Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures

Akash Kothari*
UIUC, USA
akashk4@illinois.edu

Abdul Rafae Noor*
UIUC, USA
arnoor2@illinois.edu

Muchen Xu
UIUC, USA
muchenx2@illinois.edu

Hassam Uddin
UIUC, USA
hassamu2@illinois.edu

Dhruv Baronia
UIUC, USA
baronia3@illinois.edu

Stefanos Baziotis
UIUC, USA
sb54@illinois.edu

Vikram Adve
UIUC, USA
vadve@illinois.edu

Charith Mendis
UIUC, USA
charithm@illinois.edu

Sudipta Sengupta
Amazon Web Services, USA
sudipta@amazon.com

Abstract

As modern hardware architectures evolve to support increasingly diverse, complex instruction sets for meeting the performance demands of modern workloads in image processing, deep learning, etc., it has become ever more crucial for compilers to provide robust support for evolution of their internal abstractions and retargetable code generation support to keep pace with emerging instruction sets. We propose Hydride, a novel approach to compiling for complex, emerging hardware architectures. Hydride uses vendor-defined pseudocode specifications of multiple hardware ISAs to automatically design retargetable instructions for *AutoLLVM IR*, an extensible compiler IR which consists of (formally defined) language-independent and target-independent LLVM IR instructions to compile to those ISAs, and automatically generated instruction selection passes to lower AutoLLVM IR to each of the specified hardware ISAs. Hydride also includes a code synthesizer that automatically generates code generation support for schedule-based languages, such as Halide, to optimally generate *AutoLLVM IR*. Our results show that Hydride is able to represent 3,557 instructions combined in x86, Hexagon, ARM architectures using only 397 AutoLLVM IR instructions, including (Intel) SSE2, SSE4, AVX, AVX2, AVX512, (Qualcomm) Hexagon HVX, and (ARM) NEON vector ISAs. We created a new Halide compiler with Hydride using only a formal semantics of Halide IR, leveraging the auto-generated AutoLLVM IR and back-ends for the three hardware architectures. Across kernels from deep learning and image processing, this compiler is able to perform just as well as the mature, production Halide compiler on Hexagon, and outperform on x86 by 8% and ARM by 3%. Hydride also outperforms the production Halide's LLVM back end by 12% on x86, 100% on HVX, and 26% on ARM across the same kernels.

ACM Reference Format:
Akash Kothari, Abdul Rafae Noor, Muchen Xu, Hassam Uddin, Dhruv Baronia, Stefanos Baziotis, Vikram Adve, Charith Mendis, and Sudipta Sengupta. 2024. Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, San Diego, CA, USA, 16 pages. https://doi.org/10.1145/3620665.3640385

1 Introduction

Domain-specific hardware accelerators and complex vector extensions to existing architectures are emerging to efficiently support modern workloads in domains such as deep learning, image processing, etc. These custom accelerators and CPU/GPU extensions provide high performance and energy efficiency for important operations like matrix multiplication, tensor convolution, and others. Architectures such as Qualcomm's Hexagon DSP [9, 17] and Intel's AVX-512 vector ISA [12] have been extended with specialized instructions to optimize tensor and stencil computations. These include SIMD and non-SIMD (cross-lane) instructions to perform dot products, reduction operations, and in-register data movement across vector lanes (aka, swizzles). The number of instructions in ISAs such as x86 and Hexagon will continue to grow to support new workloads in evolving domains.

*Equally contributing authors.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640385

Challenges. A major challenge faced by compilers for such architectures is the difficulty of generating efficient code for these complex instructions. Domain-specific language (DSL) compilers such as Halide [11], TVM [7], and XLA [23] rely on retargetable compiler infrastructures such as LLVM [14] to generate code for a wide range of architectures. These languages depend heavily on complex vector hardware for high performance. Unfortunately, extending infrastructures like LLVM to keep pace with evolving hardware instruction sets has proven difficult *because the target-independent compiler IR has no inherent extensibility mechanisms that make adding support for new instructions naturally simple.* This process of extending compilers to support new instructions, especially complex ones, presents a number of challenges: (a) designing a new target-independent operation for a compiler IR entails meticulously searching long documents spanning thousands of pages on different hardware ISAs and finding common type of instructions (for example dot product instructions across multiple hardware targets) and then manually designing a target-independent instruction that can be lowered to those target instructions – this process is cumbersome and time-consuming; (b) generating code for each hardware target requires arduously implementing pattern matching code to translate input sequences of target-independent instructions to hardware instructions – this requires lot of engineering time and assiduous effort to ensure generated code is correct, and this gets significantly harder as ISAs grow larger and more complex; (c) and finally, the compiler IR and lowering support must automatically evolve with hardware ISAs.

Current Approaches. In order to overcome challenge (a) in compiler infrastructures such as LLVM and GCC, developers often implement target-specific intrinsics in compiler IRs that map to the corresponding target instructions directly. This approach eliminates the need to design target-independent operations that can be retargeted to other target ISAs. However, this approach severely hinders retargetability (and compilers mentioned below suffer from the same limitation because they all rely on LLVM). These compiler infrastructures do not address challenge (b) because they do not support automatic generation of target-specific intrinsics for complex instructions for a given input program. And neither of these state-of-the-art production compiler infrastructures address challenge (c) since their IRs and the lowering support for them does not evolve with target ISAs without manual engineering effort.

Because code generation and optimization support for LLVM IR is unable to automatically generate efficient, complex non-SIMD and swizzle instructions for different architectures, performance-sensitive DSL compilers like Halide use separate, target-specific back ends generating target-specific LLVM intrinsics for architectures such as x86, Hexagon, ARM, Power, etc. These back-ends in Halide use manually-crafted, *target-specific* pattern-matching rules to map the DSL-specific IR to complex vector hardware instructions. This approach defeats one key advantage of sophisticated, language-independent compiler infrastructures like GCC and LLVM because it reimplements a complex part of their functionality for a single high-level language. This approach also fails to address challenges (b) and (c). Manually engineering a back end for every target for each DSL compiler leads to replication of engineering effort and is, therefore, extraordinarily inefficient. Manually crafting separate, target-specific pattern-matching rules is cumbersome, error-prone and requires a lot of engineering effort and time, especially for large and complex ISAs, and the back-ends must be manually updated when new target instructions need to be supported.

Vegen [8] attempts to address challenge (b) by automatically generating pattern-matching rules using x86 instruction semantics. However, Vegen falls short due to several reasons: the pattern-matching rules are brittle since it does not generate multiple variants of patterns and any deviation from expected input code sequence would result in a failure to generate optimal code; it does not support specialized swizzle instructions critical for performance of tensor and stencil computations; and it fails to address (c), above, because it uses a fixed, language-independent compiler IR with no mechanism to evolve the IR.

Other projects, like Rake [1, 20], Diospyros [27] and Porcupine [10] partially address challenge (b) by leveraging program synthesis techniques for code generation but they only support a small number of target instructions and do not scale well to support synthesis of large, complex ISAs. Moreover, Rake requires users to modify source code of Halide programs[1] that use common computations like matrix multiplication to expose the dot product and data swizzle patterns explicitly to make synthesis tractable, which places a heavy burden on the programmer and violates the core Halide principle of separating computational specifications from scheduling optimizations. Rake [1, 20] defines a target-agnostic, language-independent IR, called Uber IR, to synthesize HVX and ARM intrinsics in LLVM. However, its Uber IR is manually designed and that approach is not scalable when considering large, evolving ISAs like x86 or extending support for HVX and ARM; therefore, fundamentally fails to address challenge (c).

Our Approach. We propose an alternative approach in which formally defined language-independent and target-independent LLVM IR operations, along with target-specific instruction selection passes, are *automatically generated* using only the vendor specifications of one or more hardware ISAs. We call this autogenerated IR the *AutoLLVM IR*. This enables a shared compiler infrastructure to evolve rapidly, while also achieving nearly complete hardware instruction coverage: together, these capabilities make the infrastructure

[1] `github.com/uwplse/rake/blob/master/benchmarks/hexagon/halide/matmul/src/matmul_generator.cpp`

both, highly extensible (to support new instructions in an existing ISA) and retargetable (across multiple ISAs). Moreover, because all the IR operations are defined using a precise, executable formal semantics, we use program synthesis in a language front-end (e.g., for a DSL like Halide) to generate highly efficient target-specific code that benefits fully from evolving vector hardware operations, without using brittle pattern matching.

Using this approach, we design a system called Hydride, which simultaneously solves three key problems: enabling automated design of the language-independent compiler IR to support multiple, rapidly evolving hardware ISAs, maximizing instruction coverage for complex ISAs, and avoiding the brittle pattern-matching approach to instruction selection. Hydride operates in two phases. The *offline* phase automatically generates compiler code during compiler development time (including the AutoLLVM IR, its formal semantics, target-specific LLVM IR instructions and their semantics for each specified target, and simple lowering from the former to the latter). The *online* phase uses program synthesis for target-specific translation from the front-end IR to AutoLLVM IR. This approach achieves both high performance (by using target information when synthesizing AutoLLVM IR code sequences) and scalability (because AutoLLVM IR is target-agnostic and therefore much more compact than individual target ISAs (Section 3)).

In summary, our key contributions are:

- A new methodology to automatically design a language-independent and target-independent compiler IR using a novel technique based on "similarity checking" of target instructions of multiple architectures.
- A new compiler infrastructure, Hydride, in which all compiler stages from DSL IR down to target-specific instructions (front-end IR to AutoLLVM IR translation, language-independent AutoLLVM IR operations, and translation to target-specific LLVM intrinsics) are automatically generated.
- A novel, scalable synthesis methodology that exploits the equivalence classes of target instructions, creates synthesis grammars separately for input subexpressions, together with synthesis strategy for complex cross-lane swizzle operations, to make program synthesis times reasonable with Hydride.
- An evaluation of this compiler for various kernels from image processing and deep learning on x86, Hexagon and ARM architectures, with no modifications to original Halide source code. Our results show that we achieve similar performance to the carefully engineered production compiler for Halide for all three architectures, *despite orders-of-magnitude lower compiler engineering effort.*

2 Hydride Overview

The key idea in Hydride is to *use the vendor-defined specifications of (multiple) hardware ISAs to automatically generate their formal semantics and reason about similarities between the various ISAs to automatically design or extend a formally-defined language-independent and target-independent compiler IR using parameterized operations.* This idea has several implications for the capabilities of Hydride. First, formal semantics of hardware ISAs enable Hydride to automatically design a retargetable compiler IR that can evolve automatically with hardware ISAs. Second, Hydride can automatically maximize the coverage of large ISAs with orders-of-magnitude lower engineering effort than state-of-the-art systems. Third, the formal semantics of Hydride's compiler IR enables language front ends to automatically target it using program synthesis techniques and, moreover, parameter selection for the target-independent IR operations enables the synthesizer to generate *target-specific* instruction sequences for high performance; together, these eliminate the need for manually engineering target-specific back ends in these compilers.

