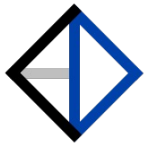
MISAAL: Synthesis-Based Automatic Generation of Efficient and Retargetable Semantics-Driven Optimizations

ABDUL RAFAE NOOR, University of Illinois at Urbana-Champaign, USA
DHRUV BARONIA, University of Illinois at Urbana-Champaign, USA
AKASH KOTHARI, University of Illinois at Urbana-Champaign, USA
MUCHEN XU, University of Illinois at Urbana-Champaign, USA
CHARITH MENDIS, University of Illinois at Urbana-Champaign, USA
VIKRAM S. ADVE, University of Illinois at Urbana-Champaign, USA

2025. 12. 09



CODE Lab
Computing Optimization and
Data-driven Exploration Lab

3-Line Summary

- **Challenge: Bridging HW Instructions and Compiler Optimizations**
  - Modern ISAs (AVX, Hexagon, ARM Neon) introduce complex compute and swizzle instructions.
  - Manual backend development for DSL compilers (Halide, TVM) is error-prone and unscalable.
  - Existing synthesis-based approaches (Hydride, Rake) still suffer from large search spaces and poor performance.
- **Solution: MISALL**
  - (1) Misaal addresses scalable enumeration for synthesis of rewrite rules
    - **Relevance pruning**
    - **Equivalence-class based enumeration**
    - Offline execution
  - **(2) Optimized cross-lane compute & data movement (without hurting scalability)**
  - **(3) Reducing the number of rewrite rules (Abstaraction)**
    - Rewrite rule abstraction
    - Numeric parameter lifting
    - Retargetable templates

Outline

II. Background

Background : The Gap

- Demands of modern workload & Extending HW specialized instructions
  - Qualcomm's Hexagon DSP, Intel's AVX ISA, ARM Neon ISA
  - Specialized instructions to optimize vector, tensor, and stencil computations
- Conventional practice of supporting emerging ISAs and adding new extensions
  - It has been to **manually** implement backend for each HW target in DSC(Halide, TVM, MLIR, etc)
  - → Entails implementing a large set of pattern-matching rules (map input operations to complex target instructions)
  - Pattern-matching rules manually written have (1) error-prone; (2) engineering effort; (3) brittle
- Recent papers have proposed the use of program synthesis techniques
  - These works eliminate the need to manually implement compiler backends using pattern-matching
  - They leverage (1) the semantics of an input language (2) IR (3) target hardware instruction to generate target code
  - These approaches provide formal correctness guarantees without compromising performance

Background : Related works

- Compiler construction for specific contexts
  - Diospyros [1] : A vectorizer that use a set of manually-defined rewrite rules (12 vector instructions) using **equality saturation** (for Tensilica DSP)
  - Isaria [2] : Automatically generates rewrite rules using ISA specification
  - → But, both works don’t support generation of (1) complex compute and (2) data swizzle instructions
- Advanced Compiler construction based on program synthesis
  - Rake [3] : Uses program synthesis to lower input code operations to target vector instructions at compile time (small subset, semantics of target by hand)
  - Use specialized heuristics for targeting cross-lane compute instructions and data de/interleave (But, it take several hours to complete)
  - Pitchfork [4] : Uses Rake in an offline stage and apply lightweight term rewriter to apply the rewrite rules on input code (But, it does not support data move)
- Hydride
  - It automatically generates formal semantics of instructions from their pseudocode in official document
  - These semantics is used for creating similar classes of ISA operations, and generating AutoLLVM
  - However Hydride still suffers from serious performance limitations like previous synthesis based compiler

