u8x, 1)`
	camera_pipe	`uint8x128( max( min(wild_i16x, x128(127)), x128(0)))`	`/* Latency: 4 */ vpackub( // saturate to uint8 vmax( vmin(wild_i16x, vsplatb(127)), vsplatb(0)))`	`/* Latency: 3 */ vpackub( vmin(wild_i16x, vsplat(127)))`
	add	`int16x128(wild_u8x) << 6 + x128(int16(wild_u8) * -64)`	`/* Latency: 6 */ x = vzxt(wild_u8x) vmpyi-acc( // Multiply-add vsplat(int16(wild_u8) * -64), (lo x), 64) // Lower 64 elements vmpyi-acc( // Multiply-add vsplat(int16(wild_u8) * -64), (hi x), 64) // Upper 64 elements`	`/* Latency: 2 */ vmpy-acc( // Widening multiply-add vsplat(int16(wild_u8) * -64), wild_u8x, 64)`
Semantic Reasoning	l2norm	`x64(wild_i32) * int32x64(wild_i16x)`	`/* Latency: 6 */ vmpyio( // Mul i32s with odd i16s vsplat(wild_i32), wild_i16x) vmpyio( vsplat(wild_i32), vaslw(wild_i16x, 16)) // Shift-left`	`/* Latency: 4 */ vmpyio( // Mul i32s with odd i16s vsplat(wild_i32), wild_i16x) vmpyie( // Mul i32s with even i16s vsplat(wild_i32), wild_i16x)`
	gaussian3x3	`uint8x128( (wild_i16x + x128(8)) >> x128(4))`	`/* Latency: 8 */ vshuffeb( // extract lower byte vasr( // shift-right wild_i16x + vsplat(8), 4), vasr( wild_i16x + vsplat(8), 4))`	`/* Latency: 2 */ // Fused shift right, round // and saturate vasr-rnd-sat( wild_i16x, 4)`

Figure 12: Rake uses search to find new optimization opportunities not considered by the existing Halide HVX backend. Here, `wild_{i|u}bb{x|}` represents a subtree of signed (i) or unsigned (u) bb-bit value that is either a scalar (no suffix) or vector (x).

7.1.3 Data Movement. The third major category of perfomance improvements comes from improved data movement.

- **`gaussian3x3`, `conv3x3a32`:** In some benchmarks, such as the Sobel filter and `conv3x3a32`, Rake exploits fused multiply-add sliding-window instructions such as `vtmpy` (3-wide reduction) or `vrmpy` (4-wide reduction). A major advantage of of these instructions is that they reduce the number of vector loads necessary. For instance, row (a) in Figure 4 shows that in addition to the smaller compute latency, the `vtmpy` instruction requires one fewer vector load.
- **`add`, `mul`, `average_pool`, `max_pool`, `matmul`:** We found that Rake frequently avoids unnecessary data shuffling operations introduced by Halide's optimizer. The most common example of this was Halide adding an interleave operation to undo the implicit deinterleaving of a prior compute instruction (or vice versa). While Halide's optimizer has an optimization pass dedicated specifically to eliminating such unnecessary interleaves and deinterleaves, it is not always able to do so.

7.2 Compilation Performance

Table 1 shows a breakdown of compilation times for each benchmark. On average, Rake took 62 minutes to compile each benchmark, with a median compilation time of roughly 21 minutes. The largest expression optimized by Rake required a sequence of 103 HVX intrinsics to implement.

The mean time spent lifting to the Uber-Instruction IR was 154 seconds, which equates to 9% of the compilation time. Synthesizing swizzle-free sketches was more expensive, taking on average 397 seconds or 21% of the total compilation time. Synthesizing data movement accounted for the majority of compilation time, taking on average 53 minutes and making up almost 70% of the total synthesis time. There are three reasons why considerably more time is spent on synthesizing data movement than on synthesizing swizzle-free sketches. First, the search space for swizzles suffers from *symmetry*, resulting in a larger set of candidate expressions. In program synthesis, symmetries arise when multiple expansions of a grammar result in the same program; the synthesizer unnecessarily considers the same program multiple times. Second, Rake can specialize the search space for compute instructions by leveraging semantic information exposed by lifting. The same cannot be done for swizzling, so Rake always considers the full set of shuffling instructions. Lastly, due to the backtracking nature of our instruction selection algorithm, Rake often spends considerable time trying to prove that the required swizzle cannot be implemented within a given instruction budget.