Figure 1 shows the workflow of Hydride. Hydride operates in two phases: an *offline* and an *online* phase. During the *offline* phase, Hydride Automatic IR Generator (Section 3) uses ISA pseudocode specifications provided by hardware vendors to automatically generate language-independent instructions for the LLVM compiler IR, along with their semantics, which we call the ***AutoLLVM IR***. We define these new operations as LLVM intrinsic functions to avoid the need for changes to existing LLVM passes. During the *online* phase (i.e., at compilation time), Hydride's Code Synthesizer (Section 4) uses syntax-guided synthesis to translate code for an input Halide program from the Halide front-end's IR to AutoLLVM IR.

3 Hydride's Automatic IR Generator

The Hydride Automatic IR Generator uses hardware ISA semantics to look for similar instructions (described in detail below). One of the main ideas in the approach used in Hydride is that *similar instructions from one or more machine instruction sets (ISAs) can be grouped into a (parameterized) equivalence class and represented as a machine-independent IR operation with symbolic parameters.* Different assignments of concrete values to those parameters represent different members of the equivalence class, i.e., different machine-specific instructions. This also implies that the target-specific back-end instruction selectors only need nearly trivial one-to-one translation. Moreover, instead of requiring a formal ISA semantics (needed for similarity checking) to be manually specified for each hardware ISA, Hydride automatically generates the ISA semantics from pseudocode specifications of instruction sets already specified by the hardware vendors in their respective programmer's manuals: [13] by Intel;

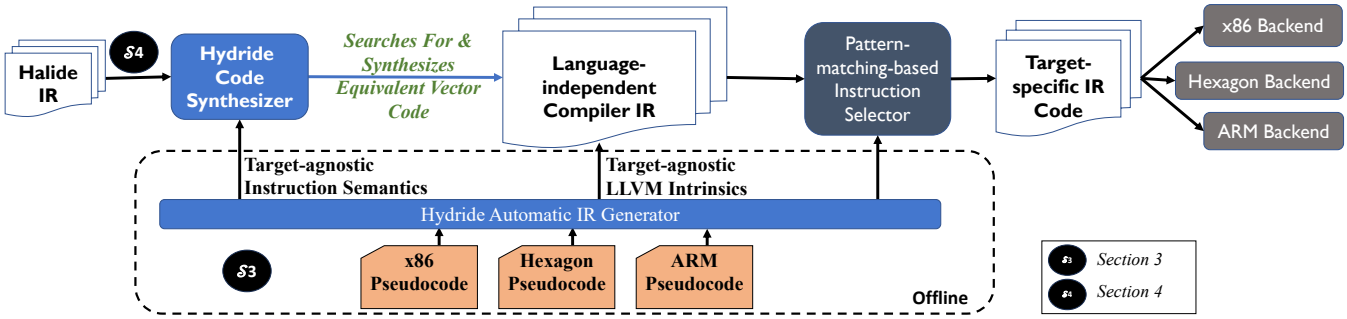


Fig. 1. Overview of compilation workflow Hydride with Halide.

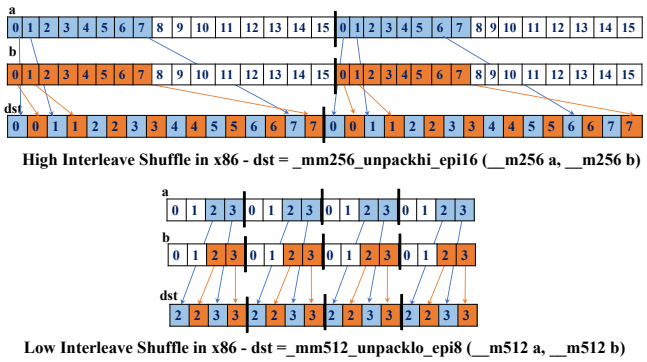


Fig. 2. Example of similar interleave instructions in x86.

[18] by Qualcomm; and [2] by ARM. An example of such specification for an Intel x86 instruction is shown in Figure 3. These specifications are parsed (using ISA-specific parsers we wrote) and translated into formal, machine-executable ISA semantics. Hydride uses the instruction semantics to perform similarity checking, explained next.

3.1 Similar Instructions and Equivalence Classes

Intuitively, Hydride deems two instructions as being *similar* if they perform equivalent computations and/or equivalent data movements, after abstracting away target-specific numerical properties like element bitwidth, vector lengths, shift factors, bit offsets, etc. For example, the x86 instruction `_mm512_add_epi16` operates on 512-bit registers with 16-bit elements and `_mm256_add_epi8` operates on 256-bit registers with 8-bit elements. Both fundamentally perform the same kind of computation: element-wise vector add. Therefore, these instructions are deemed to be similar.

The parameterization for equivalence can be quite expressive, in practice. Figure 2 shows an example of more complex similar instructions. These interleave shuffles are commonly used for changing data layouts (for example, performing matrix transpose) by interleaving data at different offsets into the lanes of input vector registers. Even though the vector sizes, element sizes, size of window of elements being interleaved and offset into the windows in both instructions are different, the data movement pattern are similar; the instruction-specific numerical quantities and details can be abstracted away while analyzing mainly the data layout pattern to realize that they are similar.

Now we formalize the notion of and the process of finding *similar instructions*. Suppose an instruction I has operational semantics $\Phi(I, \vec{k}^I)$, which depends on numerical parameters, $\vec{k}^I = k_1^I \cdots k_r^I$. For example, `_mm512_add_epi16` has two numerical parameters, $k_1 = 16$ (element size), $k_2 = 512$ (vector size). We construct the parameterized (symbolic) operational semantics of I, denoted $\Sigma(I, \vec{\alpha}^I)$ by substituting the numerical parameters with corresponding symbolic parameters, $\vec{\alpha}^I = \alpha_1^I \cdots \alpha_r^I$, or more formally, $\Sigma(I, \vec{\alpha}^I) = \Phi(I, \vec{k}^I)|_{\alpha_i/k_i, 1 \le i \le r}$.

We define two instructions, I and J as *similar* if both their semantics have the same number of numerical parameters (i.e., $|\vec{k}^I| = |\vec{k}^J|$) and the parameterized operational semantics of both instructions are equivalent, i.e., $\Sigma(I, \vec{\alpha}^I) \equiv \Sigma(J, \vec{\alpha}^J)$ for the same concrete values for corresponding numerical parameters. For example, when the numerical parameters of `_mm512_add_epi16`, $k_1 = 16$ and $k_2 = 512$, are substituted into numerical parameters, k_1 and k_2, of `_mm256_add_epi8`, their operational semantics can be verified to be equivalent using satisfiability modulo theories (SMT) solvers. Likewise, when the numerical parameters of `_mm512_add_epi16` ($k_1 = 8$ and $k_2 = 256$) are substituted into numerical parameters of `_mm512_add_epi16`, their operational semantics are deemed as equivalent. Section 3.3 describes the algorithm for finding similar instructions.

Given one or more target hardware ISAs, Hydride constructs *equivalence classes of all machine instructions in those ISAs*, such that two instructions I, J are assigned to the same equivalence class if and only if $\Sigma(I, \vec{\alpha}^I) \equiv \Sigma(J, \vec{\alpha}^J)$. Since all instructions in an equivalence class will have the same number of parameters, Hydride can define a (target-independent) instruction with that many symbolic parameters to represent all instructions in the class, say, $V(\vec{\alpha}^V)$. Once Hydride will have computed similarity as explained later, below, it simply constructs the equivalence classes of similar instructions and defines a target-independent IR operation, V, in this

manner for each equivalence class. We call the resulting IR definition the AutoLLVM IR. Note that each target-specific instruction, I, in a class has a fixed assignment of numerical values to those symbolic parameters. Thus, a language front end can perform AutoLLVM IR generation in an optimized target-specific manner simply by selecting the appropriate numerical parameter values. Alternatively, machine code generation from the symbolic V to any particular target instruction in the equivalence class simply requires selecting the appropriate numerical parameter values.

3.2 Hydride IR Generator

Checking for similarity between instructions requires analyzing the formal semantics for target ISA instructions ($\Phi(\cdot)$) and for their abstracted counterparts ($\Sigma(\cdot)$). The semantics of such instructions can be expressed using bitvector and integer operations [3]. We define a straightforward program representation called *Hydride IR* to represent semantics, together with several transformations on the IR, including loop rerolling, inlining, and constant propagation, and a dataflow analysis for type inference.

Checking for instruction similarity requires proving equivalence of the semantics. Solver-aided DSLs (SDSL) leverage SMT solvers to provide verification, synthesis and solving capabilities. Therefore, defining Hydride IR as a solver-aided DSL to express formal, machine-executable ISA semantics is useful in Hydride. Hydride uses Rosette [26] to define and implement Hydride IR, but makes some improvements to Rosette's synthesis engine for better efficiency (Section 4).

Hydride parses pseudocode specifications of x86, ARM and HVX instructions (such as the example in Figure 3(a)) using separate parsers and translates them into their semantics represented in Hydride IR. This approach enables Hydride to maximize the coverage of supported instructions.

3.3 Similarity Checking Engine

This component of Hydride performs the following steps.

Canonicalization of Hydride IR code. Hydride canonicalizes Hydride IR by performing function inlining, loop rerolling, etc. to ensure that all semantics of instructions contain at least two loops in a loop nest: one outer loop for representing iteration over lanes of input/output vector registers, and an inner loop for representing iteration over elements in a given lane (example in Figure 3(b)). This makes instruction-specific quantities such as number of elements, element size, vector size, etc. easier to identify before they are abstracted away. For SIMD instructions which have one loop iterating over elements of vectors, this step adds an artificial inner loop with one iteration.

