A Synthesis-aided Compiler for DSP Architectures (WIP Paper)

Alexa VanHattum*
Cornell University
USA

Rachit Nigam*
Cornell University
USA

Vincent T. Lee
Facebook
USA

James Bornholt
University of Texas at Austin
USA

Adrian Sampson
Cornell University
USA

Abstract

Digital signal processors (DSPs) offer cutting-edge energy efficiency for embedded real time multimedia computations, but effectively targeting them necessitates programmability trade-offs. To get the best performance from a modern DSP, programmers need to work at a low level of abstraction—manually tailoring vendor-specific instructions to enable vector and VLIW computation. Diospyros is a synthesizing compiler that searches for optimal data layouts to enable efficient vectorized code on DSPs. Preliminary results show that for small fixed-size matrix multiply and 2D convolution, Diospyros achieves a 6.9–7.5× speedup compared to vendor-provided optimized kernels, and a 6.4–28.6× speedup over baseline loop-based kernels at an `-O3` optimization level with the vendor's included compiler.

1 Introduction

Compute-heavy multimedia applications—from embedded vision for augmented reality to 5G networking—have driven interest in DSPs as accelerators in embedded systems on chip (SoCs). DSPs optimize for extreme power efficiency while maximizing parallelism, particularly for real-time vision tasks such as object recognition.

However, this efficiency comes at a programmability cost. DSPs achieve their unique performance characteristics via in-order cores with exotic VLIW and vector instruction sets. Even state-of-the-art compiler techniques for vectorization underperform expert-written code. For peak performance, programmers must manually tailor instructions via intrinsic function calls to control vector registers and data layouts.

```
for (int oRow = 0; oRow < outRows; oRow++) {
  for (int oCol = 0; oCol < outCols; oCol++) {
    for (int fRow = 0; fRow < fRows; fRow++) {
      for (int fCol = 0; fCol < fCols; fCol++) {
        // ...
        o[oRow][oCol] += in[iRow][iCol] * f[fRowT][fColT];
```

(a) Portable C implementation. Simple index computations for `iRow`, `iCol`, `fRowT`, and `fColT` are omitted.

```
VEC_LOAD(I_0_4, I, 0, 16);
// ...
gathered_I = VEC_SELECT(I_0_4, Z, {1, 1, 2, 6});
gathered_F = VEC_SHUFFLE(F_0_4, {2, 3, 2, 2});
VEC_MULTIPLY_ACCUMULATE(O_0_4, gathered_I, gathered_F);
// ...
VEC_STORE(O, O_0_4, 0, 16);
```

(b) Tuned C with intrinsics for the Fusion G3.

Figure 1. Two implementations of a 2D convolution kernel. For fixed matrix sizes, the tuned code (b) is 28.6× faster than the baseline (a) on a specific DSP.

Figure 1 shows that target-specific tuned code outperforms a straightforward implementation by an order of magnitude. Figure 1a shows a standard implementation of a 2D convolution operator, and Figure 1b previews an intrinsic-laden fast implementation for the Tensilica Fusion G3 DSP [6]. The tuned implementation manually loads data into vector registers (named `I_0_4` and similar), shuffles the data using calls to the `VEC_SELECT`/`VEC_SHUFFLE` intrinsics, and uses the `VEC_MULTIPLY_ACCUMULATE` to perform a target-optimized MAC instruction on the shuffled data. The resulting implementation outperforms the naive loop nest by 28.6× (Section 4.2).

The key to the optimized code's performance is its *custom data layout*. The Fusion G3 has 4-wide SIMD vector instructions, and the code needs to maximize the utilization of these execution resources. It works by irregularly packing parts of the matrix and filter operands into vector registers to set up for 4 simultaneous multiply–accumulates. The `VEC_SELECT` intrinsic uses a vector of indices into the input registers—here, the "magic numbers" `{1,1,2,6}`.

*These authors contributed equally.

LCTES'20, June 16, 2020, London, United Kingdom

ACM ISBN ISBN 978-1-4503-7094-3/20/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/XXXXXX.XXXXXX

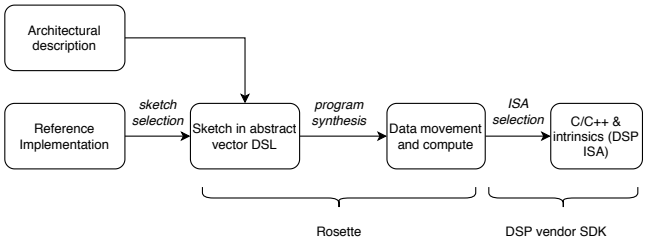


Figure 2. Proposed system architecture.

