Vectorization for Digital Signal Processors via Equality Saturation

Alexa VanHattum
Cornell University
Ithaca, NY, USA

Rachit Nigam
Cornell University
Ithaca, NY, USA

Vincent T. Lee
Facebook Reality Labs Research
Redmond, WA, USA

James Bornholt
The University of Texas at Austin
Austin, TX, USA

Adrian Sampson
Cornell University
Ithaca, NY, USA

ABSTRACT

Applications targeting digital signal processors (DSPs) benefit from fast implementations of small linear algebra kernels. While existing auto-vectorizing compilers are effective at extracting performance from large kernels, they struggle to invent the complex data movements necessary to optimize small kernels. To get the best performance, DSP engineers must hand-write and tune specialized small kernels for a wide spectrum of applications and architectures. We present Diospyros, a search-based compiler that automatically finds efficient vectorizations and data layouts for small linear algebra kernels. Diospyros combines symbolic evaluation and equality saturation to vectorize computations with irregular structure. We show that a collection of Diospyros-compiled kernels outperform implementations from existing DSP libraries by 3.1× on average, that Diospyros can generate kernels that are competitive with expert-tuned code, and that optimizing these small kernels offers end-to-end speedup for a DSP application.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; • **Hardware → Digital signal processing**; • **Theory of computation → Vector / streaming algorithms**.

KEYWORDS

Vectorization, DSPs, Program Synthesis, Equality Saturation

ACM Reference Format:
Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446707

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446707

1 INTRODUCTION

Compute-heavy embedded sensing applications, from augmented reality to 5G networking, rely on digital signal processors (DSPs). DSPs target power- and energy-constrained domains with real-time performance targets, so their design optimizes for power efficiency over programmability and software compatibility. Their simple in-order cores help meet strict real-time deadlines but also mean that unoptimized code performs poorly. For performance, DSP architectures expose VLIW and vector instruction sets with exotic architecture-specific extensions. These instruction sets offload the burden of parallelization onto the compiler and programmer.

DSP applications typically rely on on two categories of computational *kernels*[1]: (1) large-scale kernels operating on high-dimensional data (much larger than the machine's vector width), and (2) small-scale kernels operating on low-dimensional data (on the order of the vector width). In an industrial context, the distribution of kernels tends to be bimodally distributed: many have small dimensionality (~3–6), and the remaining are much larger (~100–1000). While compiler toolchains and vendor libraries for DSPs often focus their attention on large-scale kernels—shipping linear algebra libraries tuned for large, dense operations—small-scale kernels still consume a non-trivial portions of the end-to-end performance of many emerging DSP applications. Some DSP applications are bottlenecked by small-scale kernels as part of the "last mile" of a larger computation. In other words, a variety of small kernels impose an Amdahl limitation [8, 25, 42] that yields diminishing returns from speeding up just the large-scale loops. Other applications, such as simultaneous localization and mapping (SLAM) [20, 21, 33, 34] and structure from motion [35], have many components that are dominated entirely by small-scale kernels.

Compiling efficient small-scale kernels is challenging even for state-of-the-art compiler techniques because the best performance requires complex data movement strategies that are beyond the scope of most automatic vectorization. Moreover, DSP architectures are extremely diverse: they offer per-application instruction set customization and can even support custom proprietary ISA extensions [11]. As a result, DSP engineers still manually apply device- and kernel-specific optimizations by hand-writing vector intrinsics [2, 17, 43]. This manual effort does not scale with the plethora of kernels and target architectures. For example, products and convolutions of small 3×3 and 4×4 matrices are commonplace

[1] Here, we define a kernel to be a function that consumes one or more multidimensional input matrices and produces one or more multidimensional output matrices. A kernel can be implemented as multiple nested source-level functions.

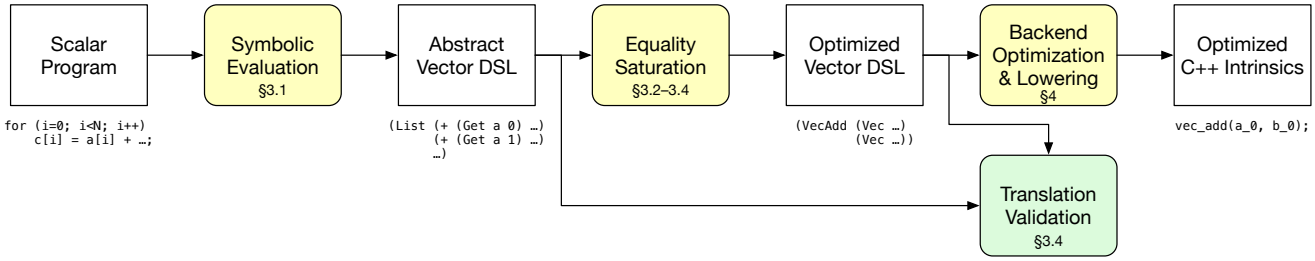


Figure 1: The Diospyros compiler workflow. Diospyros first lifts scalar input programs into a high-level DSL via symbolic evaluation and then searches for equivalent optimized programs using equality saturation. The optimized program is finally lowered to C++ intrinsics for compilation with a DSP toolchain.

in various machine perception applications, but the most efficient implementations for these two sizes are very different. Specialized kernels for each size can vastly outperform general implementations in linear algebra libraries [15, 31].

This paper designs a compiler, Diospyros, that aims to compete with manual tuning by DSP experts while baking in minimal assumptions about the target hardware. Diospyros frames compilation as a search problem in a space of candidate programs. It defines this search space using a system of rewrite rules that encompass both high-level functional specifications and low-level device-specific instructions. Crucially, the resulting program space includes implementations that use arbitrary indexing to express complex data movement patterns. Unlike traditional approaches to general-purpose vectorization [16], Diospyros focuses on using the *shuffle* and *select* instructions common in DSPs to implement the irregular data movement necessary to pack as much work as possible into vector lanes.

Figure 1 shows the Diospyros compilation workflow. Diospyros takes a program in a scalar, imperative language and lifts it to a high-level vector DSL using symbolic evaluation. The core optimization engine is an exhaustive search in a restricted space of candidate programs from this DSL using *equality saturation* [13, 36, 40]. Most compilers apply rewrite rules in a fixed order, which offers predictable compilation but sacrifices optimality. Equality saturation effectively applies *all* rewrite rules simultaneously by representing the input program as an E-graph [23] and performing congruence closure using the rewrite rules as an equivalence relation. The saturated E-graph compactly represents the entire space of candidate programs, from which Diospyros can extract the most efficient one according to an abstract cost model. After extracting the optimal program, Diospyros lowers it to C vector intrinsics for code generation via a backend DSP compiler.

We implement Diospyros to target Tensilica DSPs and show that it can compile kernels that outperform optimized library functions from the Tensilica SDK by a geometric mean speedup of 3.1×. Compared to one expert-written kernel hand-tuned for a fixed matrix size, Diospyros produces code within 8% of the expert performance within 2.2 seconds of compilation time. To show that Diospyros-compiled kernels offer end-to-end speedups on realistic applications, we integrate them into code from Theia [35], an open-source computer vision library for structure from motion (SFM). The Diospyros version of this application performs 2.1× faster on our selected functionality than Theia's original implementation, which uses the Eigen template library for linear algebra [12].

This paper's contributions include: (1) a strategy for using symbolic evaluation and equality saturation to search for SIMD implementations of high-level specifications, (2) Diospyros, an end-to-end compiler design that uses the rewrite system to optimize computational kernels for DSP architectures, and (3) an evaluation on a range of realistic DSP computations and a commercial DSP target showing performance improvement over optimized baselines.

2 MOTIVATING EXAMPLE

This section shows how an example DSP kernel poses challenges to traditional compilers and how hardware-specific manual tuning can outperform them. We give an overview of how Diospyros's design can mimic the hand-tuning process.