Extraction of constants. Hydride extracts the constants from Hydride IR to abstract away any instruction-specific quantities like vector sizes, element sizes, etc. To ensure that constants for different parameters are not conflated together, and to ensure that bitwidths of two bitvectors are

Algorithm 1 Algorithm for the Similarity Checking Engine

```
Input: List of canonicalized ISA semantics in Hydride IR ISA_Sema
Output: Set of equivalence classes EqClasses
  function RunSimilarityCheckingEngine(ISA_Sema)
      EqClasses ← A set of Equivalent classes generated by Hydride
      SymSema ← ExtractConstants(ISA_Sema)        // Produce symbolic semantics
      EqClasses ← {}
      PerformEqChecking(SymSema, EqClasses)       // Perform similarity checking
      PermuteArgs(EqClasses)                      // Permute args of functions in EqClasses
      PerformEqChecking(SymSema, EqClasses) // Perform similarity checking again
      RefineEqClasses(EqClasses)                  // Refine equivalence classes
      SymSema ← ExtractConstants(ISA_Sema) // Produce symbolic semantics again
      PerformEqChecking(SymSema, EqClasses) // Perform similarity checking again
      EliminateUnnecessaryArgs(EqClasses)         // Eliminate dead arguments
      return EqClasses
  end function
  function PerformEqChecking(SymSemanticsList, EqClasses)
      for all InstSema1 in SymSemanticsList do
          for all InstSema2 in SymSemanticsList do
              if InstSema1 != InstSema2 then
                      // Verify equivalence on symbolic inputs using SMT solvers
                  if verifyEquivalence(InstSema1, InstSema2) == true then
                      EqClasses ← EqClasses ⋃ InstSema1 ⋃ InstSema2
                  end if
              end if
          end for
      end for
  end function
```

not extracted twice if they are guaranteed to have the same bitwidth, Hydride traverses the use-def chains in Hydride IR and performs a simple bitwidth analysis by accounting for legality constraints of bitvector operations. For example, if the IR code contains `%c = bvadd %a, %b`; %a, %b and %c must have equal bitwidths for the bitvector add to be legal; therefore, Hydride only extracts bitwidth of one of them only once. Figure 3(c) shows the constants extracted as parameters from the semantics of the high interleave instruction. Hydride traverses Hydride IR for each instruction with semantics $\Phi(I, \vec{k}^I)$ to extract constants $\vec{k}^I = k_1^I \cdots k_r^I$ and generate the symbolic semantics $\Sigma(I, \vec{\alpha}^I)$.

Equivalence Checking Hydride uses the following criteria to determine whether to perform similarity checking between two instructions since performing it naively between thousands of instructions would take several hours.

- # of arguments in the IR functions must be the same.
- # of arguments of bitvector type must be the same.
- # of arguments of integer type must be the same.

Hydride uses Algorithm 1 for performing similarity checking. The algorithm shows how Hydride performs similarity checking between representative instructions from each *equivalence class*, and not all individual instructions. After all the equivalence classes are created, Hydride performs further transformations discussed below.

Reorder Instruction Arguments. Instructions such as `_mm512_mask_blend_epi8` and `_mm512_mask_mov-_epi8` are semantically equivalent (same vector length for inputs and output, same element size, etc.) except the order of the vector arguments is different. So once the equivalence classes are generated by executing Algorithm 1, Hydride permutes the arguments of functions representative of each equivalence class and performs similarity checking again

(a) Intel's pseudocode documentation for _mm256_unpackhi_epi16

```
DEFINE INTERLEAVE_HIGH_WORDS(src1[127:0], src2[127:0]){
  dst[15:0]:=src1[79:64]; dst[79:64]:=src1[111:96];
  dst[31:16]:=src2[79:64]; dst[95:80]:=src2[111:96];
  dst[47:32]:=src1[95:80]; dst[111:96]:=src1[127:112];
  dst[63:48]:=src2[95:80]; dst[127:112]:=src2[127:112];
  RETURN dst[127:0];
}
dst[127:0]:=INTERLEAVE_HIGH_WORDS(a[127:0],b[127:0]);
dst[255:128]:=INTERLEAVE_HIGH_WORDS(a[255:128],b[255:128]);
```

(b) Canonicalized Semantics in Hydride IR

```
;; After loop rerolling, function inling, etc.
func _mm256_unpackhi_epi16 ( bv256 a, bv256 b ) {

  ;; Outer loop INTERLEAVE_HIGH_WORDS's call sites
  for ([%out.it (range 0 256 128)]) {
    ;; Inlined code from INTERLEAVE_HIGH_WORDS
    for ([%in.it (range 0 128 32)]) {
      %factor = div i32 %in.it, i32 2
      %offset.0 = add i32 %factor, i32 64
      %low.i = add i32 %out.it, i32 %offset.0
      %high.i = add i32 %low.i, i32 15
      ;; Extract 16-bits long slices from a and b
      %ext.a = bvextract bv256 a, i32 %low.i, i32 %high.i
      %ext.b = bvextract bv256 b, i32 %low.i, i32 %high.i
      ;; Insert slices from a and b into dst
      bvinsert bv16 %ext.a, bv256 dst, ...
      bvinsert bv16 %ext.b, bv256 dst, ...
    }
  }
  ret bv256 dst
}
```

(c) Extracted Constants

```
;;Code after extracting constants
func _mm256_unpackhi_epi16(bv a, bv b, i32 %vectsize,
    i32 %winsize, i32 %winoffset, i32 %interleavesize,
    i32 %elemsize, i32 %arg0, i32 %arg1, i32 %arg2){
  ;;Loop bounds are extracted and constants from loop body
  ;;are extracted using backwards and forwards data flow analysis.
  for([out.it (range 0 %vectsize %winsize)]){

    for([in.it (range %winoffset %interleavesize %elemsize)]){
      %ifactor = div i32 in.it, i32 %arg1
      %offset0 = add i32 %factor, i32 %arg2
      %low.i = add i32 out.it, i32 %offset0
      %high.i = add i32 %low.i, i32 %arg0
      ;; Extract 16-bits long slices from a and b
      %ext.a = bvextract bv256 a, i32 %low.i, i32 %high.i
      %ext.b = bvextract bv256 b, i32 %low.i, i32 %high.i
      ;; Insert slices from a and b into dst
      bvinsert bv16 %ext.a, bv256 dst, ...
      bvinsert bv16 %ext.b, bv256 dst, ..
    }
  }
  ret bv dst
}
```

(d) Insert hole

(e) Hole Definition for Synthesis

```
;; Definition of hole.grammar function
(define-grammar (hole.grammar x y z len)
  [expr
    (choose x y z
      (+ (expr) (expr)) (- (expr) (expr))
      (/ (expr) (expr)) (* (expr) (expr))
      ;; 0, 1, 2, 3, … (len – 1)
      (range 0 len)
)])
```

```
;;Code with hole generated by Hydride
func _mm512_unpacklo_epi8(bv256 a, bv256 b, i32 %vectsize,
    i32 %winsize, i32 %winoffset, i32 %interleavesize,
    i32 %elemsize, i32 %arg0, i32 %arg1, i32 %arg2){
  for([out.it(range0 %vectsize %winsize)]){
    for([in.it(range %winoffset %interleavesize %elemsize)]){
      %ifactor = div i32 in.it, i32 %arg1
      %offset0 = add i32 %factor, i32 %arg2
      %low.i = add i32 out.it, i32 %offset0
      ;;Hole inserted by Hydride.
      ;;Missing op in terms of out.it, in.it, %low.idx.
      %hole = call@hole.grammar(i32 out.it, i32 in.it, i32 %low.i, i32 256)
      %high.i = add i32 %hole, i32 i32 %arg0
      ;;Hole is lower index of extractops
      ;;Extract 16-bits long slices from a and b
      %ext.a = bvextract bv256 a, i32 %hole, i32 %high.i
      %ext.b = bvextract bv256 b, i32 %hole, i32 %high.i
      ;; Insert slices from a and b into dst
      bvinsert bv16 %ext.a, bv256 dst, ...
      bvinsert bv16 %ext.b, bv256 dst, ..
    }}
  ret bv dst
}
```

Fig. 3. Hydride parses ISA pseudocode in (a) to emit Hydride IR which is canonicalized using transformations such as loop rerolling, function inlining, etc. in (b). Hydride then extracts constants to produce code in (c) and inserts hole in (d) (hole defined in (e)) to synthesize the missing operation. After synthesis, `%hole = add i32 %low.i, i32 0`; where 0 is abstracted away before similarity checking.

τ_a : ArgType ::== intsz | bvsz
τ_s : FuncType ::== $(x : \tau_a) \rightarrow \tau_s$ | unit
τ : Type ::== τ_a

note: we use $\cdot*$ to mean 0 or more

```
e : BVExpr ::==  variable
               | C_BV
               | BVExpr ⊕ BVExpr
               | bvneg BVExpr
               | bvextract BVExpr
               | bvinsert BVExpr
               | bvextend BVExpr
               | bvconcat BVExpr BVExpr
               | bvcast IntExpr

i : IntExpr ::== variable
               | C_Int
               | IntExpr × IntExpr
               | ⊖ IntExpr
               | intcast BVExpr

⊕   represents binary operators for bitvector types
-   represents unary operators for bitvector types
×   represents binary operators for integer types
⊖   represents unary operators for integer types

s : Stmt ::== s;s                      sequencing
           | for x in range[e, e, e]   loops
           | if e then s  else s       conditional
           | call (e *, i*)            function call
```

Fig. 4. Abstract syntax for the Hydride IR.

to merge the equivalence classes if they are deemed to be equivalent.

Refining Equivalence Classes. Access patterns of elements of input vectors of the two instructions in Figure 2 are similar: only the offsets into the lane at which the elements in input vectors are extracted in two instructions are different, i.e. the offset is 0 for the `_mm256_unpackhi_epi16`, and it is 2 for the `_mm512_unpacklo_epi8`. Hydride's `extract` operations extract bit-slices from vectors, their operands for low and high indices dictate the coordinates of a slice extracted from a vector. The offset into the lanes for `_mm512_unpacklo_epi8` is be expressed by adding 2 to the low index for `extract` operations for the input vector (the high index is taken care of since it is expressed in terms of low index and bitwidth of a slice as: `low_index + bitwidth - 1`); however, no such addition is necessary for `_mm256_unpackhi_epi16` since the offset is 0 (Figure 3(b)). In order for Hydride to capture the similarity between the access patterns of the two instructions, Hydride inserts a hole in the semantics to add to the low indices to account for the missing operation in `_mm256_unpackhi_epi16`; the `hole.grammar` function representing such a hole is defined and invoked in Figure 3(e)). Based on this definition, Hydride synthesizes an expression of this hole in terms of inner and outer loop iterators, low index, and constant values less than bitvector length. In this case, Hydride synthesizes an `add` operation with %low.i added to 0 – this constant is extracted and similarity checking is performed

again (Algorithm 1), and this time the two instructions are deemed to be similar. This technique can also account for other access patterns such as striding access patterns.