Background : Main Compenents of Hydride

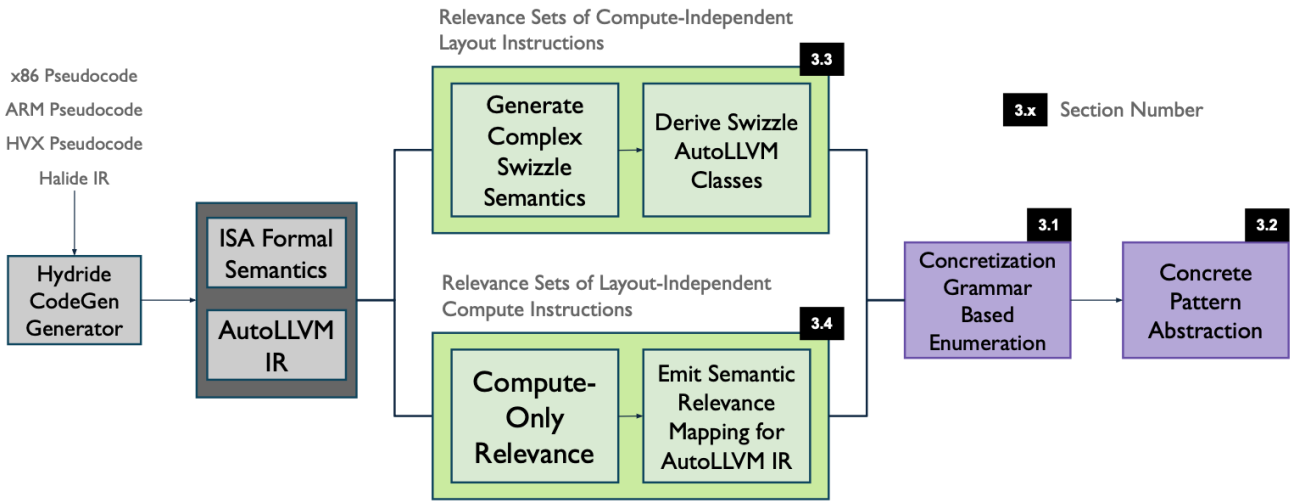
- HYDRIDE Automatic IR Generator
  - Automatically generate formal semantics of HW instructions by parsing official pseudocode documentation
  - Performs a **similarity analysis** across target-specific instructions, and put them to equivalence instruction class
  - In equivalence classes, class is represented as parameterized target-independent instructions (*AutoLLVM IR)*
  - This component also automatically generates support for performing *ono-to-one translation from AutoLLVM to target-specific*
- HYDRIDE Code Synthesizer
  - Compiles expression in a frontend language, such as Halide IR or MLIR dialects to AutoLLVM IR expressions (at compile time)
  - (1) Partition large expression into non-overlapping sub-expression, (2) AutoLLVM IR Pruning, (3) hand-crafting swizzles pattern
  - Howerver, it takes, in some cases, several hours despite using several heuristics due to inherent a large search space
- MISALL
  - Builds on top of the HYDRIDE to replace its Code Synthesizer and generate code in a matter of a few seconds

Background : Challenges

- Why we use synthesis-based compiler instead of pattern matching?
  - (a) Synthesis techniques instead of pattern matching
  - (b) Automatic generation of translation components from (1) input language semantics (2) compiler IRs (3) target ISAs
  - (c) Partial progress towards better instruction coverage for complex instruction sequences
  - (d) Generating rewrite rules in an offline stage, performing term rewriting at compile time
- Challenge 1 : Program synthesis for large ISAs is intractable due to an **exponentially large search space**
- Challenge 2 : **Complex cross-lane** vector instructions with implicit data movements prohibit scaling problem
- Challenge 3 : **The rewriting systems used at compile** time are intractable with today's approaches.

Background : Solution

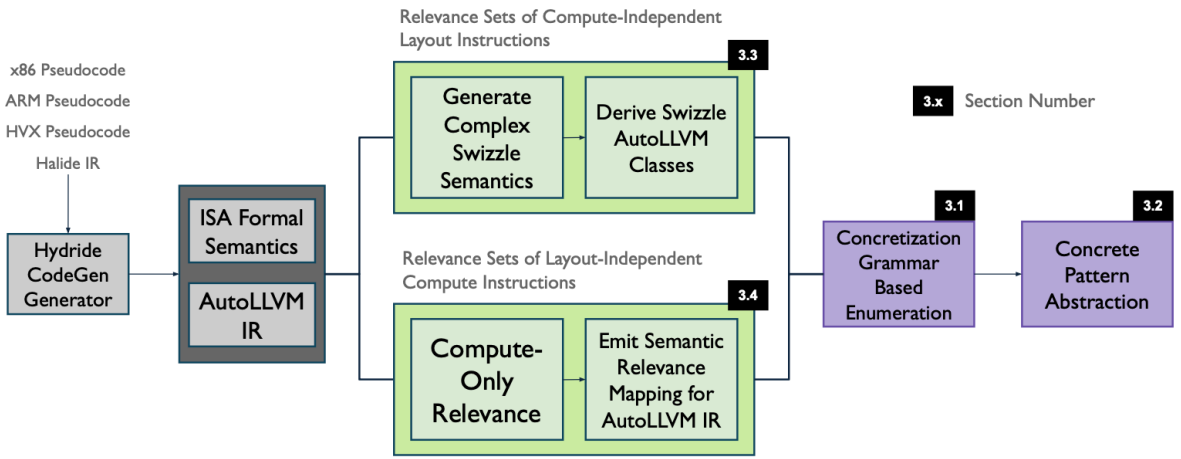
- Solution1 : Uses the **equivalence classes** of target instructions to reduce the exponentially large search space
- Solution2 : Automatically derive the appropriate **data swizzle** patterns required to use those instruction
  - Runs HYDRIDE's **similarity analysis** to represent those patterns using
  - → But this approach still results a large number of candidate graph applied to input code at compiler-time
- Solution3 : Automatically **abstract rewrite rules** to produce a significantly smaller set of rewrite rules