Consider optimizing a fixed-size matrix convolution for a DSP. Embedded DSP applications typically rely on specialized kernel implementations for fixed, small data sizes—for example, a convolution with a 3×5 input matrix and a 3×3 filter:

```
for (oRow = 0; oRow < 5; oRow++) {
  for (oCol = 0; oCol < 7; oCol++) {
    for (fRow = 0; fRow < 3; fRow++) {
      for (fCol = 0; fCol < 3; fCol++) {
        fRT = 3 - 1 - fRow; fCT = 3 - 1 - fCol;
        iRow = oRow - fRT; iCol = oCol - fCT;
        if (iRow >= 0 && iRow < 3 &&
            iCol >= 0 && iCol < 5)
          o[oRow][oCol] += in[iRow][iCol] * f[fRT][fCT];
} } } }
```

The outer loops run 5 and 7 times because they iterate over the output matrix. This convolution "pads" the input matrix at the boundaries and produces a slightly larger output matrix.

In this example, we will optimize this convolution for the Tensilica Fusion G3 DSP [6], which has a 4-wide floating-point SIMD vector unit. SIMD instructions are critical in DSP programming for both performance and efficiency: they both enable parallelism and amortize the energy cost of fetching and dispatching instructions. While statically specifying the sizes allows Tensilica's vectorizing compiler to improve on this naive `for`-loop-based implementation by 1.6×, the best implementation we have found with Diospyros uses machine-specific vector intrinsics to achieve a further speedup of 22.9×. We explore why and how this gap arises in general for this

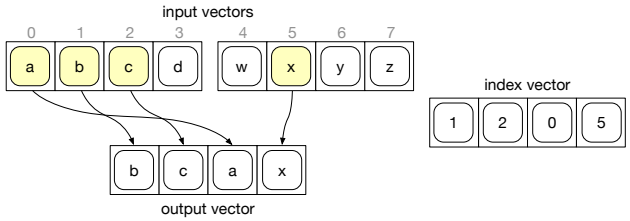


Figure 2: An ISA-specific shuffle instruction that takes three arguments—two input vectors and an index vector—and produces a single output vector with the specified combination of values. Experts can use similar instructions to orchestrate complex data movement strategies.

category of DSP kernels, where the problem dimensions are close to the vector width. Namely, for these kernels, boundary conditions make up a large proportion of the kernel's work, which hinders straightforward approaches to parallelization.

Traditional automatic parallelization. Two commonplace compiler techniques for vectorizing sequential code are loop-level vectorization and superword-level parallelism (SLP) optimizations [16]. For 2D convolution, the index math for transposing the filter (`fRT` and `fCT`) and the `if` for the boundary conditions pose a problem to loop-level vectorization. While loop-level vectorization works well when the data dimensions are large enough that there is a steady state that admits processing in 4-wide chunks, smaller loops do not have such a steady state. In this convolution example, no loop executes more times than twice the vector width—so every loop iteration is a boundary condition.

Because the array sizes for our problem are fixed, a compiler could unroll the loops and apply non-loop vectorization techniques such as SLP [16]. And indeed, specializing the array sizes leads to the aforementioned 1.6× speedup over a version with variable array sizes. However, this approach still leaves some performance on the table. Because the matrix dimensions (3×5 and 3×3) are close to the machine's vector width (4), SIMD instructions do not apply "cleanly" to the input arrays. Furthermore, the memory accesses to `f` are not contiguous, meaning that a simple vector load will not suffice to enable vectorized arithmetic. The Tensilica compiler's vectorization pass fails to find perfectly aligned runs of 4 identical operations, and it does not attempt to gather or shuffle disparate values to fill a vector. Alternatively, the `if` for the boundary condition means that a straightforward vectorized version will need to use predicated operations, wasting some potential computation bandwidth. Traditional vectorization optimizations rely on regularity in data movement and computation that is not present in specialized DSP kernels like this one, where loops are imperfect and data sizes are not much larger than the vector width.

Hand tuning. Instead, an expert programmer can use the Fusion G3's special instructions for data movement to pack computation into the vector lanes. The DSP supports gather/scatter and shuffle operations that pack data irregularly into vector registers for subsequent regular processing. For example, this intrinsic call:

```
int indices[4] = {1, 2, 0, 5};
xb_vecMx32 vec3 = PDX_SEL_MX32(vec1, vec2, indices);
```

computes a new 4-wide vector value by selecting specific hard-coded indices from the concatenation of two other vectors, `vec1` and `vec2`, as illustrated in Figure 2. The programmer can use this strategy to implement tactics for gathering data to fill vector lanes for later computation, like this multiplication:

```
xb_vecMx32 vec4 = PDX_MUL_MX32(vec1, vec3);
```

With judicious use of vector intrinsics and manual derivation of index operands, an expert implementation can surmount the limitations of traditional auto-vectorization. A manually tuned kernel can be an order of magnitude faster than the automatically parallelized version. However, the tuning required is specific to both the Fusion G3 target and the specific specialized size of the convolution kernel. A different vectorization strategy with completely different shuffle indices will be optimal for a 4×4 filter, for example.

Vectorization via rewriting. Diospyros uses term rewriting to search for DSP vectorization strategies that exploit this kind of irregular data layout techniques to optimize for vector unit utilization. Our system starts with an imperative reference implementation and, using symbolic evaluation (Section 3.1), extracts a specification describing the value to compute for each element of the kernel's output(s). For our convolution example, the specifications for the first four values of the output matrix are:

$$\boxed{i_{0,0} \times f_{1,1} + i_{0,1} \times f_{1,0} + i_{1,0} \times f_{0,1} + i_{1,1} \times f_{0,0}}$$
$$\boxed{i_{0,0} \times f_{1,2} + i_{0,1} \times f_{1,1} + i_{0,2} \times f_{1,0} + i_{1,0} \times f_{0,2} + i_{1,1} \times f_{0,1} + i_{1,2} \times f_{0,0}}$$
$$\boxed{i_{0,1} \times f_{1,2} + i_{0,2} \times f_{1,1} + i_{0,3} \times f_{1,0} + i_{1,1} \times f_{0,2} + i_{1,2} \times f_{0,1} + i_{1,3} \times f_{0,0}}$$
$$\boxed{i_{0,2} \times f_{1,2} + i_{0,3} \times f_{1,1} + i_{0,4} \times f_{1,0} + i_{1,2} \times f_{0,2} + i_{1,3} \times f_{0,1} + i_{1,4} \times f_{0,0}}$$

Here, the first expression is smaller because of the kernel's boundary condition. Diospyros uses a term rewriting system to find vectorization opportunities across these mathematical expressions. For example, the `vec_multiply_accumulate` rule can apply here to show that the above outputs are equivalent to expressing the last product in each element as a fused multiply–accumulate vectorized operation, VecMAC:

```
(VecMAC (...)
  (Vec (Get I 6) (Get I 7) (Get I 8) (Get I 9))
  (Vec (Get F 0) (Get F 0) (Get F 0) (Get F 0)))
```

Vec and Get are ISA-agnostic data movement abstractions that represent accessing the specified indices of a memory (with 2D arrays flattened to 1D access). Our full vector domain specific language is described in Section 3.1 and shown in Figure 3.

Due to the commutativity and associativity of + and ×, there are many possible shuffles a programmer could use to generate valid VecMAC operations. Diospyros uses an equality saturation approach to consider many possible shuffles—rather than applying destructive rewrites, as a traditional compiler would—and selects the pattern best suited to an abstract model of our architecture's data movement instructions. For example, here each Vec references the elements of only a single input array, which can be implemented with in-register data movement.

When targeting the Fusion G3, Diospyros produces this code for the vectorized expression:

```
shuf_I = PDX_SEL_MX32(I_4_8, I_8_12, [6, 7, 8, 9]);
shuf_F = PDX_SHUF_MX32(F_0_0, [0, 0, 0, 0]);
PDX_MAC_MFX32(out_0_4, shuf_I, shuf_F);
```

The full implementation that Diospyros generates for this problem size is 22.9× faster than a naive fixed-size implementation and 4.5× faster than an optimized vendor library kernel.