Eliminating unnecessary arguments. Hydride conservatively extracts all constants from the semantics before checking for similarity, which leads to complex function signature with many arguments. Hydride then fixes the signatures by going over all instructions in all equivalence classes and eliminating arguments that have the same concrete value across all the instructions in an equivalence class.

Using all these techniques, Hydride's Similarity Checking Engine generates 136 equivalence classes for 2,029 x86 scalar and vector instructions, 177 classes for 1,221 ARM instructions and 115 classes for 307 HVX instructions (Table 1). Hydride forms 397 equivalence classes for x86, ARM and HVX instructions combined (3,557 hardware instructions). The results are detailed in Section 6.2.

3.4 AutoLLVM IR Instructions

Hydride uses the equivalence classes generated by the Similarity Engine to generate instructions for AutoLLVM, implemented as LLVM intrinsic functions [24]. These are retargetable instructions with parameters representing the abstracted constants, such that each combination of (constant) parameter values generates a particular target-specific instruction in the equivalence class. The signature of the function in Hydride IR is used to generate the signature of the AutoLLVM intrinsic, except that vector type information like number of elements and size of each element are folded into the LLVM vector type. Hydride automatically generates an LLVM TableGen file with definitions of all AutoLLVM intrinsics. Each of these AutoLLVM intrinsics is retargetable to the target-specific instructions in the equivalence class it represents. An example of AutoLLVM IR vector interleave and dot product instructions from compiling Matrix Multiplication kernel on x86 is shown below.

```
%0 = call <32xi16> @autollvm.interleave(<16xi16> %arg1,
     ↪ <16xi16> %arg3, 0 /*lane offset*/)
%1 = call <32xi16> @autollvm.interleave(<16xi16> %arg2
     ↪ ,<16xi16> %arg4, 8 /*lane offset*/)
%2 = call <32xi16> @autollvm.dotproduct(<32xi16> %arg0,
     ↪ <32xi16> %0, <32xi16> %1)
```

3.5 Low-level Code-Gen Generator

This component enables Hydride to automatically generate parts of code generation support using the information about similar instructions (both inter-architectural and intra-architectural) from the Similarity Checking Engine. The Code-Gen Generator consists of the following components.

Rosette-to-LLVM Translator. The code synthesized by Hydride's Code synthesizer (Section 4) is Rosette code with target-agnostic instructions represented as opaque function calls. The Rosette-to-LLVM Translator translates the synthesized code to AutoLLVM IR instructions.

Pattern-matching-based, Target-specific Code Generator. The Hydride Code-Gen generator automatically generates simple, 1-1 pattern matching rules that translate AutoLLVM IR instructions to existing target-specific intrinsics in LLVM IR. Hydride is able to do this because it keeps track of original values of all the instruction-specific quantities that were abstracted away to form AutoLLVM IR instructions. For example, this helps generate x86 instructions for AutoLLVM IR code above shown in Row 2 of Table 3.

4 Hydride Code Synthesizer

The Hydride Code Synthesizer is the second major component of Hydride. It enables automatic generation of a front-end translator to AutoLLVM instructions, thereby eliminating the need for target-specific back ends in front-end compilers. It leverages the AutoLLVM IR semantics generated by Hydride's Similarity Checking Engine.

We designed a prototype front-end for the Halide DSL, creating a compiler from Halide to LLVM using Hydride. Like Rake [1], our front-end takes as input Halide IR lowered from an input Halide program after all scheduling optimizations have been applied, including vectorization, parallelization and tiling, and synthesizes an equivalent target-specific program. However, Hydride provides two major improvements over Rake, both of which are important for practical usage. First, Hydride synthesizes an equivalent target program from thousands of potential ISA operations, instead of just a few. The results of the offline stage of Hydride are essential to making the synthesis scalable enough for this to be practical, which occurs in two distinct ways: (1) The automatic generation of the target-specific and AutoLLVM instruction semantics is essential to target multiple large instruction sets, like x86 and ARM, with thousands of instructions. (2) The results of similarity checking in the offline stage are useful in identifying "potentially relevant instructions" and then pruning instructions from those subsets by using their similarity. Second, Hydride explores greater expression depths in the input Halide code compared with Rake, which enables it to generate complex non-SIMD operations like dot-product automatically, instead of requiring manual changes to the algorithms in Halide source programs, as needed in Rake.

4.1 Automatically Generated Synthesis Framework

To perform synthesis, Hydride's Code Synthesizer consumes the automatically generated dictionary of AutoLLVM Instructions and creates a Syntax Guided Synthesis (SyGuS) framework in Rosette. Rosette's SyGuS functionality requires users to provide formal semantics of the input and output languages, an input program, and a grammar for the output programs. The synthesis algorithm then searches for a candidate program in the grammar which is equivalent to the input program. Importantly, this synthesizer is completely independent of the target ISA and also of the source language

(i.e., it is not specific to Halide): the target details are captured by the dictionary and the source language is captured by a language semantics, both of which are inputs to the synthesizer. Using these inputs, it automatically generates a number of components in an offline stage:

- **Interpreter** : Implements the operational semantics of the AutoLLVM IR operations defined by the Hydride's IR Generator in Rosette; these semantics are used during synthesis.
- **Cost Model** : Defines cost model that maps AutoLLVM IR expressions to an integer cost value, using a simple sum of the individual latencies in an expression.
- **Grammar Generator** : Generates a grammar to drive the SyGuS approach for each sub-expression in the input program. This grammar includes only potentially relevant AutoLLVM IR instructions in order to make synthesis cost tractable. The grammar uses a bounded, parameterized depth to ensure scalability (Section 4.3).
- **Memoization Cache** : Records synthesis results for each input expression to enable reuse of these results.

4.2 Synthesis Strategy

Hydride uses Counterexample Guided Inductive Synthesis (CEGIS) [22], a capability added by us for Hydride, to compile Halide IR operations to AutoLLVM IR while minimizing an objective cost function. The Halide IR program is first lowered into an equivalent expression in Rosette, using the semantics of Halide IR from [1]. To keep synthesis feasible, Hydride extracts sub-expressions (which we call windows) of bounded depth from the Halide IR program and synthesizes equivalent code in AutoLLVM IR. The window size and maximum output sequence size are parameters in Hydride. Hydride incrementally increases the size of the output sequence (i.e. the depth of the synthesis grammar) when searching for minimum-cost expressions. This leads Hydride to favor shorter instruction sequences and ones that take less time to execute, estimated by summing target instruction latencies. Additionally, Hydride leverages the parameterization of the AutoLLVM IR to uniformly scale (not truncate) the number of lanes in the vector ISAs for synthesis. Solver time complexity grows exponentially with the sizes of the bitvectors, and so reducing the sizes of the bitvectors enables synthesis to be tractable for targets such as HVX which can have 2048-bit vectors.

Algorithm 2 shows the iterative synthesis procedure Hydride uses. Before invoking this procedure for an expression, *Expr*, Hydride automatically generates a *pruned* grammar, *G*, tailored to the expression, as described in Section 4.3.

Starting with a seed set of concrete inputs (line 4) , and output vector lanes to verify equivalence (line 5), Hydride generates constraints that the synthesized program should produce the equivalent output as the input program only for the vector-lanes in *Failing-Lanes*. A common paradigm in

Algorithm 2 CEGIS for compiling a single expression

Input: Grammar G, Specification Expr, CostModel C, ScaleFactor Sc
Output: Program Sol s.t. $\forall x.Sol(x) = Expr(x) \wedge C(Sol)$ is minimized.

```
1:  function LANEWISE_SYNTHESIS(G, Expr, C, Sc)
2:      ScaledExpr ← ScaleDown(Expr, Sc)        // Scale down number of lanes
3:          // Create initial set of concrete inputs for synthesis
4:      CEX ← {RandomInputs(ScaledExpr), RandomInputs(ScaledExpr)}
5:      Failing-Lanes ← {0, 0}                  // Test both inputs on lane 0
6:      do
7:          asserts ← EqualOnLane(G, ScaledExpr, CEX,Failing-Lanes)
8:              // Synthesize solution that satisfies asserts & minimizes Cost C
9:          Sol ← Optimize(G, C, asserts)
10:         if Sol = unsat then
11:             G′ ← IncrementGrammarDepth(G)
12:             return LANEWISE_SYNTHESIS(G′, Expr, C, Sc)
13:         end if
14:             // Verify symbolically, returning counter example
15:         CEX′ ← VerifiedEqv(Sol, ScaledExpr)
16:         if CEX′ ≠ {} then
17:             CEX ← CEX ⋃ CEX′
18:                 // Diff[i] = 1 if Sol(CEX')[i] ≠ ScaledExpr(CEX')[i],else 0
19:             Diff ← DifferencePredicate(Sol, ScaledExpr, CEX′)
20:             Failing-Lanes ← Failing-Lanes ⋃ FirstNonZeroIdx(Diff)
21:         end if
22:     while CEX′ ≠ {}
23:     if VerifiedEqv(ScaleUp(Sol, Sc), Expr) = {} then
24:         return ScaleUp(Sol, Sc)
25:     end if
26:     return LANEWISE_SYNTHESIS(G, Expr, C, 1)
```

vector ISAs is that a pattern of computation repeats across certain sets of lanes: for SIMD operations the pattern repeats for every lane; and for non-SIMD operations it repeats every 2, 4, etc lanes. Hydride leverages this repetition to synthesize programs using constraints over a subset of the output vector lanes, as the computation pattern is repeated across intervals in the vector lanes. The constraints are represented as assertions (line 7). Then Hydride synthesizes programs which are equivalent to the specification under these constraints while minimizing an objective cost function C (line 9). If the solver returns *unsat* (line 10), Hydride re-attempts synthesis with larger grammar depth (line 12). Otherwise, Hydride verifies equivalence of the synthesized program with the input program for **all** lanes jointly with symbolic inputs (line 15). If verification succeeds, then the desired solution is achieved, and the algorithm exits the synthesis loop (line 6 – 22). Otherwise, the solver returns a counterexample (line 16). For any counterexample identified by the verifier, Hydride identifies the failing lane for which the output vectors differ (line 19-20) and appends it to the next iteration of constraints. Synthesis is then repeated (loop from line 6 – 22) with these additional constraints. Once an expression is synthesized, Hydride scales the expression back up (i.e increases the number of vector lanes to the original amount) and verifies its correctness against the specification. If verification finds a counterexample, Hydride repeats the procedure without scaling (line 26).