III. MISAAL

MISAAL : Design

- Scaling Synthesis for Large ISAs at the offline stage
  - (1) The ISA formal semantics are generated and abstracted into the AutoLLVM IR representation
    (folds multiple semantically similar operations into target agnostic IR with parameterization)
  - → These compact AutoLLVM IR is enumerated and derive rewrite rules, which can be abstracted into retargetable rewrite rules
  - (2) Extracts the data-access patterns from each ISA operation and is folded into new AutoLLVM IR (for data-swizzling semantics)
  - (3) Prune the enumeration space according to classes that share some semantic relatedness, regardless of their data access
  - (4) Abstract the generated rewrite rules into higher-order retargetable rewrite rules with symbolic parameterizations
- Lightweight Online Compilation
  - Retargetable rewrites generated offline are fed to term rewriting system
  - Finally lowered to the equivalent target ISA operations



MISAAL : Automatic Complex Swizzle Discovery

- Difficulty of discovering rewrite rules using complex cross-lane operations
  - General permute instruction is intractable
  - Compiler writer have to manually reason about the semantics of the ISA operations and relate swizzling to computation instructions
  - There is little guarantee of completeness of being sufficient to capture the many complex data-movements
- Automatic Complex Swizzle Discovery
  - Misaal automatically derives a set of complex data-swizzles offline purely from the semantics of each target architecture ISA.
  - *Vmpybv,* To leverage this instruction on contiguous values, the input program must apply swizzling respect to the ordering of the input values

  → Analyze the data-access patterns and to create two categories of swizzles which produce vectors in the required data-access patterns.
    - Interleaved the operands (Figure(b))
    - Deinterleaved the result of (Figure(c))
  - Applies HYDRIDE's Similarity analysis on these swizzle to create AutoLLVM IR of target-specific swizzle
  - We can enumerate expression with swizzle operations

(a) Simplified version of HVX's widening multiplication

```
%arg0 = [A0, A1, A2, A3]
%arg1 = [B0, B1, B2, B3]
// Sign Extend Inputs
%arg0.sext = [A0.s, A1.s, A2.s, A3.s]
%arg1.sext = [B0.s, B1.s, B2.s, B3.s]
// Interleaved Access Pattern
%widen.mul = [(A0.s x B0.s), (A2.s x B2.s), (A1.s x B1.s), (A3.s x B3.s)]
```

(b) Interleave Operands before widening multiplication

```
%arg0 = [A0, A1, A2, A3]
%arg1 = [B0, B1, B2, B3]
// Interleave Operands
%arg0.i = [A0, A2, A1, A3]
%arg1.i = [B0, B2, B1, B3]
// Sign Extend Inputs
%arg0.sext = [A0.s, A2.s, A1.s, A3.s]
%arg1.sext = [B0.s, B2.s, B1.s, B3.s]
%widen.mul = [(A0.s x B0.s),(A1.s x B1.s),
              (A2.s x B2.s),(A3.s x B3.s)]
%result = vector-add %widen.mul [C0, C1, C2, C3]
```

(c) Deinterleave result after widening multiplication

```
%arg0 = [A0, A1, A2, A3]
%arg1 = [B0, B1, B2, B3]
// Sign Extend Inputs
%arg0.sext = [A0.s, A1.s, A2.s, A3.s]
%arg1.sext = [B0.s, B1.s, B2.s, B3.s]
%widen.mul = [(A0.s x B0.s), (A2.s x B2.s),
              (A1.s x B1.s), (A3.s x B3.s)]
// Deinterleave product after widening
%widen.mul.d = [(A0.s x B0.s), (A1.s x B1.s),
                (A2.s x B2.s), (A3.s x B3.s)]
%result = vector-add %widen.mul.d [C0, C1, C2, C3]
```