3 REWRITING FOR VECTORIZATION

Our core vectorization formulation uses equality saturation [36] to search for optimized implementations. This section describes the optimization workflow. Programmers write an imperative reference implementation using scalar operations, symbolic evaluation lifts this to an abstract vector DSL, then Diospyros searches for an optimal vectorized program using an equality saturation engine—trading off efficiency and completeness in the search. Next, Section 4 shows how Diospyros compiles the optimized program back to the imperative DSL to produce efficient code for the DSP target.

3.1 Defining and Lifting Specifications

Diospyros takes as input scalar programs written in a simple imperative language with first-class matrix and vector objects and operations, implemented as an embedded Racket DSL. For example, this code specifies a simple vector-vector add:

```
(define (vector-add-spec A B n)
  (vec-decl 'A n 'input)
  (vec-decl 'B n 'input)
  (define C (make-vector n))
  (for ([i n])
    (vector-set! C i
      (add (vector-ref A i)
           (vector-ref B i))))
  C)
```

Here, `A` and `B` are vectors of input data and `n` is a compile-time parameter that determines the input size.

This input language is both convenient to write and straightforward to compile to executable code for use in validation or testing. It supports arbitrarily complex indexing expressions and control flow, as long as they are independent of the input data. The input language provides the usual scalar arithmetic operations, such as `+`, but users can also define custom scalar functions to reflect a given target DSP and application.

While we could optimize this language directly (in the spirit of Denali [13]), doing so would conflate details of the imperative implementation with the underlying abstract mathematical computation. To focus on the latter and simplify the search, Diospyros first *lifts* imperative input programs into a mathematical representation. It symbolically evaluates the input program using Rosette [37], which extends Racket DSLs with symbolic evaluation support.

The symbolic evaluation step produces an expression in Diospyros's vector DSL, shown in Figure 3. The vector DSL includes both scalar and vector versions of common arithmetic operations (+, −, ×, etc.), as well as operations to initialize vectors with literals or variables and to extract individual vector lanes. The lifting process, however, only produces the scalar subset of the language—the rewriting system in the next section will use the vector constructs. Lifting supports calls to user-defined functions by introducing uninterpreted functions. The same symbolic evaluation engine also powers the translation validation tool that Diospyros uses to verify its output (see Section 3.4).

$$
\begin{aligned}
\langle prog \rangle &::= (\text{List } \langle expr \rangle^{+}) \mid \langle expr \rangle \\
\langle expr \rangle &::= \langle scalar \rangle \mid \langle vector \rangle \\
\langle scalar \rangle &::= \langle integer \rangle \mid \langle variable \rangle \\
&\mid (+ \langle scalar \rangle \langle scalar \rangle) \mid (- \langle scalar \rangle \langle scalar \rangle) \\
&\mid (* \langle scalar \rangle \langle scalar \rangle) \mid (/ \langle scalar \rangle \langle scalar \rangle) \\
&\mid (\text{sgn } \langle scalar \rangle) \mid (\text{sqrt } \langle scalar \rangle) \mid (- \langle scalar \rangle) \\
&\mid (\text{Get } \langle variable \rangle \langle integer \rangle) \\
&\mid (\langle func \rangle \langle scalar \rangle^{*}) \\
\langle vector \rangle &::= (\text{Vec } \langle scalar \rangle^{+}) \mid (\text{Concat } \langle vector \rangle \langle vector \rangle) \\
&\mid (\text{VecAdd } \langle vector \rangle \langle vector \rangle) \mid (\text{VecMinus } \langle vector \rangle \langle vector \rangle) \\
&\mid (\text{VecMul } \langle vector \rangle \langle vector \rangle) \mid (\text{VecDiv } \langle vector \rangle \langle vector \rangle) \\
&\mid (\text{VecMAC } \langle vector \rangle \langle vector \rangle \langle vector \rangle) \\
&\mid (\text{VecSgn } \langle vector \rangle) \mid (\text{VecSqrt } \langle vector \rangle) \\
&\mid (\text{VecNeg } \langle vector \rangle) \\
\langle func \rangle &::= \langle symbol \rangle
\end{aligned}
$$

Figure 3: Diospyros's vector DSL. A top-level program is a (possibly singleton) list of outputs. Expressions operate over both scalars and vectors.

To expose vectorization opportunities for the rewriting system, the lifting process converts matrix and vector outputs into a single List output term, with one element for each value in the program output. For example, the vector-vector add above with `n` = 2 lifts to this expression:

```
(List
  (+ (Get a 0) (Get b 0))
  (+ (Get a 1) (Get b 1)))
```

Here, `Get` is list access and `List` constructs a new output list holding the two elements of the output vector.

3.2 Rewriting Strategy

To vectorize the lifted program in the abstract DSL, Diospyros uses a family of built-in (though user-extensible) rewrite rules. The key equivalence that enables vectorization is that the rewrite rules consider a List to be equivalent to a concatenation of fixed-size vectors. For example, Diospyros can rewrite our vector-vector add with `n` = 4 and a vector width of two this way:

```
(List (+ (Get a 0) (Get b 0))
      (+ (Get a 1) (Get b 1))
      (+ (Get a 2) (Get b 2))
      (+ (Get a 3) (Get b 3)))
⇝
(Concat (Vec (+ (Get a 0) (Get b 0))
             (+ (Get a 1) (Get b 1)))
        (Vec (+ (Get a 2) (Get b 2))
             (+ (Get a 3) (Get b 3))))
```

Vec constructs a vector from a configurable machine–width number of scalar values (here, two), and Concat concatenates two vectors into a list. In real DSP code, they correspond to vector load and store instructions (see Section 4). Diospyros's rewrite rules can pad lists with zeros if their lengths are not a multiple of the vector width.

This rewriting into vector-sized chunks creates opportunities to use vectorized computation. The rewrite system finds Vec expressions that contain similar scalar expressions and replaces them with

their vectorized equivalents. For example, the rule for introducing vectorized add instructions, VecAdd:

$$(\text{Vec } (+\ a\ b)\ (+\ c\ d)) \rightsquigarrow (\text{VecAdd } (\text{Vec } a\ c)\ (\text{Vec } b\ d))$$

applies twice to the example above, producing:

```
(Concat (VecAdd (Vec (Get a 0) (Get a 1))
                (Vec (Get b 0) (Get b 1)))
        (VecAdd (Vec (Get a 2) (Get a 3))
                (Vec (Get b 2) (Get b 3))))
```

Here, the indices in the Get expression determine the data movement strategy required for this program. In this case, the pairs of indices 0, 1 and 2, 3 can each be implemented by a vector load without additional data movement. This example is now fully vectorized because all Vec expressions contain simple memory lookups and no scalar computations expressions remain.

Diospyros's code generation backend (Section 4) produces DSP code from this vectorized program by emitting C intrinsics resembling this pseudocode:

```
vecreg a_0_2 = load(a, 0, 2);
// ...
vecreg b_2_4 = load(b, 2, 2);

vecreg add_1 = vec_add(a_0_2, b_0_2);
vecreg add_2 = vec_add(a_2_3, b_2_4);

store(out, add_1, 0, 2);
store(out, add_2, 2, 2);
```

While this simple example has perfectly aligned vector accesses, most realistic code requires nontrivial data movement to fill the vector registers. Diospyros's backend consumes these Vec expressions to produce actual loads and data movement instructions based on the high-level strategy found by the rewrite engine. During code generation, the backend selects vector shuffle code to implement each given Vec expression. Similarly, real code mixes both vector and scalar computation; Diospyros generates a mixture of both.

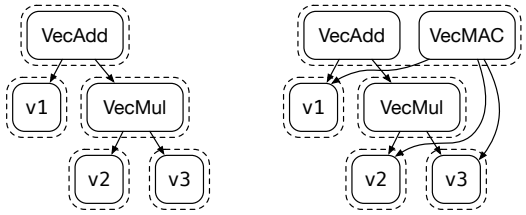
3.3 Searching for Rewrites

In general, applying the rewrite rules directly (like a traditional compiler) does not promise optimality—we must be sure to apply the right rules in the right order to find the optimal program (with respect to our rule set). This section describes how Diospyros searches the space of all rewrite rule applications by representing the lifted program as an *equality graph* (*E-graph*) [23] and using *equality saturation* [36] for efficient search.