Therefore, Hydride optimizes the synthesis process across vector lanes in two ways: **(1)** Constraining synthesis over only a subset of lanes, and verifying the synthesized expression over the rest of the lanes. **(2)** Scaling down the number of vector lanes for AutoLLVM IR operations for synthesis,

which in turn reduces the bitvector sizes for the operands and results, and then scaling the number of lanes back up.

4.3 Pruned Grammar Generation

Synthesizing code for a given input expression for large ISAs such as x86, with thousands of instructions, can be intractable because synthesis has doubly exponential complexity. Therefore, another heuristic in Hydride to improve tractability and speed up synthesis is to generate multiple versions of *pruned* grammars and synthesize in parallel; in other words, generate multiple smaller grammars with different subsets of AutoLLVM IR instructions, instead of one complete one. Following steps describe how pruned grammars are generated.

(a) Hydride omits an entire equivalence class (described in Section 3.3) if *either* none of the operations (e.g., bvmul, bvadd) in an equivalence class matches any operation in the input expression, *or* none of the instructions in an input expression for the given target uses a vector length and element size supported by the equivalence class.
(b) Hydride also eliminates any instruction that uses a smaller element size than the minimum element size in the input expression because generating such an instruction would lead to information loss.
(c) Some viable instructions are more likely to be appropriate than others for a given input expression – depending on the number of matching bitvector operations, and whether an instruction's vector lengths and element sizes match with the input expression. AutoLLVM IR operations which satisfy more of the above properties are assigned a higher score value. Hydride selects the top scoring k ops from each equivalence class in the final grammar, where k is a parameter for the Hydride code synthesizer (we use k between 3-4 in our evaluation). The rationale for this heuristic is that all of the operations within an equivalence class perform similar computations, so Hydride chooses only those that have most in common with the input expression. The distribution of *compute* operations (e.g. arithmetic operations) is balanced with type-conversion operations (e.g., broadcast, sign-extend or truncate operations). Swizzle operations (Section 4.4) are always included independently of k.

4.4 Synthesizing Sequences of Swizzle Instructions

After identifying viable non-swizzle instructions for the pruned grammar, Hydride adds specialized swizzle patterns to the grammar, instead of adding a single general permute operation, which causes the synthesis to become intractable. This approach enables synthesis to be practical and facilitates the use of more complex non-SIMD vector operations. Hydride supports the following swizzle patterns:

1. **Full Interleave Two Vectors**: interleaves elements of two vectors of same size.
2. **Interleave/De-interleave Single Vector**: interleaves/de-interleaves the first and second halves of a vector.
3. **Interleave First/Second Half of Two Vectors**: interleaves the first/second halves of two vectors of same size and produces a vector of same size.
4. **Concatenate First/Second Half of Two Vectors**: Concatenates the first/second halves of two vectors of same size and produces a vector of same size.
5. **Vector Rotate-Right**: Right-rotates the order of elements of vector.

These swizzles are common in modern hardware architectures; however, if a target architecture uses a novel swizzle pattern, it would have to be added to Hydride manually.

5 Hydride Implementation

The pseudocode parsers for x86, HVX and ARM for Hydride IR generation are all implemented in Python. To increase confidence in the generated ISA semantics, we use random fuzz testing for individual instructions and compare the results of machine-executable semantics in Hydride IR against target-specific C builtins on randomly-generated inputs. Specifications for x86 and HVX instructions do not clearly distinguish between logical and arithmetic right shifts, and do not make explicit the need to widen operands while performing saturating left shifts, and so we manually modified the vendor's pseudocode specifications in such cases. Hydride's Similarity Checking Engine is also implemented in Python but it generates Rosette code for checking if semantics for two given instructions are equivalent for symbolic bitvectors in Racket. Hydride generates semantics of the target-agnostic AutoLLVM IR instructions and maps the instructions to target-specific instructions using a Python dictionary. We reuse Halide IR semantics (in Rosette) from Rake [1]. Hydride Code Synthesizer is implemented in Python and Rosette. The memoization cache for the synthesizer is implemented as a hash table in Racket, the host language of Rosette.

6 Evaluation

We evaluate Hydride against the state-of-the-art, production Halide's target-specific back ends [11] and Halide's LLVM Back end [14] and Rake [1] across 33 Halide benchmarks from image processing and deep learning domains on three targets: x86, Hexagon and ARM. To compare against Halide's LLVM Back end, we let Halide emit LLVM IR (with optimizations such as loop unrolling, tiling, etc. applied) and apply optimizations on LLVM IR such as loop vectorization, Superword-Level Parallelism Vectorization, Aggressive Instcombine, etc. These benchmarks have been hand-tuned by us for x86, and by Qualcomm and Adobe for ARM and HVX. The benchmarks include Matrix Multiplication on tensors of

low batch sizes (1, 2 and 4) which have low arithmetic density and commonly found in large language models; and we also evaluate some fused versions of deep learning kernels that are commonly found in various neural networks (such as average/max pool + add), and in MLP blocks, in particular (matmul + bias + activation + matmul). For all experiments, we target x86 with an Intel Xeon Silver 4216 CPU (16 cores, 2.1GHz, 22 MB L3 cache) with hyperthreading disabled; HVX with a cycle-accurate simulator in Hexagon SDK v3.5.2 provided by Qualcomm; ARM with an Apple M2 CPU (3.49GHz, 16 GB memory, 16 MB L3 cache).

6.1 Case Study: Support for ARM in Hydride

x86 and Hexagon were the first two targets Hydride supported. The design and implementation of Hydride took a little over a year, with two students (first two authors) responsible for implementing the major components. After a year long effort, Hydride had proven to perform competitively with the state-of-the-art production Halide compiler, and significantly better than Halide's LLVM Back end (with additional LLVM IR optimizations) on x86 and Hexagon architectures (as shown in Figs. 6a and 6b, and discussed later, below). ARM support was added by a third student who was not part of the project and was completely unfamiliar with the implementation of Hydride. Without much guidance or documentation, he implemented a parser for ARM pseudocode specification from [2], integrated it into Hydride, and added support for ARM in Hydride in nearly 3 months, and achieved performance nearly the same as the production Halide, as shown in 6c. Companies like Google, Adobe, and Qualcomm have invested in large engineering teams over the past decade to develop, maintain and (in significant part) heavily optimize the code generation support in the production Halide compiler: the earliest Github commits to Halide's x86 back end were made on November, 2013 (10 years); the HVX back end in March, 2016 (8 years); the ARM back end in February, 2013 (over 10 years). The fact that Hydride is able to approximately match Halide's performance across real-world benchmarks for three complex architectures with the engineering effort of three people in little over a year is a testimony to the degree of automation and retargetability Hydride provides, and significant reduction in engineering effort and time it achieves.

6.2 AutoLLVM IR Results

Table 1 shows the number of instructions Hydride supports for each architecture and the number of AutoLLVM IR instructions it generates for each of them individually and for all of them combined. HVX has fewer equivalence classes than the other two since it is a much smaller, and more specialized, instruction set for DSP; therefore, HVX has fewer common equivalence classes with x86 and ARM. ARM has few common equivalence classes with x86 and HVX because

Table 1. AutoLLVM IR results for each architecture.

Architecture	ISA Size	AutoLLVM Size	IR Size % of ISA Size
x86	2,029	136	6.7%
HVX	307	115	37.5%
ARM	1,221	177	14.5%
x86 + HVX	2336	232	9.9%
x86 + ARM	3250	302	9.3%
HVX + ARM	1528	286	18.7%
x86 + HVX + ARM	3,557	397	11.2%

Table 2. Bugs found in Rake. Operations are abbreviated as ARS: Arithmetic Right Shift, LS: Left Shift.

Files	Lines	Bug Description
halide/ir/interpreter.rkt	536	Semantics of ARS not masked.
hvx/interpreter.rkt	1146	ARS' operands not masked.
hvx/interpreter.rkt	1163	Rounding/Saturating ARS not masked.
hvx/interpreter.rkt	795	LS operands not masked.
hvx/interpreter.rkt	802	fused LS and accumulate not masked.

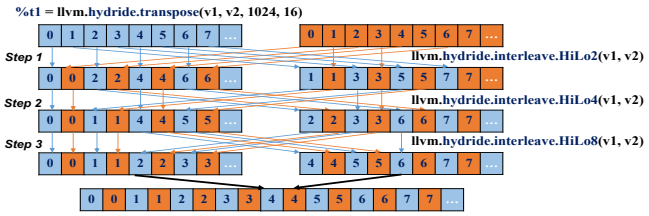


Fig. 5. Hydride generates this complex HVX instruction that performs 2x2 block transpose between vectors. x86 has no such instruction, but data layout transformations shown in each step can be achieved using x86's interleaving instructions.

significant number of ARM instructions are fused instructions (e.g., fused SIMD add and subtraction instructions) that do not exist in x86 and HVX.

Rake only supports 164 HVX and 200 ARM instructions; Hydride supports ~2x more HVX and ~6x more ARM instructions than Rake. Rake requires implementing code generation to target instructions by manually implementing semantics of target instructions, which is an onerous, cumbersome and time-consuming process. Table 2 lists bugs we found in Rake's implementation of HVX semantics. Hydride, however, automatically generates semantics for ISAs once the parsers for ISA pseudocodes are implemented; so this is a one-time effort. Neither Rake nor Hydride support synthesis of memory instructions such as loads and stores. Hydride supports more complex instructions than Rake in HVX: Hydride supports and often generates `vshuffvdd` (shown in Figure 5 as composition of x86 instructions) and `vdealvdd` instructions to effectively transpose huge vectors. Hydride supports several variants of dot product and swizzle instructions that are not supported by Rake; these are critical for high-performance code and result in speedups against Rake (Figure 6b). Like Rake, Hydride only supports integer instructions; floating-point is left to future work.