Compared to pattern-matching optimizers, such as Halide's existing HexagonOptimizer, Rake presents a different performance to compilation time trade-off. As an offline optimizer, Rake can be used to fine-tune applications for a given hardware platform. In addition, Rake can be a valuable tool to inform compiler developers of patterns and optimizations that are both missing in their machinery and important for performance. Furthermore, since Rake

Table 1: Compilation statistics for the Hexagon HVX backend.

Benchmark	Optimized Exprs	Lifting Queries	Sketching Queries	Swizzling Queries	Lifting Time (s)	Sketching Time (s)	Swizzling Time (s)	Total Synthesis Time (s)
sobel	4	348	252	3748	12	27	7274	7313
dilate	8	560	1376	1168	45	48	114	207
box_blur	4	76	820	597	4	538	1033	1575
median	40	1440	5256	5360	119	202	397	718
gaussian3x3	4	368	100	376	46	52	137	235
gaussian5x5	8	364	98	1636	22	20	214	256
gaussian7x7	20	864	320	6835	50	173	1358	1581
conv3x3a16	4	320	73	1254	96	2552	953	3601
conv3x3a32	4	404	41	1560	204	1628	750	2582
camera_pipe	44	2037	6177	9002	1153	429	13782	15364
matmul	10	594	456	6224	104	175	6835	7114
add	4	306	403	638	47	118	431	596
mul	4	492	427	576	129	184	1559	1872
mean	2	43	37	298	11	82	186	279
l2norm	4	262	163	301	49	13	77	139
softmax	18	755	679	1311	221	352	683	1256
average_pool	6	186	476	2246	6	42	248	296
max_pool	6	24	156	126	1	6	8	14
fully_connected	39	608	198	416	63	160	512	735
conv_nn	140	3673	4165	1612	558	1002	19335	20895
depthwise_conv	77	2926	3542	7941	301	538	10593	11432

uses SMT solvers for search and verifies its transformations, it can even highlight bugs in the rule-based compiler. In fact, during our manual inspection of the generated code, we found three bugs in Halide's HVX code-generation involving unsafe instruction selection. We have communicated these bugs to the Halide developers and the appropriate fixes have been merged into the master branch. We intend to keep working with Halide developers and share the optimization patterns we discovered.

7.3 Limitations

While Rake's algorithm is designed to generalize to other ISAs, such as ARM and Intel's AVX, our prototype currently only supports Hexagon's HVX backend. Our experiments also highlighted two significant limitations in the way Rake is integrated into Halide. Unlike Halide's existing optimizer, which may modify the layout in which data is stored in an intermediate buffer to enable better instruction selection across multiple expressions, Rake optimizes each expression individually. This was the key reason behind the performance degradation observed in the `depthwise_conv` benchmark, but also reduced the speedup observed in other benchmarks like `average_pool`. Secondly, in some cases, better instruction selection was possible if Rake had access to certain loop-invariants. For example, when the Halide schedule re-uses reads in a rotating buffer, Rake is unaware of the relationship between data read in this iteration and the data read in previous iterations, preventing it from using more optimal sliding window instructions.

8 RELATED WORK

Program Synthesis searches for programs that satisfy some user-provided constraints [13]. A number of approaches have been developed to solve the synthesis problem [6, 12] using different algorithms, such as constraint-based search [25], enumerative search [19], or stochastic search [24]. Rake relies on Rosette [27] as the underlying synthesis framework. While these general-purpose synthesis algorithms encode many clever pruning strategies and heuristics to make decisions, they scale poorly to the instruction selection problem [19].

Superoptimization is the task of using search to optimize low-level programs based on instruction semantics. Traditionally superoptimizers like Optgen [7], Lens [19] and the one proposed by Bansal and Aiken [5] have focused on peephole optimization over short instruction sequences, and target a broader class of optimizations than just instruction selection. Rake by contrast is designed to be a target-specific instruction-selector that can operate on much larger instruction sequences. Souper [23], a middle-end LLVM IR superoptimizer, performs rewrites on its own IR via dataflow analysis. Unlike Rake, Souper does not support vector instructions or hardware-specific intrinsics.

Autovectorization is the related task of converting scalar code into vectorized implementations. Vegen [8] is an auto-vectorizer that jointly performs instruction-selection for complex vector ISAs. Unlike Rake, Vegen uses automatically generated pattern matching rules for instruction selection. While automatically generated, the pattern matching rules suffer from the same limitations as Halide's optimizer: they cannot perform semantic reasoning on expressions and may miss creative applications of instructions. Diospyros [28] is another auto-vectorizer designed to synthesize efficient implementations of kernels on DSP architectures. It uses equality saturation to identify creative swizzles and vectorization patterns in its own IR. When lowering the discovered shuffles to the backend target, it delegates the instruction-selection process to the vendor-supplied DSP compiler toolchain.