Equality saturation. An E-graph is a data structure for efficiently representing a large set of terms and equivalences between them. The nodes of an E-graph are function symbols or terminals, and subgraphs represent terms. Each node is associated with an equivalence class, and the E-graph guarantees that two nodes are in the same equivalence class if and only if the program terms rooted at them are equivalent. When used for program optimization, the equivalence relation is program equivalence.

Initially, the E-graph represents only one program and its subterms (the input program in the abstract DSL). Equality saturation then applies rewrite rules (program transformations) to the E-graph, which introduces new nodes into the graph and annotates them with the appropriate equivalence classes to maintain congruence. For example, this is a rewrite rule for fused multiply–accumulate:

$$(\text{VecAdd } a\ (\text{VecMul } b\ c)) \leftrightsquigarrow (\text{VecMAC } a\ b\ c)$$



(a) Before rewriting **(b) After rewriting**

Figure 4: An E-graph before and after applying a rewrite rule for fused multiply–accumulate. Solid boxes are nodes and represent program terms. Dashed boxes represent equivalence classes. After rewriting, the VecAdd and VecMAC terms are in the same equivalence class.

Figure 4 illustrates the application of this rewrite rule to an E-graph which initially represents the program (VecAdd `v1` (VecMul `v2` `v3`)). Applying the rule introduces a new VecMAC node into the graph, with the variables `v1`, `v2`, and `v3` as children, and adds the new node to the equivalence class of the existing VecAdd node.

Equality saturation iteratively applies all rewrite rules (possibly multiple times), terminating when no potential rewrite rule application would change the graph—the graph has *saturated*—or a timeout is reached. At this point (unless the timeout is reached), the saturated E-graph represents all programs that could be produced by applying the rewrite rules in any order. This property allows us to avoid the phase ordering problem common to compilers.

We use the egg [40] library for E-graphs and equality saturation. In egg, a rewrite rule comprises two parts: a *searcher* that looks for nodes that can be rewritten, and an *applier* that applies a rewrite. egg exposes a pattern DSL to specify simple syntactic rewrites and a Rust API to implement custom searchers and appliers with more complex logic than simple pattern matching.

Custom matching for vectorization. Simple unary scalar operations can be vectorized using rules of the form shown in Section 3.2. However, DSP kernels often do not fit exactly within the target architecture's vector lanes (for example, a 3×3 matrix multiply on an architecture with vector width 4). To vectorize operations while maximizing hardware utilization, Diospyros provides rewrite rules that work even when some lanes of a vector computation are empty. For example, the following concrete rewrite is sound and enables vectorizing an addition with irregular shape:

$$(\text{Vec } (+\ a\ b)\ 0\ (+\ c\ d)\ 0) \rightsquigarrow (\text{VecAdd } (\text{Vec } a\ 0\ c\ 0)\ (\text{Vec } b\ 0\ d\ 0))$$

To avoid specifying every permutation of zeros on the left-hand side of this rule, and repeating this specification for each binary operation, Diospyros uses egg's support for custom rewrite rules that go beyond pattern matching. The custom rule first matches on the outer vector and then identifies whether each lane matches either the operator pattern ($\langle op \rangle\ x\ y$) or chosen concrete values (in this case, a constant zero). Using these custom rules makes it easier to extend Diospyros with DSP-specific instructions without developing a comprehensive new rewrite rule family.

Associativity & commutativity. A common challenge in rewrite systems is handling operators that are associative or commutative (or both). For example, we want this rewrite:

$$(+\ (+\ a\ b)\ 0) \leftrightsquigarrow (+\ a\ b)$$

to also apply to associative or commutative variants of the LHS such as $(+\ a\ (+\ b\ 0))$. But applying associative and commutative variants of such rules to saturation dramatically increases the size of an E-graph; the decision problem of whether two terms can be unified modulo associativity and commutativity (the *AC-matching* problem) is NP-complete [4]. This theoretical problem is also a scalability challenge for equality saturation in practice [22].

Diospyros addresses AC-matching by optionally allowing users to disable associativity and commutativity rules during saturation. This approach sacrifices completeness in terms of missing some potential rewrites, but reduces memory requirements and thus allows Diospyros to compile kernels with deeper syntax trees over associative and commutative operators. To regain some of the power of associativity and commutativity, we use more complex rewrite rules to selectively re-enable some limited forms of AC rules that we have found to be profitable in practice.

For example, consider the following 4-wide vector:

```
(Vec (+ a0 (* b0 c0))
     (+ a1 (* b1 c1))
     (+ a2 (* b2 c2))
     (+ (* b3 c3) a3))
```

We would like to optimize the scalar operations in this vector into a single vectorized multiply–accumulate. However, without a general commutativity rule for +, the fourth lane prevents introducing a VecMAC operation. We work around this limitation using a custom searcher that matches on each lane independently with one of several pattern options, and then combines the results. For vector multiply–accumulate, each lane must match one of these patterns:

$$(+\ a\ (*\ b\ c)) \qquad (+\ (*\ b\ c)\ a) \qquad (*\ b\ c) \qquad 0$$

The applier (right-hand side) of this rule collects the arguments into vectors (mapping "missing" values to zero) and applies the fused operation:

```
(VecMAC (Vec a0 a1 a2 a3)
        (Vec b0 b1 b2 b3)
        (Vec c0 c1 c2 c3))
```

Unlike an approach that includes AC rules when saturating the E-graph, this custom searcher approach does not persist its discovered equivalences. This difference trades off memory for compute: rather than persisting these equivalences in the E-graph, we re-compute them every time we try to apply the custom searcher. In practice, we have found this to be a worthwhile trade-off, allowing larger kernels that previously exhausted the memory of a 512 GB host to successfully compile. We expect that similar customizations for AC searching would be beneficial in a variety of domains beyond vectorization.

Floating point accuracy. Diospyros's rewrite rules are correct with respect to the real numbers. They do not adhere to strict floating point semantics which, for example, would not allow associativity in addition or multiplication. Diospyros shares this characteristic with other modern optimizing compilers for compute kernels that prioritize speed over numerical stability [14, 29]. We measure floating point error in our evaluation (Section 5) and find Diospyros-generated code to match reference implementations within several decimal places.

3.4 Extraction

After equality saturation completes, Diospyros has a single E-graph representing many programs that are equivalent to the input program (according to the rewriting system). Each program would be a valid solution to the compilation problem, but we want to extract the most efficient solution. We cannot explicitly enumerate the programs to search for an optimal one—doing so would sacrifice the compactness of the E-graph representation. Prior equality-saturation-based superoptimizers [13] extract efficient code by generating cost-related verification conditions from the E-graph and discharging them with a SAT solver, but this requires a detailed architecture-specific cost model.

Diospyros extracts an efficient solution from the E-graph using a cost model that assigns a fixed cost to each operator in the vector DSL. This cost model reflects the time and energy savings of vectorization as well as the cost of reading values from registers versus memory. To support efficient extraction from the E-graph (linear in the number of E-graph nodes rather than the number of candidate programs), this cost function must be strictly monotonic, i.e., an expression's cost is greater than the sum of the costs of its subexpressions. This limitation makes extraction efficient because it avoids the need to explore all the zero-cost variants of a candidate expression. While this restriction limits the cost models Diospyros can express, in our experience we can still extract fast programs, as Section 5 demonstrates.

Our cost model for data movement is intentionally high-level—Diospyros assigns a lower cost to shuffles that gather data from a single input array (or zeros) than to shuffles across different inputs or non-zero scalars. The Fusion G3's fast, unrestricted shuffle instruction allows this abstract cost model to serve as a good proxy for data movement costs. This approach may be a poorer fit for architectures without support for flexible shuffles (Section 6 discusses this limitation further).