(a) Dot-product with non-cross lane access with elements from across operands

```
%0 = <8xi32> sign-extend <8xi16> %arg0
%1 = <8xi32> sign-extend <8xi16> %arg1
%2 = <8xi32> vector-mul %0, %1
%3 = <8xi32> sign-extend <8xi16> %arg2
%4 = <8xi32> sign-extend <8xi16> %arg3
%5 = <8xi32> vector-mul %3, %4
%6 = <8xi32> vector-add %2, %5
%7 = <8xi32> vector-add %6, %accum
```

(b) Dot-product with cross-lane accesses requiring swizzles

```
// Interleave 16-bit elements 2 from 128-bit
vectors
%0 = misaal.interleave.16b.2op.128b %arg0 %arg2
%1 = misaal.interleave.16b.2op.128b %arg1 %arg3

// Apply dot product ( cross-lane access )
// on interleaved vectors
%2 = _mm256_dpwssd_epi32 %accum %0 %1
```

MISAAL : Semantics-Based Search Space Pruning

- Limitation of using equivalence classes
  - In terms of reducing the enumeration space by orders of magnitude(3 x 10^12 → 2.9 x 10^8, 9000x reduction)
  - **But, scalability issues of enumerating deeper expressions still persist** → This implies some manner of pruning is inevitable
- Semantic-Based Search Space Pruning
  - When enumerating for complex instructions we should include some operations and others should be excluded (This intuition may not be portable)
  - Create distinct sets of AutoLLVM IR operation to enumerate → Relevance Sets →
  - For solving Compute-Only synthesis problem, authors define Relevance Domain Specific Language
    - An instruction / is included in the relevance set of R if and only if there exists a synthesized expression E, composed of the RDSL and /→ Relevance set include /
  - For every vector operation, simple or complex, there is a repeating pattern for groups of lanes. → Lane Scaling (Cannot treat swizzle problem)
  - $BVSlices$(R, Oldx,Ildx) --> Relevance synthesis problem directly includes the corresponding input operands bitvector slices + Minimum set of lanes / reduction

Compute-Only Instruction Relevance: _mm_cvtepu8_epi16 for _mm256_dpbusd_epi32

```
%0 = <4xi16> rdsl-sext <4xi8> BVSlices(_mm256_dpbusd_epi32, /* output lane */ 0, /* %arg0 */ 0)
%1 = <4xi16> _mm_cvtepu8_epi16 <4xi8> BVSlices(_mm256_dpbusd_epi32, /* output lane */ 0, /* %arg1 */ 1)
%2 = <4xi16> rdsl-mul %0, %1
%3 = <1xi16> rdsl-vector-reduce-add 4 <4xi16> %2 // Horizontal vector reduction with reduction factor 4
%4 = <1xi32> rdsl-sext <1xi16> %3
%5 = <1xi32> rdsl-add %4, %arg2
```

4-point dot product in x86

```
// _mm256_dpbusd_epi32
%0 = <32xi16> sign-extend <32xi8> %arg0
%1 = <32xi16> zero-extend <32xi8> %arg1
%2 = <32xi16> vector-mul %0, %1
%3 = <8xi16> vector-reduce-add 4 <32xi16> %2
%4 = <8xi32> sign-extend <8xi16> %3
%5 = <8xi32> vector-add %4, %arg2
```

MISAAL : Equivalence Class Based Enumeration

- Limitation of Traditional Method
  - LHS and RHS are consequently target specific ISA operations **with fixed types** (In definition of LHS = RHS for checking rewrite rules)
  - Rewrite rules for the 128-bit dot-product and for 512-bit dot-product program are distinct instructions
  - → Coupled with the additional ranges of intermediate value bit-widths and vector sizes, this demonstrates why enumeration becomes intractable
- AutoLLVM IR
  - Employ Hydride's Similar Instruction analysis to automatically create an abstraction which represents multiple semantically similar instructions
  - AutoLLVM Ir is automatically derived target agnostic abstracton for target instructions
  - $\boldsymbol{Abstraction}(TargetInstExprC)$= AutoLLVM IR representation of C
  - $\boldsymbol{Concretization}(AutoLLVMInstExpr\ A)$= { Set of all concretely parameterized programs represented by expression A }
  - → Instead of enumerating using target specific ISA, Misaal enumerates using automatically generated AutoLLVM IR to generate rewrite rule templates