Finding these advantageous vector packing schemes—and implementing them using target-specific intrinsics—is a challenging task that requires both insight into the computation and deep knowledge of the hardware. We present ongoing work on Diospyros, a compiler that automates the search for high-performance DSP kernels using *program synthesis*. Diospyros automatically discovers vector packing schemes for a given computation and hardware target and generates implementations in intrinsic-based C.

Diospyros will enable programmers to use portable, high level implementations while providing nearly the performance of expert-tuned optimized kernels. Diospyros targets small, fixed-size matrix kernels to replace hot loops in large applications such as deep learning. Diospyros will open up new possibilities for efficient multimedia acceleration in embedded systems programming. In order to scale to larger kernels, we are working towards both more efficient synthesis encodings and decomposition of data into multiple subproblems that can be solved iteratively.

2 Related Work

Classical vectorization techniques—from loop dependency analysis [1] to modern auto-vectorization techniques [10, 11]—produce fairly regular vectorized iteration code because they do not attempt to aggressively reorder and reuse data into irregular patterns. Existing techniques prioritize efficient compilation over optimality: they are designed to run on millions of lines of code but find mostly specific types of vectorization. Our more general vectorization problem formulation gives an automated compiler the ability to aggressively reorder data.

Previous work has used the Halide image processing language [13] to target DSPs, but has not supported exploration of a large search space of irregular data movement strategies [15]. Other approaches can generate target-specific shuffles to implement known permutations, but do not find the permutation strategies themselves [8, 9]. While our search strategy comes at the cost of compilation time, Diospyros aims to automate more of the process of producing code that is competitive with hand-tuned implementations.

3 The Diospyros Synthesis Workflow

Diospyros synthesizes efficient DSP kernels using the workflow in Figure 2. Diospyros requires an unoptimized reference implementation of a kernel and a *sketch*—a syntactic template for the target program. Diospyros then uses an off-the-shelf program synthesizer [14] to fill in the template with an efficient implementation. Finally, it lowers the completed template from the DSL into concrete DSP instructions.

3.1 Vector DSL

Diospyros optimizes DSP efficiency by searching for an optimal data layout and associated movement. DSP architectures are typically in-order cores with wide vector instructions. To extract high performance, hand-tuned DSP kernels use intricate data layouts and movements to pack data into vector registers and reach maximum utilization of the architecture's resources for a fixed input size, output size, and operation. This hand tuning is necessarily specialized to the particular DSP architecture and application, and so is rarely portable.

Diospyros focuses on movement by defining a *platform-agnostic* domain-specific language (DSL) for vector operations. The DSL is low level and includes common vectorized operations such as memory loads and stores, arithmetic, and data shuffles. However, it abstracts away concrete details of the DSP architecture, deferring them to a later architecture-specific instruction selection phase (Figure 2). For example, Diospyros's DSL includes a vector shuffle operation:

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

that takes as input an array of `indices` defining where to move each element of `inputs`. The DSL does not restrict the possible values of `indices`, offering the flexibility to discover optimal data layouts and movements. Lowering this instruction to the target DSP architecture requires selecting an instruction sequence that achieves this desired movement from among the architecture's available shuffle operations.

3.2 Sketch Selection and Synthesis

Diospyros uses program synthesis to search for a more efficient implementation of a reference kernel. Program synthesis is the task of automatically generating a program that satisfies a specification; here, the specification is equivalence to the reference program.

To make this search tractable, Diospyros applies *syntax-guided synthesis* [2], which uses a syntactic template called a *sketch* to direct the search. The sketch is a program in the target DSL, but with some missing expressions called *holes* for the synthesizer to fill in. For example, a sketch could use the `vec-shuffle` instruction above, but with a hole in place of the `indices` argument. The synthesizer would then search for `indices` (i.e., where to move the data) that makes the completed sketch equivalent to the reference implementation. Diospyros uses the Rosette solver-aided language [14], which extends Racket with synthesis support.

Given a sketch, Rosette fills the holes by using an off-the-shelf satisfiability modulo theories (SMT) solver [4] to search for completions that satisfy the reference implementation.

Constructing a good sketch is critical to the scalability of any syntax-guided synthesis tool. If the sketch has too many holes, the search space is too large to explore in reasonable time; if it has too few holes, it does not offer enough flexibility to generate novel solutions. Diospyros sketches focus on data layout and movement, while avoiding the need for the synthesizer to rediscover implementation details such as moving data into the final output registers. A Diospyros sketch comprises a fixed function prologue and epilogue, with holes for the synthesizer to generate the instructions in between. For example, the loop body of a sketch for 2D convolution might comprise two vectorized shuffle instructions with unknown indices, followed by a multiply–accumulate that outputs to an unknown register:

```
(vec-shuffle 'reg-I (??indices) (list 'I 'Z))
(vec-shuffle 'reg-F (??indices) (list 'F 'Z))
(vec-mac (??reg) 'reg-I 'reg-F)
```

Here, the `??indices` and `??reg` functions create holes that can be filled by indices and a register, respectively.