Timeouts. Saturating an E-graph guarantees that it captures all possible orderings of the rewrite rules. In practice, saturation can be very expensive, and so we impose both a wall-clock timeout and an E-graph node limit to terminate early. Diospyros can still produce a solution from a timed-out compilation by applying the above extraction process to the partially saturated E-graph. Half of our benchmarks in Section 5 time out, and yet most still outperform optimized libraries. Section 5.5 studies the impact of timeouts on the quality of Diospyros's output.

Translation validation. Diospyros depends on a set of rewrite rules to define the search space of equivalent programs. The equality saturation engine trusts these rules; while most rules are simple, an incorrect one can cause Diospyros to miscompile a program. We address this risk by re-using the symbolic evaluation engine from Section 3.1. We use this engine to optionally perform translation validation on the final extracted program, using Rosette [37] to

prove that the extracted program is equivalent to the input one for all possible inputs.

The validation assigns no semantics to the uninterpreted functions that represent user-defined functions, and so programs involving them may produce spurious validation failures (for example, we would not know that a user-defined `square` function only produces non-negative values). The user can optionally provide (possibly partial) semantics for user-defined functions as a Racket function, which Rosette lifts to operate on symbolic inputs and uses to validate translations. These semantics are used only at the translation validation stage and not by the rest of the compiler.

Translation validation removes the equality saturation engine and the rewrite rules themselves from the trusted computing base of the compiler. However, the validation is between two programs in the vector DSL, and so both the initial lifting from imperative code into that DSL and the backend code generation (Section 4) are still trusted. Diospyros's translation validation models values in the theory of real arithmetic, rather than with precise floating point semantics. Anecdotally, we have found translation validation very useful when developing and debugging new rewrite rules and vector DSL extensions.

4 LOWERING & CODE GENERATION

After extraction from the E-graph, we are left with a vectorized program in an idealized vector DSL. This section describes how Diospyros compiles this program, first to a lower-level vector IR and then to C++ specific to the target DSP architecture.

Abstract vector IR. To capture the essence of vector computation with data movement, the Diospyros backend defines a machine-independent vector intermediate representation (IR). At this abstraction level, kernels operate on user-specified input arrays to produce outputs using an imperative language free of control flow. The IR includes common vectorized operations such as memory loads and stores, arithmetic, and data shuffles, as well as user-defined uninterpreted functions for both scalar and vector operations. While the IR is at a fairly low level of abstraction, it abstracts away concrete details of the DSP architecture, deferring them to a later architecture-specific instruction selection phase (Figure 1).

One key challenge to solve at this compilation step is how to translate instances of Vec in the vector DSL. Vec terms represent vector initializations, and each vector lane can be populated from an arbitrary memory location. For example, the quarternion product benchmark we evaluate in Section 5 includes a Vec term in its output of the form:

```
(Vec (Get a 1) (Get a 2) (Get a 0) (Get a 3))
```

To initialize this vector, the backend IR includes a vector *shuffle* operation:

```
(vec-shuffle inputs indices)
```

that takes as input an array of `indices` defining where to move each element of `inputs`. The IR does not restrict the possible values of `indices`, offering the flexibility to compile vectorization patterns discovered by equality saturation that require complex data movement. Lowering this instruction to the target DSP architecture requires selecting an instruction sequence that achieves this desired movement using the architecture's available shuffle operations.

IR-level optimization. Diospyros's compilation flow includes fully unrolling loop nests, which can create very large programs with redundant terms. This redundancy is not an issue during equality saturation, because the E-graph representation implicitly deduplicates redundant terms. However, a naive lowering from the high-level vector DSL would include this redundancy and produce kernels far too large for resource-constrained targets. The Diospyros backend implements a local value numbering (LVN) pass to eliminate redundant terms. This pass is highly effective: for the quarternion product benchmark in Section 5, it reduces the output size from over 100,000 lines of C++ to under 500 lines.

Instruction selection. The final phase of compilation is to perform instruction selection for a concrete architecture. Diospyros delegates much of this work to the vendor-supplied DSP compiler toolchain, avoiding the need to integrate deep target-specific knowledge into Diospyros for each new DSP target architecture. The lowering phase translates the low-level IR into C++ compiler intrinsics that are then compiled with the DSP toolchain. The programmer can provide the name and type signature of target-specific instructions for both scalar and vector operations.

5 EVALUATION

Our evaluation has two main components: a demonstration of speedups for individual kernels compiled with Diospyros (Section 5.4), and a more detailed examination of an application that can benefit from replacing library calls to fixed-sized linear algebra kernels with Diospyros kernels (Section 5.7).

5.1 Implementation

Diospyros currently targets Tensilica's Xtensa Fusion G3 family of DSP architectures [6]. The backend lowers the `vec-shuffle` instruction in the low-level IR to the Xtensa `PDX_SHFL_MX32` (single-register shuffle) and `PDX_SEL_MX32` (two-register select) intrinsics. To implement arbitrary shuffles with more than two registers, Diospyros uses nested select instructions.

Diospyros's implementation spans two languages. 4,800 lines of Racket, using the Rosette framework [37], implement the domain-specific vector languages, lifting, translation validation, and backend compilation phases. 1,400 lines of Rust implement the rewrite rules and cost model using the egg [40] equality saturation library.

5.2 Methodology

We report cycle counts from Tensilica's cycle-level simulator for the Fusion G3 DSP processor [6], `xt-run`. We use use `xt-run`'s default memory model, which assumes an ideal, unit-delay memory for all accesses. The simulator is deterministic, so we report results for a single execution. We compile all implementations (baseline loops, library-provided functions, and Diospyros-generated code) with the `xt-xcc`/`xt-xc++` compiler from the Tensilica Xtensa SDK

Table 1: Benchmark kernels used in the evaluation. We list the lines of code in the reference implementation and show the time and maximum memory used for compilation, including symbolic evaluation, optimization, and code generation but not translation validation.

Benchmark	Description	Ref. LOC	Size	Time	Memory
2DConv	2D convolution	131	3×3, 2×2	2.2s	145 MB
			3×3, 3×3	5.6s	145 MB
			3×5, 3×3	30.3s	626 MB
			4×4, 3×3	23.8s	370 MB
			8×8, 3×3	3m 16s†	3.8 GB
			10×10, 2×2	21.6s	401 MB
			10×10, 3×3	3m 24s†	4.1 GB
			10×10, 4×4	3m 11s†	5.0 GB
			16×16, 2×2	1m 8s	1.2 GB
			16×16, 3×3	3m 9s†	4.7 GB
			16×16, 4×4	3m 57s†	4.4 GB
MatMul	matrix multiply	71	2×2, 2×2	1.9s	144 MB
			2×3, 3×3	2.2s	136 MB
			3×3, 3×3	2.7s	124 MB
			4×4, 4×4	5.8s	130 MB
			8×8, 8×8	3m 22s†	4.0 GB
			10×10, 10×10	3m 30s†	6.0 GB
			16×16, 16×16	3m 38s†	4.5 GB
QProd	quaternion product	144	4, 3, 4, 3	6.7s	128 MB
QRDecomp	QR matrix decomposition	174	3×3	4m 38s†	2.2 GB
			4×4	4h 25m†	35.4 GB

† Equality saturation timed out after 180s.