Verified Lifting [14] is a technique for extracting the semantics of code by re-writing it in a higher-level abstraction through program synthesis. Prior work has applied verified lifting as a means of recovering intent from legacy code to port it to newer DSLs and frameworks [2, 3, 9, 14]. We apply verified lifting to the instruction-selection problem and use it to infer the higher-level compute patterns in the input IR code.

Data Swizzling is the task of inferring permutations of data and computation to optimize performance. Swizzle Inventor [18] is a tool that infers swizzles to optimize applications for GPU memory hierarchies. Unlike Rake, Swizzle Inventor can only synthesize data-movement and requires the users to provide a sketch describing where swizzles can be added; on the other hand, integrating Swizzle Inventor into Rake could improve the quality of our swizzle synthesis.

9 CONCLUSION

In this work, we described Rake, which takes code in the Halide intermediate representation and uses program synthesis to perform target-specific instruction selection for the Hexagon HVX digital signal processor. Rake scales to real-world vectorized expressions by decomposing the task into three different synthesis queries. On a suite of 21 real-world benchmarks, Rake improves performance an average of 18% and up to 2.1× over the existing combination of Halide and LLVM, which use human-created pattern-matching rules to perform instruction selection.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1955488, IIS-2027575, CCF-1723352, DOE award DE-SC0016260, ARO grant W911NF2110339, and ONR grant N00014-21-1-2724; the Intel-NSF CAPA center, and gifts from Adobe, Facebook, Google, and VMware. We would also like to thank Dillon Sharlet, Rastislav Bodik, Samuel Kaufman, and Krzysztof Drewniak for sharing their valuable feedback and insights during this research.

REFERENCES

[1] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. 2010. The Frankencamera: An Experimental Platform for Computational Photography. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) *(SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1833349.1778766

[2] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1205–1220. https://doi.org/10.1145/3183713.3196891

[3] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics* 38 (11 2019), 1–13. https://doi.org/10.1145/3355089.3356549

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–17. https://doi.org/10.1109/FMCAD.2013.6679385

[5] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 394–403. https://doi.org/10.1145/1168917.1168906

[6] Rastislav Bodik and Barbara Jobstmann. 2013. Algorithmic Program Synthesis: Introduction. *Int. J. Softw. Tools Technol. Transf.* 15, 5–6 (oct 2013), 397–411. https://doi.org/10.1007/s10009-013-0287-9

[7] Sebastian Buchwald. 2015. Optgen: A generator for local optimizations. In *International Conference on Compiler Construction*. Springer, 171–189. https://doi.org/10.1007/978-3-662-46663-6_9

[8] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[9] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

[10] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. 2014. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro* 34, 02 (mar 2014), 34–43. https://doi.org/10.1109/MM.2014.12

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[12] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (Hagenberg, Austria) *(PPDP '10)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1836089.1836091

[13] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[14] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 711–726. https://doi.org/10.1145/2908080.2908117

[15] David Ryan Koes and Seth Copen Goldstein. 2008. Near-Optimal Instruction Selection on Dags. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Boston, MA, USA) *(CGO '08)*. Association for Computing Machinery, New York, NY, USA, 45–54. https://doi.org/10.1145/1356058.1356065

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

[17] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 122–126. https://doi.org/10.1145/36206.36194

[18] 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 *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3297858.3304059

[19] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 297–310. https://doi.org/10.1145/2980024.2872387

[20] William K. Pratt. 2007. *Digital Image Processing: PIKS Scientific Inside*. Wiley-Interscience, USA. https://doi.org/10.1002/0470097434

[21] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM* 61, 1 (Dec. 2017), 106–115. https://doi.org/10.1145/3150211

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[23] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2018. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL]

[24] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (Jan. 2016), 114–122. https://doi.org/10.1145/2863701

[25] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. *SIGPLAN Not.* 42, 6, 167–178. https://doi.org/10.1145/1273442.1250754

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[27] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA,

135–152. https://doi.org/10.1145/2509578.2509586

[28] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[29] Kyle Wiggers. 2017. Qualcomm Hexagon 685 DSP is a Boon for Machine Learning. https://www.xda-developers.com/qualcomm-snapdragon-845-hexagon-685-dsp/