`<8 x i16> mul` + `<8 x i16> add` ⟶ `@llvm.x86.avx2.pmaddwd`

`<16 x i16> mul` + `<16 x i16> add` ⟶ `@llvm.x86.avx2.pmaddwd.256`

`<32 x i8> mul` + `<32 x i8> add` ⟶ `@llvm.x86.avx2.pmaddubsw`

(d) autollvm.dot.prod

```
%0 = (choose* {sign-extend, zero-extend, truncate, saturate})
%1 = (choose* {sign-extend, zero-extend, truncate, saturate})
%2 = (choose* {vector-mul}) %0, %1
%3 = (choose* {vector-reduce-add, vector-reduce-signed-sat-ad
%4 = (choose* {vector-add, vector-signed-sat-add, ...}) %3, (
```

MISAAL : Equivalence Class Based Enumeration

- **Equivalence Class Based Enumeration**
  - $\exists l \in Concretization(LHS\ IR)$, $\exists r \in Concretization(RHSIR)$, $l \equiv r$ (Valid template for a rewrite rule if and only if)
  - Finding the $l$and $r$can be efficiently solved by formulating it as SysGus(Syntax Guided Synthesis) problem
  - The symbolic choices may be from (1) a set of operands, (2) intermediate values, or (3) AutoLLVM IR concretizations.
  - All variants on different condition are captured in the above single concretization grammar
  - More generally, for any given rewrite rule $LHSIR \simeq IR\ RHSIR$, the synthesis problem jointly searches for any concretizations of both expressions under which they produce symbolically equivalent outputs.
  - The output of this phase produces only a single valid concretization as a template, if it exists, for a pair of AutoLLVM IR expressions.
  - → how the derived concretization template is abstracted to apply across these (potentially) multiple concretizations (Section 3.2)

(d) autollvm.dot.prod

```
%0 = (choose* {sign-extend, zero-extend, truncate, saturate}) (choose* {%arg0, %arg1, %arg2})
%1 = (choose* {sign-extend, zero-extend, truncate, saturate}) (choose* {%arg0, %arg1, %arg2})
%2 = (choose* {vector-mul}) %0, %1
%3 = (choose* {vector-reduce-add, vector-reduce-signed-sat-add, ...}) (choose* {2,4,8}) %2
%4 = (choose* {vector-add, vector-signed-sat-add, ...}) %3, (choose* {%arg0, %arg1, %arg2})
```

MISAAL : Target-Agnostic Rewrite Rule Abstraction

- **One valid concretization implies others may exist**
  - (1) Other concretizations are extracted by issuing synthesis queries from Section 3.1 with additional semantic constraints
    - Different vector register sizes for inputs and/or outputs
    - Fixing the parametrization of specific AutoLLVM IR operations in source/target expressions
  - (2) Abstract numeric parameters into symbolic expressions
    - Misaal abstracts the numeric parameters such that a single higher-order rewrite represents all enumerated concretizations.
  - This abstraction enables Misaal to represent a large number of rewrite rules while covering a larger number of potentially valid concretizations.

(a) Element-wise add on 256-bit vector

```
%0 = <32 x i8 > vec-add %arg0 %arg1
```

(b) Concatenation of two 128-bit vector additions

```
// Slice Low Half ...
%0 = <16 x i8> slice_vector %arg0
            /* offset */       0,  /* Stride */ 1,
            /* Num Elements */ 16,
            /* Bitwidth */     8,
            /* Vector Size */  256
%1 = <16 x i8> slice_vector %arg1
            /* offset */       0,  /* Stride */ 1,
            /* Num Elements */ 16,
            /* Bitwidth */     8,
            /* Vector Size */  256
// Elided Slice High Half for space ...
%4 = <16 x i8> vector-add    %0 %1
%5 = <16 x i8> vector-add    %2 %3
%6 = <32 x i8> concat_vector %4 %5 8 128
```

(c) Abstracted Element-wise add on full vector

```
%0 = vec-add %arg0 %arg1 %bitwidth %vectorsize
```