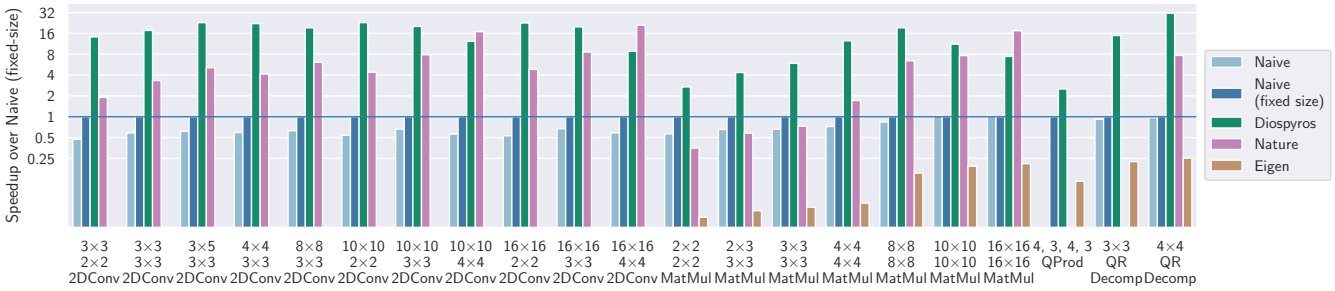


Figure 5: Speedup over Naive (fixed size) in simulated cycles, log scale. Bars above the blue line indicate a speedup. *Naive* is a naive loop nest, *Naive (fixed size)* is a loop nest with fixed bounds, *Diospyros* is our system, *Nature* is a vendor-supplied library function, and *Eigen* is a C++ template linear algebra library.

at the highest optimization level, `-O3`.[2] We run experiments on a machine with two Intel Xeon E5-2620v4 CPUs running CentOS 7.6. We give Diospyros a 3-minute timeout for equality saturation with a node limit of 10,000,000. We run without full associativity and commutativity enabled (as described in Section 3.3).

[2]Tensilica also provides a second compiler, called `xt-clang++`, that is not well-documented in our version of the Xtensa SDK. Xtensa specifies that `xt-clang++` does not include a loop transformation framework, such as the one in `xt-xc++` at the `-O3` optimization level; however, it does perform better on some scalar code due to more aggressive inlining and a different software pipelining scheduler. To ensure a consistent baseline, we use the better documented, default `xt-xc++` compiler.

We compare Diospyros with the Nature DSP library included with Tensilica's SDK. Nature is optimized specifically for the Fusion G3 using vector intrinsics, so it performs better than naive C++; however, the library's performance is limited by the need to be generic over matrix sizes. Not all sizes have Nature comparisons because the library often restricts dimensions to multiples of 4 to match the machine vector width. We also compare with Eigen [12], a portable (not Xtensa-optimized) C++ template library for linear algebra, where available. Although Nature and Eigen are the competitive baselines, we also include straightforward loop-nest-based

implementations for reference: one with parametric sizes and one with sizes fixed at compile time (with `#define`). Figure 5 normalizes simulated cycle times as speedups over the fixed-size naive baseline.

5.3 Kernel Benchmarks

Table 1 lists the benchmark kernels we use, which are inspired by use cases in computer vision and machine perception. QProd, for instance, is a Euclidean Lie group product [32], which includes quaternion and translational product components and appears in applications such as pose estimation or camera models.

The table also shows the total compilation time for each benchmark. While we set the timeout for equality saturation at just 3 minutes, some benchmarks take a significant amount of time to do backend optimization and code generation. QRDecomp at the 4×4 size is a pathological case. The kernel when fully unrolled is extremely large: the extracted specification alone is a 509 MB text file. As a result, the E-graph does not saturate and it finds no vector instructions. The expression is heavily redundant, so our post-processing optimizations (Section 4) take several hours and gigabytes of memory to remove redundancy before generating output program, producing only 457 lines of C as output. Here, the performance benefits of the additional common subexpression elimination enabled by symbolic evaluation (and exploited by our local value numbering optimization) are enough to beat the naive and library implementations, even without vectorization. We discuss this effect further in Section 5.6.

5.4 Kernel Performance Results

Figure 5 compares the Diospyros-generated kernels against straightforward loop-based implementations (with both parametric sizes and inlined fixed sizes to facilitate more aggressive `-O3` optimizations) and the Nature DSP and the Eigen library functions. On average, Diospyros-optimized kernels outperform the best non-expert baseline by 3.1×.

The Diospyros-generated matrix multiply kernels are between 2.7× and 19.3× faster than the fixed-size naive loop nests. The trends in Figure 5 indicate that even highly-optimized code such as Nature can perform poorly on small kernels, such as the 2×2 square matrix product, due to the control overhead of the parametrized unrolling.

In the case of 2DConv, our example from Section 2, Diospyros finds solutions that are up to 7.5× faster than the library implementations. Nature outperforms Diospyros on 2DConv at two sizes that are greater than or equal to the vector width: input sizes 16×16 and 10×10, with filter size 4×4. The Nature library's 2D convolution makes extensive use of vector intrinsics for loads, stores, and arithmetic operations; however, its unrolling strategies are not amenable to cases where the filter size is near but not equal to the vector width.

In the case of matrix multiply, we also have access to proprietary hand-tuned code written for the Fusion G3 by a DSP expert for a single fixed size, 2×3 by 3×3. The Diospyros-generated kernel compiles with full equality saturation in 2.7 seconds and produces runtime performance that is within 8% of the expert performance (39 vs. 36 cycles). The Diospyros kernel and the expert kernel perform the same number and type of vector operations (two multiplies and four multiply–accumulates), but Diospyros's logic to load elements into registers from main memory is less efficient. We believe this performance gap could be eliminated with additional engineering effort in improving code generation.

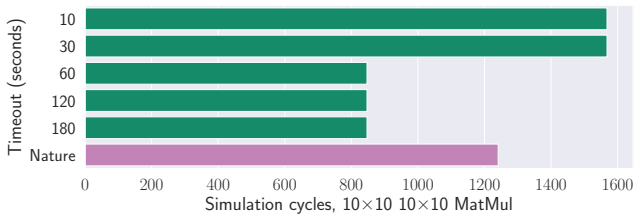


Figure 6: Effect of search timeout on MatMul performance.

5.5 Timeout Ablation Study

Diospyros's rewrite engine uses a timeout to emit suboptimal solutions even when it does not reach full equality saturation. Shorter timeouts stop Diospyros from completely vectorizing the kernel but still emit an executable C kernel. Figure 6 shows the effect of increasing the timeout on our MatMul benchmark for the largest size, 10×10 by 10×10. With a 10-second timeout, the Diospyros generated kernel performs far better than a naive kernel (1,568 cycles), but not as well as the size-agnostic implementation in the Nature library (1,241 cycles). Increasing the timeout improves the quality of the generated benchmark, ultimately saturating the E-graph and finding a kernel that beats even the Nature library taking 847 cycles. This formulation allows programmers to trade off compilation time for runtime performance of the generated kernel.

5.6 Vectorization Ablation Case Study

As the results for QRDecomp at the 4×4 size demonstrate, symbolic evaluation alone enables loop unrolling and common subexpression elimination that yield performance benefits even without explicit vectorization. To isolate the performance advantage of our vectorization strategy over other factors, we measure performance for Diospyros with all vector rewriting rules disabled. Compiling kernels with Diospyros without these vector-related rules (but with symbolic evaluation, scalar rewrite rules, and common subexpression elimination) yields code that performs 2.2× better than the best non-Diospyros baseline, compared to 3.1× with vector rewrite rules. In 4 out of 21 kernels, the non-vectorized code is actually faster than the Diospyros-vectorized code because the vendor's compiler can produce more heavily optimized scalar code. We believe Diospyros could improve on these cases with a better cost model that reflects the overheads of vector packing and engineering enhancements to the backend code generation.

5.7 Application Case Study

We implement a piece of a digital signal processing application that can use Diospyros-generated kernels to observe their effect in context. Sensing applications such as structure from motion (SFM) [35] are rich with small-scale linear algebra kernels calls that can become bottlenecks if they are implemented in a generic way.

This section studies a camera model computation from the Theia open-source SFM package [35], which is representative of the kinds of embedded vision workloads that are common on DSPs. Theia is

well optimized and uses the popular Eigen [12] library of matrix kernels, but it is not specifically optimized for DSP architectures. It uses a camera model to define how points in 3D space project into a 2D image plane captured by the sensor array. We focus on this initialization function in Theia's camera model:

```
bool Camera::InitializeFromProjectionMatrix(
  const int image_width,
  const int image_height,
  const Matrix3x4d projection_matrix)
```

The core functionality is in `DecomposeProjectionMatrix`, a function that initializes camera parameters projecting to a rotation matrix using a Jacobi SVD decomposition and then decomposing the matrix using RQ decomposition. We port `DecomposeProjectionMatrix` to Tensilica's Fusion G3 DSP. We compare against a version using single-precision floating-point numbers (the original code uses double-precision FP, but both the original and our optimized versions are accurate within 10^{-6} even with single precision). We found that 61% of the run time was spent on a call to a 3×3 QR decomposition from the Eigen library.