6.3 Runtime Performance

Performance on x86. Hydride outperforms Halide's production x86 back end by a geomean of 8% and Halide's LLVM back end for x86 by 12% (Figure 6a). It gets significant speedups of over 1.30x over Halide's x86 back end and 1.42x over Halide's LLVM Back end for Matrix Multiplication benchmarks – Row 2 of Table 3 shows the faster non-SIMD (dot product and interleave) instructions Hydride generates vs the slower SIMD code that Halide's x86 Back end generates. Hydride performs at least as well as Halide's x86 back end on 31 benchmarks (speedups up to 1.35x) including convolution benchmark on which Hydride gets a modest speedup of 4%, again, due to generation of slightly better dot product instructions (Row 3 in Table 3). Hydride gets small slowdowns on 2 benchmarks, by 5% on add and 4% on softmax, because LLVM's x86 back end (which is used in Hydride) pattern-matches specialized swizzle instructions to higher-latency x86 permute instructions. Hydride performs just as well as or outperforms Halide's LLVM Back end (maximum speedup of 81%).

Performance on Hexagon. Hydride performs just as well as Halide's Hexagon Back end and outperforms Halide's LLVM Back end on Hexagon by geomean of ~2x (Figure 6b). Hydride performs at least as well as Halide on 17 benchmarks (speedups up to 1.36x), and performs within 15% of Halide on 14 benchmarks. Hydride suffers from more serious slowdowns relative to Halide on 2 benchmarks: gaussian7x7 (0.54x) and conv3x3a16 (0.78x). For gaussian7x7, Halide supports specialized patterns that enable it to analyze and pattern-match code in a large window of instructions spanning multiple basic blocks and generate a four-way dot product `vrmpy` instruction, which Hydride fails to generate because the window to synthesize code for is too large for the synthesis to be tractable. Despite generating similar, and in some cases better, non-SIMD (i.e., cross-lane) code than Halide in other benchmarks including matrix multiplication (Row 1 of Table 3), Hydride gives slight slowdowns because Hydride is unable to move intermediate pair of interleave and deinterleave instructions across multiple basic blocks closer and eliminate them, whereas Halide deploys a specialized optimization pass in HVX back end. Hydride outperforms Halide's LLVM Back end significantly on all benchmarks with maximum speedup of 81% on blur3x3. We could not compile baseline benchmarks for convolution and benchmarks involving GeLU because the register allocation in LLVM's Hexagon back end fails.

We also compared Hydride against Rake [1]. Hydride outperforms Rake on all the benchmarks we could evaluate (it failed to compile 28 benchmarks), with geomean speedup of 25% and maximum speedup of 80% on average pool, with two exceptions. We see small slowdowns on add (5%) and max pool (8%) because the LLVM's HVX back end used by Hydride generates machine code with more register spills in these cases.

Performance on ARM. Hydride achieves a small geomean speedup of 3% over Halide's production ARM back end and 26% over Halide's LLVM back end on ARM (Figure 6c). Hydride gets significant speedups relative to the former on blur5x5 (28%) and blur7x7 (26%). It gets small slowdowns of less than 5% on 4 benchmarks and 10% on gaussian7x7. Hydride generally outperforms Halide's LLVM back end (maximum speedup of 5.53x) across all benchmarks. Rake purports to support ARM, but fails to successfully compile any benchmark. Matching the performance of the production Halide's ARM back end and outperforming Halide's LLVM back end decisively is an impressive result for roughly three months of work by one student.

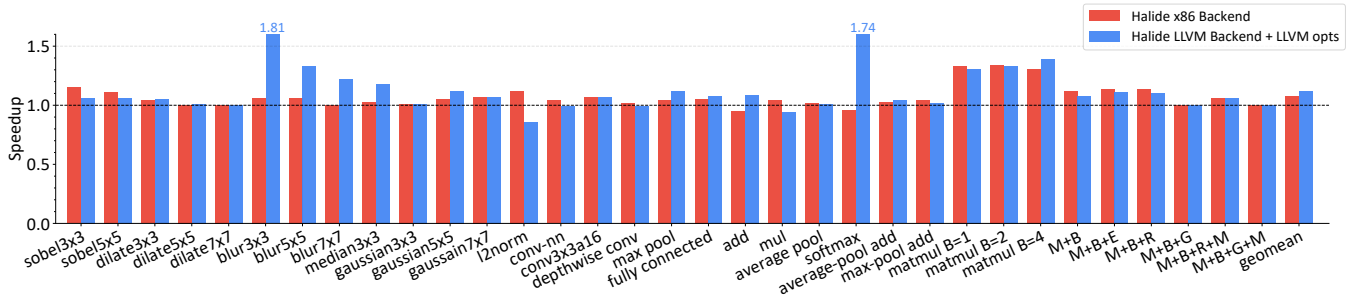
6.4 Compile Times

Table 4 shows the compilation times for benchmarks to compile with Hydride.

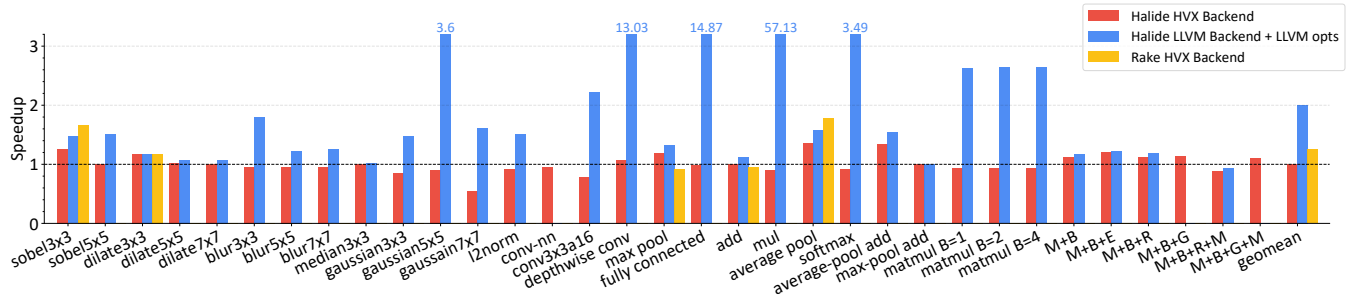
Column I shows synthesis times when starting with an empty memoization cache (Section 4.1), i.e., Hydride starts synthesis afresh for each benchmark. Synthesis takes a geomean time of 9 min. for ARM, 18 min. for X86, and 39 min. for HVX. Synthesis takes longer for HVX because of the complexity of very specialized instructions. Longer synthesis times allow for the possibility that complex instructions can be generated, important for HVX, which potentially perform better than simpler SIMD instructions sequences (i.e., that are fully independent across lanes). This is why Hydride gets large speedups against Halide's LLVM back end, which only generates these simpler SIMD instructions. Note, however, that longer synthesis times do not guarantee better performance: we see a slowdown on gaussian7x7 despite one of the highest synthesis times.

To evaluate the potential benefits of caching on compilation of new benchmarks, we evaluated compilation times when the memoization cache has been populated with synthesis results for all other benchmarks except the benchmark being compiled. The results (Column II) show that caching greatly speeds up compilation, with geomean times of ~7.5, ~9 and ~11 min for ARM, x86 and HVX, signifying significant commonality in input subexpressions across benchmarks.

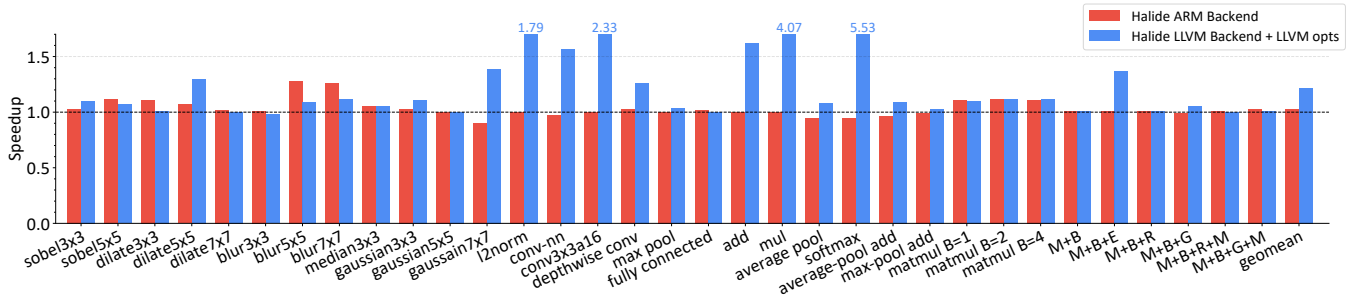
Column III represents the best-case scenario (recompiling a benchmark twice, e.g., when schedules are not changed); it shows recompilation times when memoization cache is already populated with synthesis results, so resynthesis is unnecessary. These times are far lower: ~1.5, ~2.5 and ~2 min on ARM, x86, and Hexagon. These times are mostly dominated by cache lookups for synthesis results, which is slow because of Racket's inefficient hash-table implementation and high initialization overhead (Racket initializes every time it is invoked to synthesize code). To quantify this overhead, we compiled an empty program, then multiplied that by the min, median and max number of expressions being



(a) Performance on x86



(b) Performance on HVX



(c) Performance on ARM

Fig. 6. Performance of Hydride against Halide's target-specific back ends, Halide's LLVM back end and Rake's HVX back end. Fused kernels are abbreviated as M: Matmul, B: Bias Add, R: ReLU, G: GeLU and E: Element-wise Add.

compiled across our benchmarks to account for repeated invocations. The overheads (Table 4) are as high as 5.5–60 minutes for the largest cases. A fast language like C++ would greatly reduce cache lookups times.