(d) Abstracted vector addition by slicing and concatenating

```
// Slice Low Half ...
%0 = slice_vector %arg0
    /* offset */       0,  /* Stride */ 1,
    /* Num Elements */(%vectorsize/(%bitwidth*2)),
    /* Bitwidth */     %bitwidth,
    /* Vector Size */  %vectorsize
%1 = slice_vector %arg1
    /* offset */       0,  /* Stride */ 1,
    /* Num Elements */(%vectorsize/(%bitwidth*2)),
    /* Bitwidth */     %bitwidth,
    /* Vector Size */  %vectorsize
// Elided Slice High Half for space ...
%4 = vector-add    %0 %1 %bitwidth (%vectorsize/2)
%5 = vector-add    %2 %3 %bitwidth (%vectorsize/2)
%6 = concat_vector %4 %5 %bitwidth (%vectorsize/2)
```

IV. Evaluation

Evaluation

- **Baseline**
  - SOTA Halide(v13), Synthesis-based compiler HYDRIDE, Rake using 33 benchmarks from image processing and deep learning domains
  - The primary baseline is Halide's production-quality backend developed and optimized for commercial users
  - Halide programs must be tuned manually for each hardware target by optimizing their schedules
- **Benchmarks**
  - Benchmarks are hand-tuned for **x86 by the authors**, and for **ARM and Hexagon by Qualcomm and Adobe**.
  - The image processing kernels include **image dilation, blurs, and edge detection filters across a range of filter sizes**, and others
  - We include important deep learning kernels, such as **matrix multiplication on tensors of low batch sizes (1, 2 and 4)**
  - We also evaluate some fused versions of deep learning kernels commonly found in neural networks.
- **Experiment Setup**
  - For x86, Intel Xeon Silver 4216 CPU (16 cores, 2.1GHz, 22 MB L3 cache) with hyperthreading disabled;
  - For HVX, a cycle-accurate simulator in Hexagon SDK v3.5.2 provided by Qualcomm
  - For ARM, an Apple M2 CPU (3.49GHz, 16 GB memory, 16 MB L3 cache)
  - Use AMD EPYC 7453 28-Core Processor 2.7 GHz for generating the offline components of Misaal

Evaluation : EggLog Usage

- **EggLog**
  - Misaal uses **EggLog** for online rewrite rule application to compile programs in **Halide IR to AutoLLVM IR**. (Rule application)
  - Abstracted rewrite rules are generated separately per architecture.
  - Halide IR operations are given cost 10,000 and AutoLLVM IR operations are given cost 1 (to ensure that the extracted expression is in terms of AutoLLVM IR)
  - Bursts of 5 iterations of equality saturation are run until the extracted expression is purely in terms of AutoLLVM IR.
- We terminate when the extracted expression is purely in terms of AutoLLVM IR corresponding to the target ISA.

Evaluation : Compilation Times

Benchmark	Rake	Misaal
sobel3x3	5988	300 **(20x)**
dilate3x3	3398	100 **(34x)**
avgpool	3648	329 **(11x)**
maxpool	63	413 **(0.2x)**
add	1879	517 **(3.6x)**
fully_connected	3669	193 **(19x)**
Geomean	1784	272.2 **(6.8x)**

Benchmark	Hydride Compilation Times (s)			Misaal Compilation Times (s) (Compilation Speedup)		
	x86	*HVX*	*ARM*	*x86*	*HVX*	*ARM*
sobel 3x3	8300	4740	590	81 **(103x)**	301 **(16x)**	86 **(7x)**
sobel 5x5	10511	9171	332	416 **(25x)**	629 **(15x)**	81 **(4x)**
dilate 3x3	90	80	251	45 **(2x)**	100 **(1x)**	36 **(7x)**
dilate 5x5	90	120	86	33 **(3x)**	111 **(1x)**	25 **(3x)**
dilate 7x7	45	120	41	36 **(1x)**	110 **(1x)**	27 **(2x)**
box blur 3x3	724	450	82	118 **(6x)**	210 **(2x)**	26 **(3x)**
box blur 5x5	943	226	8832	159 **(6x)**	268 **(1x)**	99 **(89x)**
box blur 7x7	1155	6900	8274	204 **(6x)**	453	224

- Misaal achieves **order-of-magnitude reduction** in compilation time
  - Geomean reduction of 16x for x86, 9x for HVX and 10x for ARM
- For kernels such as conv and l2norm where data swizzling and cross-lane vector operations are needed
  - Hydride can compile a single application for over 5 hours, whereas Misaal completes compilation in the order of seconds (224x)
  - This is expected as Hydride leverages SMT solvers, whose complexity increases with arithmetic operations such as multiplication and division.
- Hydride achieves comparable or even faster compilation times for simpler kernels such as **dilate** and **max pool**.
  - Misaal does not perform any pruning during compilation and includes all front-end and target rewrite rules.
- Hydride compiles large Halide IR programs by decomposing expressions into small windows and synthesizing each separately.
  - Results in optimizations over large sequences of operations, potentially yielding better code and also simplifying the compiler design.
- Rake vs MISAAL
  - Rake fails to compile 27 out of 33 benchmarks due to crashes in the Rosette interpreter for HVX and all of the benchmarks
  - Misaal compiles 6.84x faster than Rake for all the benchmarks for HVX we could evaluate