We substitute a QR decomposition kernel generated by Diospyros for the Eigen implementation to measure its effect on the overall computation. QR decomposition is a linear algebra kernel that takes as input a square matrix A and finds a right triangular matrix R and an orthogonal matrix Q such that $A = Q \times R$. Both Eigen and our implementation use the Householder algorithm to iteratively build both outputs, using a series of matrix multiplications along with scalar computations. The number of floating point multiplications is cubic in relation to the matrix size. We implement QR decomposition with about 170 lines of imperative Racket. The resulting SMT-based specification has over 65,000 calls to floating point multiply, demonstrating the complexity of this kernel.

For the complete projection matrix computation, the Diospyros-optimized version is 2.1× faster than the original Eigen-based implementation (30,552 vs. 64,025 cycles). The QR decomposition kernel alone is an order of magnitude faster than Eigen's implementation (see Section 5.4), and these savings translate to a substantial speedup in the complete computation.

6 LIMITATIONS & PORTABILITY

While Diospyros's design aims to generalize across DSP architectures, we built the prototype in this paper to target the Tensilica Fusion G3 specifically. Aspects of the rewriting strategy in Section 3.2 reflect the Fusion G3's ISA: namely, the vector width, the available vector arithmetic operations, and the support for flexible "shuffle" instructions for data movement. However, Diospyros's equality saturation engine is parametric over most of these target details—for example, a simple compile-time setting controls the target vector width.

To target a different DSP, a designer would need to add or remove rewrite rules that reflect the available primitive operations. For example, consider a DSP with a vectorized fast reciprocal operation. To add support for this instruction, a user would need to: (1) add a scalar rewrite rule like (/ 1 x) ⇝ (recip x), relying on existing support for division; (2) inform the rewrite engine that recip has a vector equivalent, using a rule builder available in the Diospyros library; and (3) add the target-specific intrinsic to the backend (1–2 lines of code to map VecRecip to the vendor intrinsic).

An important assumption in Diospyros is that the target can support flexible data movement between vector registers. Its rewrite rules allow unrestricted data movement during equality saturation, with a relatively abstract cost model that assigns a higher cost to gathering data across different inputs or from non-zero scalars. We expect this approach to be most appropriate for architectures with a flexible "shuffle" instruction that uses an index vector to change positions within a vector. For architectures without this kind of flexible data movement, the backend would need to fall back to scalar operations more frequently, which would be more expensive.

7 RELATED WORK

Vectorizing compilers. Classical vectorization techniques—from loop dependency analysis [1] to modern auto-vectorization techniques [17, 19, 24]—typically do not attempt to aggressively shuffle data into irregular patterns. Existing techniques prioritize efficient compilation over optimality: they are designed to run on millions of lines of code but miss vectorization opportunities.

Previous work has used the Halide language [29] to target DSPs, but has not supported exploration of a large search space of irregular data movement strategies [39]. Other approaches can generate target-specific shuffles to implement known permutations, but do not find the permutation strategies themselves [9, 18]. Our search strategy can discover novel shuffles and data movement, automating the labor-intensive hand-tuning process at the cost of increased compilation time.

SLinGen [31], part of the SPIRAL project [10, 28], optimizes small linear algebra kernels by first applying optimizations like loop reordering and vectorization and then autotuning. Like SLinGen, Diospyros works at a higher abstraction level to enable optimizations that would not be apparent at the assembly level. However, our work uses equality saturation both to avoid hand-crafting specific optimization patterns (including for custom functional units that are common on DSPs) and to offer higher coverage of the search space than autotuning.

Program synthesis. Program synthesis techniques can expend compilation time to discover novel optimized programs. Barthe et al. [3] develop an auto-vectorizer using inductive synthesis but focus on general-purpose code rather than linear algebra and so do not generate shuffles. Cowan et al. [7] generate quantized machine learning kernels using syntax-guided synthesis. Their sketches exploit the reduction structure of these kernels and so cannot invent new data movement. MSL [41] is a synthesizer that generates bulk-synchronous parallel programs. The synthesizer reduces the parallel problem to a sequential one, uses a syntax-guided synthesis tool [30] to solve the sequential problem, and then compiles the result to message-passing parallel code.

Swizzle Inventor [27] infers permutations of data and computation (*swizzles*) that are optimized for GPU memory hierarchies. Unlike Swizzle Inventor, Diospyros has the ability to change the compute code itself (e.g., by fusing multiply–accumulates) rather than just the data movement. Swizzle Inventor also requires users to provide a sketch identifying the sites of possible swizzles; Diospyros's rewrite rule system does not require sketching.

Unlike many synthesis techniques, Diospyros has the ability to extract *partial* solutions if the synthesis process takes too long. Recent work [26] explores synthesis techniques that are *best effort*, returning partially valid solutions. Diospyros's rewrite rules are sound, and so the partial solutions it returns are always valid, but the partial solutions are not provably optimal (even with respect to the limited rewrite rules). This design allows Diospyros to avoid expensive optimality proofs that can dominate synthesis time [5]. Incorporating unsound rewrite rules that can be repaired at code generation time is an appealing direction for future work.

An earlier version of Diospyros [38] relied on syntax-guided synthesis backed by an SMT solver. It generated optimized linear algebra kernels but encountered scaling issues even on small (2×2) kernels because it needed to reason about bit-level instruction semantics during synthesis. Diospyros now abstracts away arithmetic semantics and focuses on vectorization by using term rewrite instead, so it can scale to kernels 10× larger than the SMT-based version. In addition, the previous version of Diospyros required a full program sketch in addition to a specification for each kernel. The current Diospyros system allows users can reuse the same rewrite rules across different kernels.

Term rewriting systems. Diospyros's optimization approach is based on equality saturation [36, 40], a technique for optimizing compilation using equality graphs (E-graphs). Equality saturation alleviates the phase ordering problem of traditional compilers by applying rewriting rules to an E-graph, implicitly capturing all possible phase orderings. Recent work expands equality saturation to new compilation domains such as CAD models [22]. These approaches exploit the insight that equality saturation does not require backtracking so it admits an asymptotically more efficient E-graph implementation [40]. Diospyros instantiates this approach for vectorization, using equality saturation to exhaustively search candidate vectorized programs that include data movement.

Denali [13] is an equality saturation-based superoptimizer for Alpha assembly code. It saturates an E-graph using assembly-level rewrite rules and then extracts an optimal program by using a SAT solver to compute a detailed cost model. Diospyros's rewriting happens instead over an abstract DSL, which sacrifices some target-specific optimality in favor of reasoning about data movement; such higher-level optimizations are typically where expert DSP developers focus their hand-tuning efforts.

8 CONCLUSION

Diospyros combines symbolic evaluation, equality saturation, and translation validation to build an end-to-end compiler for high-performance DSP code. Diospyros is extensible: users can bring domain- and architecture-specific insights by adding new rewrite rules to the equality saturation scheme. A main avenue for future work is to exploit this flexibility to target more DSP targets and other esoteric, customizable hardware architectures beyond DSPs.

ACKNOWLEDGMENTS

We thank Jacob Delgado-López for his implementation contributions and Armin Alaghi and Max Willsey for early feedback on this work. Many thanks to the anonymous ASPLOS reviewers and our shepherd, Shoaib Kamil, for their detailed feedback.

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. It is also partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). Support included NSF awards #1845952 and #1723715. This material is based upon work supported by the NSF Graduate Research Fellowship Program under Grant No. DGE-1650441. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A ARTIFACT APPENDIX

Our artifact packages an environment to reproduce the results presented. Specifically, we package:

- The Diospyros compiler: A search-aided compiler for generating vectorized DSP kernels.
- Implementation of benchmarks in Diospyros.
- Implementation of the Theia open-source application case study.