Halide, the DSL, separates algorithms (computations) from schedules (loop transformations such as tiling, unrolling, etc.). Halide programmers often tweak the schedules by changing order of schedules, tiling, loop unroll factors, etc., without having to modify the algorithms. Column IV represents a more common and realistic scenario, where programmers tune the schedules of their programs to optimize for different shapes of input tensors, except for vectorization factors (the maximum length of the target register). We observe that in this scenario, as long as vectorization factors and the loops that are vectorized remain unchanged, Hydride will not need to resynthesize code every time programmers modify their schedules. Geomean compilation times after modifying tiling, rerolling factors, loop fusion strategies, etc. in our original benchmarks is ~3 minutes on x86, ~1.5 minutes on HVX and ~4 minutes on ARM – which are close to when compiling with full cache (Column III). with Hydride, programmers only need to synthesize with a vectorization factor once; from then on Hydride can keep reusing the synthesis results from the cache regardless of how other schedules are changed. This works because schedules primarily affect memory access patterns, data locality, etc. and Hydride only handles generation of compute and swizzle instructions and not memory instructions.

Table 3. Efficient complex, non-SIMD code generated by Hydride relative to simpler, SIMD code by Halide

Halide IR Expression	Halide Generated Code	Hydride Synthesized Code

`// Matmul (HVX)`
`%0 = <64xi32> cast-int %arg1`
`%1 = <64xi32> cast-int %arg2`
`%2 = <32xi32> mul %0[0:32] %1[0:32]`
`%3 = <64xi32> cast-int %arg3`
`%4 = <64xi32> cast-int %arg4`
`%5 = <32 x i32> mul %3[32:64] %4[32:64]`
`%6 = <64 x i32> concat %2 %5`
`%7 = <64 x i32> sat-add %6 %arg0`
	`// Cost: 18`
	`// Perform Sign Ext`
	`%0 = <32 x i32> unpack_vh %arg1[0:32]`
	`%1 = <32 x i32> unpack_vh %arg2[0:32]`
	`// 32-bit mul using 16-bit shift & add`
	`%2 = <32 x i32> vmpyieoh %0 %1`
	`%3 = <32 x i32> vmpyiewuh_acc %2 %0 %1`
	`// Repeat for [32:64] slice`
	`%8 = <64 x i32> vcombine %3 %7`
	`%9 = <64 x i32> vaddwsat_dv %arg0 %8`
		`// Cost: 13`
		`// Sign Ext with interleaving elements`
		`%0 = <64 x i32> vsh %arg1`
		`%1 = <64 x i32> vsh %arg2`
		`// Cross lane 2-point reduction with saturating acc`
		`%2 = <32 x i32> vdmpyhvsat_acc %arg0 [0:32] %0[:32] %1[:32]`
		`// Repeat for [32:64] slice`
		`%6 = <64 x i32> vcombine %2 %5`

`// Matmul (x86)`
`%0 = <16 x i32> cast-int %arg1`
`%1 = <16 x i32> cast-int %arg2`
`%2 = <16 x i32> mul %0 %1`
`%3 = <16 x i32> cast-int %arg3`
`%4 = <16 x i32> cast-int %arg4`
`%5 = <16 x i32> mul %3 %4`
`%6 = <16 x i32> add %2 %5`
`%7 = <16 x i32> add %arg0 %6`
	`// Cost: 34`
	`// Elementwise Sign-Ext from 16 to 32 bits`
	`%0 = <16 x i32> cvtepi16_epi32 %arg1`
	`... // "sequence of sign-ext instructions".`
	`// Elementwise mul to produce 64 bits,`
	`// then take lower 32 bits`
	`%4 = <16 x i32> mullo_epi32 %0 %1`
	`...`
	`// Elementwise addition`
	`%6 = <16 x i32> add_epi32 %4 %5`
	`...`
		`// Cost: 25`
		`// Interleaving swizzle to layout elements`
		`%0 = <16 x i32> cvtepu16_epi32 %arg1`
		`... // "sequence of sign-ext instructions".`
		`%4 = <32 x i16> packus_epi32 %0 %1`
		`%5 = <32 x i16> packus_epi32 %2 %3`
		`%6 = <32 x i16> unpacklo_epi16 %4 %5`
		`%7 = <32 x i16> unpackhi_epi16 %4 %5`
		`// two point reduction with sign-ext and acc.`
		`%8 = <16 x i32> dpwssd %arg0 %6 %7`

`// Conv_nn (x86)`
`%0 = <32 x i32> cast-int %arg0`
`%1 = <32 x i32> cast-int %arg1`
`%2 = <32 x i32> mul %0 %1`
`%3 = <16 x i32> reduce-add %2 2`
`%4 = <16 x i32> add %3 %arg2`
	`// Cost: 6`
	`// 2-point reduction without acc`
	`%0 = <16 x i32> madd_epi16 %arg0 %arg1`
	`// Accumulate seperately`
	`%1 = <16 x i32> add_epi32 %0 %arg2`
		`// Cost: 5`
		`// 2-point reduction with acc`
		`%0 = <16 x i32> dpwssd %arg2 %arg0 %arg1`

6.5 Synthesis Sensitivity Analysis

Because of the large sizes of individual instruction sets, adding all target instructions to the synthesis grammar renders synthesis intractable and leads to an out-of-memory exception. Hydride's Code Synthesizer employs multiple heuristics to make synthesis feasible and enable highly performant code generation across multiple targets described in Sections 4.2 and 4.3 .

Table 5 describes the effect of toggling these heuristics on synthesis times when generating the dot-product operations for x86, HVX and ARM. Including only 50 target instructions using Hydride scoring heuristic (Section 4.3 (c)), synthesis exceeds the timeout of 4 hours without terminating. Therefore, more aggressive pruning heuristics are needed to make synthesis complete in a reasonable amount of time.

Heuristics described in Section 4.3 (a) and (b) (together signified as ***BVS*** (*bitvector-based screening*) in Table 5) are the most basic heuristics that detect mismatches between characteristics of the constituent bitvector operations (in terms of operation type: bvadd, bvsub, etc.) in instruction semantics and an input expression to eliminate an equivalence class from a grammar. These heuristics cause synthesis to terminate in hundreds of seconds while producing the desired dot product expression and this is used a baseline for measuring relative speeds up achieved with Hydride's other heuristics in Figure 7.

Impact of lane-wise synthesis. As described in Section 4.2, computation patterns in vector operations repeat across lanes, and Hydride generates constraints on a few set of lanes of an output vector and synthesizes a program that satisfies equality with the input expression on those lanes only; and then it verifies the correctness of the synthesized expression over all lanes (which takes negligible amount of time). This heuristic speeds up synthesis for x86 by 2x; ARM by 1.4x; and HVX by 2.8x.

Impact of scaling. Synthesis complexity grows exponentially with register size; therefore, reducing register size reduces synthesis times, regardless of the target architecture (described in Section 4.2). Its impact is most significant for HVX which effectively uses 1024- and 2048-bit registers, whereas x86 and ARM use relatively smaller registers.

Impact of scaling + lane-wise synthesis. Both scaling and lane-wise synthesis together speed-up synthesis by 2x for x86, 12.8x for HVX and 3.6x for ARM.

Impact of score-based equivalence class ops selection (SBOS). As described in Section 4.3 (c), after applying all the aforementioned heuristics, Hydride selects operations from equivalence classes which have the greatest number of bitvector operations that are similar to those in an input expression, and this leads to a further reduction in grammar size. This heuristic, along with other heuristics, speeds up synthesis by 2.7x for x86, 20.8x for HVX and 6x for ARM.

Table 4. Compilation times with Hydride on x86, HVX, and ARM.

	I			*II*			*III*			*IV*		
	Compilation Times in seconds *(# of Expressions)*			**Compilation times of nth benchmark (s)**			**Compilation times with full cache in (s)**			**Compilation times after modifying schedules (s)**		
Benchmark	*x86*	*HVX*	*ARM*	*x86*	*HVX*	*ARM*	*x86*	*HVX*	*ARM*	*x86*	*HVX*	*ARM*
cache lookup overhead (MIN)	15 *(1)*	6 *(2)*	10 *(4)*									
cache lookup overhead (MEDIAN)	50 *(18)*	25 *(10)*	103 *(48)*									
cache lookup overhead (MAX)	880 *(409)*	330 *(136)*	2449 *(1296)*									
sobel 3x3	8300 *(4)*	4740 *(6)*	590 *(64)*	2469	4655	590	103	54	420	62	55	366
sobel 5x5	10511 *(80)*	9171 *(10)*	332 *(52)*	11037	3599	326	1060	60	279	459	101	952
dilate 3x3	90 *8*	80 *(6)*	251 *(32)*	91	80	148	74	30	108	48.5	46	107
dilate 5x5	90 *1*	120 *(6)*	86 *(4)*	34	77	20	39	45	16	36	42	15
dilate 7x7	45 *1*	120 *(12)*	41 *(8)*	35	79	38	30	53	34	50	39	31
box blur 3x3	724 *(198)*	450 *(4)*	82 *(14)*	957	105	50	500	30	42	169	40	112
box blur 5x5	943 *(203)*	226 *(22)*	8832 *(296)*	1062	140	5135	500	60	764	264	191	1072
blur 7x7	1155 *(210)*	6900 *(49)*	8274 *(784)*	1216	5200	8903	500	222	2	408	173	1922
mmedian 3x3	429 *(20)*	960 *(42)*	7247 *(20)*	422	800	6691	120	120	2	340	178	4
gaussian 3x3	2600 *(12)*	11760 *(4)*	133 *(48)*	2292	7875	168	78	60	125	54	72	111
gaussian 5x5	5326 *(8)*	10800 *(44)*	776 *(48)*	3987	10768	1371	70	150	139	41.5	71	143
gaussian 7x7	12041 *(16)*	39480 *(7)*	1574 *(268)*	5846	29460	1963	144	240	685	101	214	678
l2norm	6000 *(22)*	20600 *(7)*	7068 *(1296)*	1540	8077	5327	106	60	4181	186	75	8089
conv_nn	22000 *(284)*	54000 *(136)*	18270 *(179)*	15000	46892	16966	1221	628	581	1732	869	1829
conv3x3a16	23940 *(16)*	25200 *(31)*	304 *(128)*	23110	14799	395	263	146	317	97	159	302
depthwise conv	11000 (409)	60274 *(49)*	1672 *(361)*	4684	39000	1210	1302	525	957	4785	430	6624
average pool	640 *(9)*	4487 *(2)*	203 *(40)*	49	4860	125	91	30	103	150	64	292
max pool	100 *(2)*	68 *(7)*	58 *(14)*	58	30	52	43	30	40	89	39	136
fully connected	8563 *(11)*	36000 *(37)*	727 *(57)*	4002	3861	4877	206	160	163	292	210	148
add	2261 *(30)*	9480 *(6)*	259 *(120)*	2007	3752	535	141	79	331	284	53	514
mul	6000 *(78)*	45000 *(6)*	2210 *(120)*	484	40896	2101	307	600	330	624	350	1310
softmax	4925 *(256)*	14000 *(72)*	3107 *(720)*	2415	8100	3203	300	400	1777	670	1411	3633
matmul [batch size = 1]	125 *(16)*	500 *(4)*	97 *(16)*	129	47	57	81	60	45	83	76	83
matmul [batch size = 2]	125 *(16)*	500 *(4)*	97 *(16)*	129	47	57	81	60	45	83	76	83
matmul [batch size = 4]	125 *(16)*	500 *(4)*	97 *(16)*	129	47	57	81	60	45	83	76	83
average pool+add	3500 *(9)*	5000 *(2)*	15277 *(40)*	3193	36	4790	59	40	104	59	46	298
max pool+add	40 *(2)*	1000 *()*	83 *(8)*	60	300	70	30	20	25	34	25	86
matmul + bias	198 *(32)*	500 *(12)*	142 *(32)*	110	82	104	100	50	83	95	100	124
matmul + bias + relu	228 *(32)*	1000 *(12)*	188 *(32)*	130	90	104	120	55	86	126	65	158
matmul + bias + gelu	2980 *(48)*	450 *(20)*	815 *(192)*	200	90	785	207	68	462	1020	80	256
matmul + bias + add	227 *(32)*	1000 *(12)*	174 *(32)*	110	300	105	100	55	89	273	92	79
matmul + bias + relu + matmul	6274 *(80)*	350 *(44)*	868 *(320)*	300	170	994	278	110	725	417	200	379
matmul + bias + gelu + matmul	3100 *(80)*	300 *(44)*	848 *(320)*	300	175	999	273	100	729	208	91	394
Geomean (33 benchmarks)	1116.9 *(23)*	2348.1 *(11)*	529 *(64)*	520	673.0	455	151.9	81.1	133	179.2	98.5	252