Evaluation : Memory Usage

- Excessive memory consumption is a major obstacle to scalability for previous synthesis-based compilers, like Hydride.
  - Misaal requires orders of magnitude less memory to compile than Hydride.
  - Hydride achieves retargetability at the cost of up to 15 GB of memory for benchmarks such as depthwise convolution on x86.
  - For x86 and ARM, Misaal achieves a geomean memory reduction of 18× and 26×, whereas for HVX the reduction is 3×.
- Why HVX shows smaller memory reduction
  - This can be attributed to the characteristics of the ISA themselves.
- Practical feasibility: Edge vs server-class machines
  - Misaal reaches a maximum memory usage of 2.5 GB, which remains feasible for edge devices, whereas Hydride exceeds 15 GB.
  - Misaal is able to scale compilation across much larger program sequences, unlike Hydride which must split programs into smaller synthesis queries.
- **Offline memory usage of Misaal**
  - The offline flow parallelizes the enumeration across 64 processes
  - Across the target architectures ISAs, we observe a sustained peak memory usage of 8 Gb
  - A single synthesis task for Hydride may require gigabytes of memory.

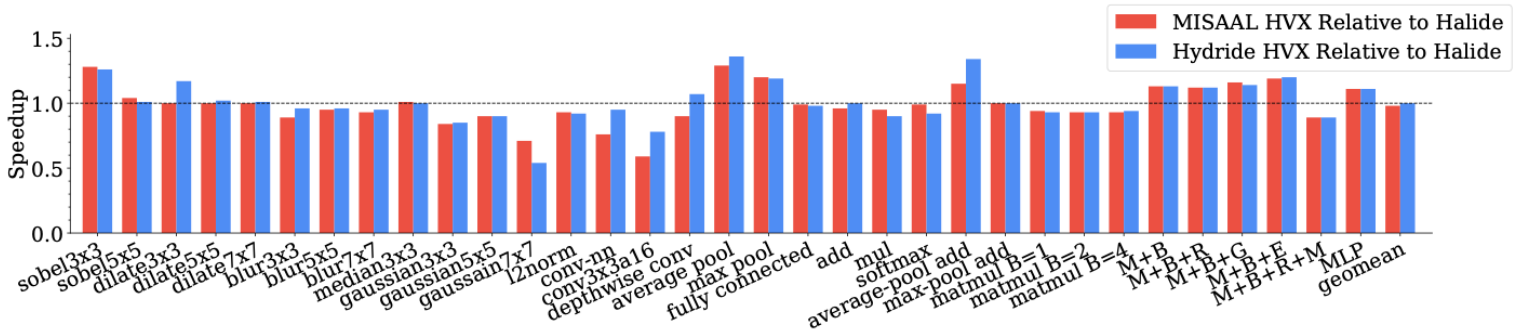
Hydride Peak Memory (Mb)			Misaal Peak Memory (Mb) (Memory Improvement)		
x86	*HVX*	*ARM*	*x86*	*HVX*	*ARM*
2.5K	1.6K	8.5K	146 **(17x)**	1.2K **(1x)**	82 **(103x)**
2.3K	1.6K	1.9K	190 **(12x)**	2.3K **(1x)**	82 **(23x)**
863	1.6K	913	124 **(7x)**	677 **(2x)**	78 **(12x)**
849	1.9K	814	113 **(8x)**	630 **(3x)**	79 **(10x)**
856	1.9K	676	119 **(7x)**	663 **(3x)**	80 **(8x)**
667	1.1K	857	101 **(7x)**	767 **(1x)**	79 **(11x)**
868	1.1K	12.5K	101 **(9x)**	917 **(1x)**	96 **(130x)**
876	7K	12.3K	102 **(9x)**	1.7K **(4x)**	96 **(129x)**