The goal of our evaluation is to reproduce the claims presented in the paper (Table 1, Figure 5, Figure 6) and to demonstrate the robustness of our tool.

Note on proprietary tools. Our evaluation uses two proprietary pieces of software: (1) the Tensilica G3 DSP simulator, and (2) the Nature library that implements DSP primitives. These tools require licenses and are not freely available. For the purposes of artifact evaluation, we have made our research server available to reviewers so that they can reproduce our studies (with permission from the ASPLOS AEC chairs).

A.1 Artifact Meta-Information

- **Algorithm**: solver-aided compilation, equality saturation, symbolic execution, vectorization
- **Hardware**: No hardware requirements for reviewer; our evaluation targets the Tensilica Fusion G3 digital signal processor (DSP) but we provide SSH access to a server with the necessary tools installed.
- **Output**: C/C++ code with intrinsics; the figures from the paper.
- **Experiments**: Implementation of Diospyros benchmarks, Theia case study.
- **How much disk space required (approximately)?**: 20 GB
- **How much time is needed to prepare workflow (approx.)?**: 15 minutes.
- **How much time is needed to complete experiments (approx.)?**: 2-3 hours for required components, an additional 4.5 hours for reproducing complete results.
- **Publicly available?**: Yes.
- **Code licenses (if publicly available)?**: MIT License
- **Archived (provide DOI)?**: 10.5281/zenodo.4331404

A.2 Description

We have split this artifact into two components:

(1) **Diospyros compiler** This is our publicly available compiler that produces C/C++ code with intrinsics. This component can be run on the provided VirtualBox virtual machine, or installed from source and run locally on the reviewer's machine.

(2) **Evaluation on licensed instruction set simulator (ISS)** Our compiler targets the Tensilica Fusion G3, which does not have an publicly accessible compiler or ISS (the vendor provides free academic licenses, but the process is not automated). To reproduce the cycle-level simulation statistics from our paper, we have provided reviews limited access to our research server (with permission from the AEC chairs).

How to access. We have made our artifact in two formats:

- In the form a virtual image that comes with all the dependencies pre-installed.
- In the form of an open-source code repository host on GitHub:

The instructions to download our virtual image or install from source can be found here:

github.com/cucapra/diospyros/blob/asplosaec/evaluation/README.md

A.3 Installation

If you use the provided VirtualBox virtual machine, it has all dependencies installed. To optionally run locally, follow the instructions for installing prerequisites from the Diospyros repository:

github.com/cucapra/diospyros

A.4 Experiment Workflow

This artifact is intended to reproduce the 4 main experimental results from the paper:

(1) **Compiling benchmarks (Table 1; Figure 5)** Compilation and simulated cycle-level performance of 21 kernels (across 4 distinct functions). We compare kernels compiled by Diospyros with kernels compiled with the vendor's optimizing compiler and optimized library functions.
(2) **Translation validation (Section 3.2)** Translation validation for all 21 kernels that the scalar specification and vectorized result (both in our abstract vector domain specific language) are equivalent.
(3) **Timeout ablation study (Figure 6)** Ablation study on a single kernel (10×10 by 10×10 MatMul) over a range of equality saturation timeouts.
(4) **Application case study (Section 5.7)** Speedup of an open source computer vision application (Theia Structure From Motion library) with a single kernel compiled by Diospyros (QR decomposition).

We provide scripts in Python to automate each of these 4 results; and provide instructions for (1) running components locally or on the provided VM, then (2) copying the new data to our research server to finish the evaluation with the licensed instruction set simulator.

A.5 Evaluation and Expected Results

This artifact aims to let other researchers:

(1) Reproduce the statistics and charts in our paper (Table 1, Figure 5, Figure 6).
(2) More easily reuse our techniques and implementation.

A.6 Notes

Because equality saturation times out after 3 minutes for some large kernels; the results may differ slightly from the paper. However, the actual execution of generated code on the instruction set simulator is deterministic.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.
[2] Michail Alvanos and Pedro Trancoso. 2016. Video SIMDBench: Benchmarking the compiler vectorization for multimedia applications. In *2016 Euromicro Conference on Digital System Design (DSD)*. IEEE, 168–175.
[3] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
[4] Dan Benanav, Deepak Kapur, and Paliath Narendran. 1987. Complexity of matching problems. *Journal of Symbolic Computation* 3, 1 (1987), 203–216.
[5] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.
[6] Cadence Design Systems, Inc. 2020. Tensilica Customizable Cores. https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.
[7] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*.
[8] Stijn Eyerman and Lieven Eeckhout. 2010. Modeling critical sections in Amdahl's law and its implications for multicore design. In *International Symposium on Computer Architecture (ISCA)*. 362–370.
[9] Franz Franchetti and Markus Püschel. 2008. Generating SIMD Vectorized Permutations. In *International Conference on Compiler Construction (CC)*.
[10] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2006. A rewriting system for the vectorization of signal transforms. In *International Conference on High Performance Computing for Computational Science (VECPAR)*.
[11] Ricardo E Gonzalez. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro* 20, 2 (2000), 60–70.
[12] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.
[13] Rajeev Joshi, Greg Nelson, and Keith H. Randall. 2002. Denali: A Goal-directed Superoptimizer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[14] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[15] Nikolaos Kyrtatas, Daniele G. Spampinato, and Markus Püschel. 2015. A Basic Linear Algebra Compiler for Embedded Processors. In *Design, Automation & Test in Europe (DATE)*.
[16] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[17] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. 2013. Exploiting Vector Instructions with Generalized Stream Fusion. In *ACM International Conference on Functional Programming (ICFP)*.
[18] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *Proceedings of the International Conference on Supercomputing*.
[19] Charith Mendis and Saman Amarasinghe. 2018. GoSLP: Globally Optimized Superword Level Parallelism Framework. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.
[20] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. 2015. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics* 31, 5 (2015), 1147–1163.
[21] Raul Mur-Artal and Juan D Tardós. 2017. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. *IEEE Transactions on Robotics* 33, 5 (2017), 1255–1262.
[22] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[23] Greg Nelson. 1980. *Techniques for program verification*. Ph.D. Dissertation. Stanford University.
[24] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-Vectorization of Interleaved Data for SIMD. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[25] JoAnn M Paul and Brett H Meyer. 2007. Amdahl's law revisited for single chip systems. *International Journal of Parallel Programming* 35, 2 (2007), 101–123.
[26] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the Enemy of Good: Best-Effort Program Synthesis. In *European Conference on Object-Oriented Programming (ECOOP)*.
[27] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement

Synthesis for GPU Kernels. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[28] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[30] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[31] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program Generation for Small-Scale Linear Algebra Applications. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*.

[32] Hauke Strasdat. 2015. Sophus Project Website. https://strasdat.github.io/Sophus/.

[33] Hauke Strasdat, Andrew J Davison, JM Martìnez Montiel, and Kurt Konolige. 2011. Double window optimisation for constant time visual SLAM. In *IEEE International Conference on Computer Vision (ICCV)*. 2352–2359.

[34] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. 2019. OpenVSLAM: A Versatile Visual SLAM Framework. In *Proceedings of the 27th ACM International Conference on Multimedia* (Nice, France) *(MM '19)*. ACM, New York, NY, USA, 2292–2295. https://doi.org/10.1145/3343031.3350539

[35] Chris Sweeney. 2016. Theia Multiview Geometry Library: Tutorial & Reference. http://theia-sfm.org.

[36] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[37] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[38] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2020. A Synthesis-Aided Compiler for DSP Architectures (WiP Paper). In *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[39] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. 2017. Extending Halide to Improve Software Development for Imaging DSPs. In *ACM Transactions on Architecture and Code Optimization (TACO)*.

[40] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

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

[42] Leonid Yavits, Amir Morad, and Ran Ginosar. 2014. The effect of communication and synchronization on Amdahl's law in multicore systems. 40, 1 (2014), 1–16.

[43] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. 2003. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 63–76.