Table 5. Synthesis sensitivity analysis for x86, HVX, and ARM. ***BVS:*** Bitvector-based screening eliminates operations in an equivalence class from a grammar if the characteristics of the constituent bitvector operations do not match that of an inpput expression. ***SBOS:*** Score-based selection of operations from an equivalence class to add to the grammar for synthesis.

Synthesis Setting	**x86**		**HVX**		**ARM**	
	# of Operations in Grammar	*Synthesis times (seconds)*	*# of Operations in Grammar*	*Synthesis times (seconds)*	*# of Operations in Grammar*	*Synthesis times (seconds)*
All target instructions	2029	Intractable	307	Intractable	1221	Intractable
Top 50 instructions based on score	50	14400+	50	14400+	50	14400+
BVS	26	236	24	997	25	628
BVS + lane-wise	26	118	24	360	25	452
BVS + scaling	26	142	24	108	25	165
BVS + scaling + lane-wise	26	115	24	78	25	175
BVS + scaling + lane-wise + SBOS	23	86	18	48	18	104

7 Related Work

Superoptimization uses instruction semantics to search for high-performance sequences of instructions equivalent to the input program's instruction sequences. Bansal and Aiken [4] developed a superoptimizer to perform peephole optimizations on short sequences of x86 instructions exhaustively. Souper [21] and Minotaur [15] are superoptimizers for peephole optimizations on LLVM IR. Unlike Hydride, they are not easily extensible to new ISAs since the semantics of their IRs are manually implemented and have no support for automatically extending a language-independent IR, and do not support generation of vector instructions. The idea of scaling down register sizes before synthesis and then scaling the results back up comes from Phothilimthana et al. [16], which they apply for superoptimization of a limited number of scalar ARM instructions. Unlike Hydride, none of these superoptimizers generate ISA semantics automatically to support program synthesis and they provide only limited

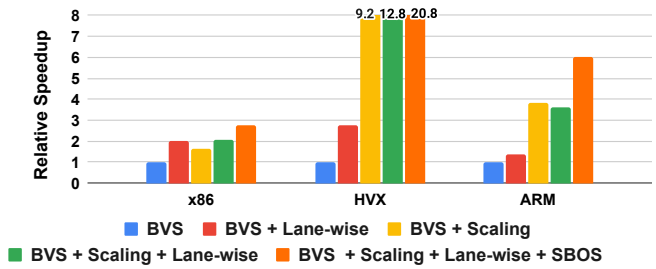


Fig. 7. Speedup of synthesis using Hydride's synthesis heuristics

ISA coverage. More generally, Hydride is a highly retargetable and extensible compiler system that automatically generates high-performance code for multiple targets.

Instruction Selection. Cattel [6] first proposed using machine descriptions to generate target-specific code generators. Buchwald, et al.[5] uses program synthesis offline on a large set of input programs to automatically generate pattern-matching rules for 32-bit scalar x86 instructions. Rake [1] is a compiler for Halide that uses program synthesis to generate a hand-implemented Uber IR to target HVX and ARM. It employs a technique similar to Hydride's lane-wise synthesis to enable synthesis of only two swizzle patterns: interleave and deinterleave. Moreover, since Rake uses manually-implemented semantics, it is not scalable to larger ISAs such as x86. Pitchfork [19] supports synthesis of fixed-point computations for x86 AVX2, HVX and ARM Neon. Importantly, it does not synthesize swizzle instructions, thereby leading to substantial slowdowns on kernels such as matmuls and convolutions.

Autovectorization. Vegen [8] is an autovectorizer for x86 that automatically generates pattern matching rules for autovectorization from x86 ISA specification. It improves the coverage of x86 instructions, especially complex non-SIMD compute instructions. However, the rules are brittle since Vegen does not generate multiple variants of patterns. Also, Vegen does not support specialized swizzle instructions which are critical for performance of tensor and stencil workloads. Diospyros [27] is a target-specific autovectorizer that synthesizes vector instructions for Tensilica for small tensor kernels using equality saturation. The semantics for Tensilica's ISA is manually implemented. Moreover, its synthesis strategy is not scalable to large tensor programs. Isaria [25] extends Diospyros to automatically generate rewrite rules from an ISA specification during an offline phase and use these rules to perform equality saturation when compiling an input program. However, it suffers from the same limitations of Diospyros. Porcupine [10] is an autovectorizer that uses manually-implemented semantics for limited set of SIMD instructions to synthesize code for homomorphic encryption. It also requires users to provide reference and sketch implementation with holes for their input programs. Hydride does not require users to supply reference and sketch and supports a wider range of instructions.

8 Conclusions

We have proposed Hydride, a novel, fully automated approach to compiler construction, which automatically generates the core language- and machine-independent compiler IR with a formal semantics, given only a set of machine ISA pseudocode specifications published by hardware vendors. The IR semantics can be used to implement compilers for high-performance languages using program synthesis instead of laboriously-engineered and brittle pattern-matching rules. This results in high instruction set coverage and high-performance code, even for large, complex instruction sets.

The Hydride approach can benefit other compiler infrastructures. For example, we are currently using Hydride in MLIR to automatically generate target-agnostic dialects and low-level target-specific dialects from ISA specifications. Such dialects are akin to 'x86Vector' and 'ArmNeon' dialects but with far better instruction coverage; moreover, it generates a dialect for Hexagon that does not current exist. The formal semantics automatically generated by Hydride can be used to automatically generate lowering code from 'vector' and 'arith' dialects to the automatically-designed dialects using synthesis. No such capability exists in MLIR today.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, Uday Kumar Reddy, for their helpful feedback on this paper. We also thank Muntasir Mallik and Li He from Qualcomm on providing useful early feedback on the work, and Hashim Sharif from AMD for his valuable comments on early drafts of the paper. This work was supported by funding from Amazon, Qualcomm, Intel, the University of Illinois Urbana-Champaign, and PRISM and ACE, two of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Maaz Bin Safeer Ahmad, Alexander J Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 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*, pages 1004–1016, 2022.

[2] ARM. ARM Developer Intrinsics. https://developer.arm.com/architectures/instruction-sets/intrinsics/f:@navigationhierarchiessimdisa=[Neon].

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, Robert Norton-Wright, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. Isa semantics for armv8-a, risc-v, and cheri-mips. 2019.

[4] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *ACM SIGARCH Computer Architecture News*, 34(5):394–403, 2006.

[5] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 300–313, 2018.

[6] RG Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):173–190, 1980.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 578–594, 2018.

[8] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 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*, pages 902–914, 2021.

[9] Lucian Codrescu. Architecture of the hexagon™ 680 dsp for mobile imaging and computer vision. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–26. IEEE, 2015.

[10] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 375–389, 2021.

[11] Halide. Halide. https://github.com/halide/Halide, 2021.

[12] Intel. Intel Deep Learning Boost. https://www.intel.com/content/dam/www/public/us/en/documents/product-overviews/dl-boost-product-overview.pdf, 2019.

[13] Intel. Intel Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2023.

[14] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[15] Zhengyang Liu, Stefan Mada, and John Regehr. Minotaur: A simd-oriented synthesizing superoptimizer. *arXiv preprint arXiv:2306.00229*, 2023.

[16] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2016.

[17] Qualcomm. Exploring the AI capabilities of the Qualcomm Snapdragon 888 Mobile Platform [video]. https://www.qualcomm.com/news/onq/2020/12/02/exploring-ai-capabilities-qualcomm-snapdragon-888-mobile-platform, 2020.

[18] Qualcomm. Qualcomm Hexagon V66 HVX Programmer's Reference Manual. https://developer.qualcomm.com/downloads/qualcomm-hexagon-v66-hvx-programmer-s-reference-manual, 2022.

[19] Alexander J Root, Maaz Bin Safeer Ahmad, Dillon Sharlet, Andrew Adams, Shoaib Kamil, and Jonathan Ragan-Kelley. Fast instruction selection for fast digital signal processing. 2023.

[20] Alexander James Root. *Optimizing Vector Instruction Selection for Digital Signal Processing*. PhD thesis, Massachusetts Institute of Technology, 2022.

[21] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

[22] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 136–148, 2008.

[23] Tensorflow XLA Team. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla, 2022.

[24] The LLVM Project. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html, 2022.

[25] Samuel Thomas and James Bornholt. Automatic generation of vectorizing compilers for customizable digital signal processors. 2024.

[26] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[27] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. 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*, pages 874–886, 2021.