Evaluation : Performance Evaluation

- Common Baseline : Halide’s manually-written and optimized target-specific back ends.
- Performance on X86
  - Achieves a geomean 10% performance improvement on x86 compared Halide’s x86 back end, with a maximum speedup of 2.08x
  - In these benchmarks, Misaal effectively leverages efficient **x86 dot-product instructions.**
  - In the batched matrix multiplication kernels, Misaal identifies that **a dot-product instruction can be repurposed as a widening multiply-add.**
  - Misaal identifies opportunities to generate saturation instructions for 64-bit values, which Halide cannot due to limited support.
  - Additionally, Misaal reduces the number of **shuffle vector operations (data-swizzling)** compared to Halide.
  - **Although Halide supports some x86-specific optimizations, its capabilities are limited by the large size of the x86 ISA.**
- Performance on ARM
  - Achieves a geomean speedup of **1.02x** against Halide on ARM, with a maximum speedup of **1.21x** over Halide’s production **ARM backend**
  - Hydride gets a geomean speedup of 3% and a maximum speedup of 1.21x
  - For the fully connected benchmark, Hydride fails to synthesize dot product instructions since it requires synthesis over a larger sequence of input operations
  - Halide generates ushll for constant multiplication, missing the opportunity to combine it with addition. (←→ umlal in MISAAL)

Evaluation : Performance Evaluation

- Baseline strength: Halide HVX backend is the most aggressively optimized
- Performance on Hexagon
  - Achieves a geomean relative performance of 0.98 against the Halide HVX back end, which is also what Hydride delivers.
  - For kernels such as sobel and average pooling, Misaal and Hydride generate appropriate scalar-vector widening multiplication operations, whereas Halide fails to do so.
  - For conv3x3a16, gaussian7x7, both Misaal and Hydride incur significant slowdowns compared with Halide.
  - For kernels such as conv_nn, Misaal generates vector-add operations on smaller vector registers, leading to slowdowns.
  - → Because HVX's arithmetic operations do not support all vector register sizes, compilers must split large vectors and concatenate them.



(b) Performance on HVX

Evaluation : Rewrite Rules Abstraction

- Quantitative Reduction in Rewrite Rules
  - x86 and ARM observe the largest reduction in the number of rewrite rules.
  - Since x86 and ARM ISAs are mostly generic and provide multiple versions of the same operations across different bitwidths and vector register sizes.
  - HVX exhibits a relatively modest reduction due to being a DSP architecture with specialized instructions.
- Compiler Construction
  - Enumeration is performed up to a depth of 4 (with up to 5 terminals), resulting in over a month-long enumeration time per target.
  - Halide IR requires significantly higher time for extracting concrete rewrite rules and rewrite rule abstraction.
  - While we exhaustively extract all possible concretizations, a sufficient cut-off can be provided as a parameter.

Table 5. Number of rewrites derived from each target and their corresponding number of abstracted rewrites.

Target	# Concrete Rewrites	# Abstracted Rewrite	Reduction in # Rewrite
x86	5,806	329	17.6x
HVX	4,586	412	11.1x
ARM	4,085	190	21.5x
Halide	27,364	1,284	21.3x
Total	**41,841**	**2,215**	**18.9x**

Table 6. Compiler Construction times for automatically generated components of Misaal. Enumeration is done up to a depth of 4 with up to 5 terminals. NA : Not applicable.

Category	x86	HVX	ARM	Halide
Compute-Only Relevance	16 hours	14 hours	21 hours	10 hours
Swizzle-Generation	37 mins	9 mins	8 mins	NA
AutoLLVM IR Enumeration	1 month+	1 month+	1 month+	1 month+
Extracting Concrete Rewrite Rules	5.9 hours	6.3 hours	3.5 hours	52 hours
Rewrite Rules Abstraction	10 mins	20 mins	6 mins	56 mins

V. Conclusion

Conclusion : MISAAL

- Misaal, a retargetable compiler for Halide based on offline synthesis and online(compile-time) rewriting
  - Most Misaal components are generated fully automatically from vendor-provided pseudocode specifications of the target ISA semantics
  - And existing formal semantics of the Halide front-end IR.
- (1) Misaal addresses scalable enumeration for synthesis of rewrite rules
  - Relevance pruning
  - Equivalence-class based enumeration
  - Offline execution
- (2) Optimized cross-lane compute & data movement (without hurting scalability)
- (3) Reducing the number of rewrite rules
  - Rewrite rule abstraction
  - Numeric parameter lifting
  - Retargetable templates
- Misaal delivers competitive and sometimes even better performance than the highly tuned, production Halide compiler on all three architectures.