While a sketch defines the search space for a synthesizer, it does not guarantee efficiency—any program equivalent to the reference is a valid solution. To guide the search towards efficient solutions, Diospyros augments the synthesis process with a *cost function* [3]. The cost function assigns a cost to each instruction in the vector DSL based on its expense. The synthesizer searches for a solution that both matches the reference specification and minimizes the cost function. Diospyros's current cost function assigns costs to shuffles proportional to how many vector registers must be gathered to implement the data movement.

Good sketches are necessarily specific to an application, because they capture problem-specific high-level insights. In the 2D convolution example, the sketch captures the essence of a convolution—data movement followed by multiply–accumulates—without fixing the details of how data is packed or moved. These sketches therefore still require domain expertise. However, Diospyros maximizes the reuse of a sketch using *sketch generation.* A Diospyros sketch is parametric in the size of its inputs, so while a synthesized Diospyros kernel is specific to a desired input size, a programmer needing a new combination can simply re-run synthesis on the same sketch using the new parameters. This flexibility avoids the need for expert hand-tuning for every new DSP application.

3.3 Instruction Selection

Because Diospyros's DSL is not architecture-specific, a program synthesized in it is not immediately executable. The final phase of Diospyros's compilation is to perform instruction selection to lower the abstract program onto a concrete architecture. This lowering phase translates abstract vector operations into C++ compiler intrinsics that can then then compiled using the DSP's standard SDK.

Currently, Diospyros targets Tensilica's Fusion G3 DSP processor [6]. Diospyros's unrestricted shuffles, for example, are lowered to target Tensilica's `VEC_SHUFFLE` (single-register) and `VEC_SELECT` (two-register) intrinsic functions. Diospyros uses nested `VEC_SELECT` instructions to implement arbitrary shuffles with more than two registers—an inefficiency that is reflected in the target-specific cost model.

4 Preliminary Evaluation

Our preliminary evaluation addresses these questions:

- **Q1** Can Diospyros compete with hand-tuned implementations for fixed-sized matrices, outperforming loop-analysis-based and manual vectorization?
- **Q2** What are Diospyros's current scalability limitations?

4.1 Methodology

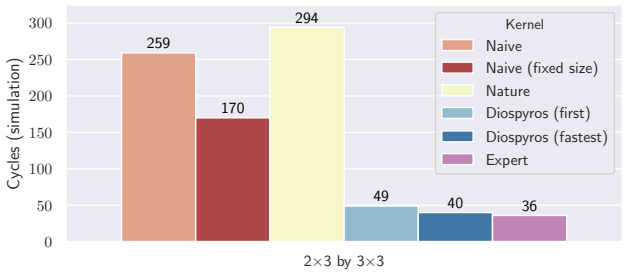
We report cycle counts from Tensilica's cycle-level simulator for the Fusion G3 DSP processor [6]. All implementations (baseline loop-based functions, library-provided functions, and Diospyros-generated functions) are compiled from C source code with Tensilica's provided compiler at the highest optimization level, `-O3`. We run experiments on a machine with two Intel Xeon E5-2620v4 CPUs running CentOS 7.6.

We compare Diospyros's performance with the Nature DSP library included with Tensilica processors. Nature provides well-optimized kernels that perform better than naive C++; however, their performance is limited by the need to be generic over matrix sizes.

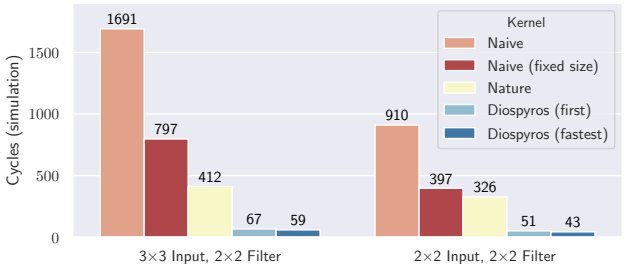
4.2 Performance Results

We demonstrate the potential of Diospyros on two ubiquitous floating point matrix computations: matrix multiply and 2D convolution (described in detail in Section 4.3). We compare Diospyros kernels with baseline loop-based implementations (with both parametric sizes and inlined fixed sizes to facilitate more aggressive standard `-O3` optimizations), the Nature DSP library functions, and in the case of matrix multiply, expert-written code hand-tuned for the fixed size. We show both the first solution found by Diospyros (without minimizing the cost model), and the best solution found within a 30 minute timeout (the lowest cycle count).

To answer **Q1**, both kernels outperform the baselines at small matrix sizes. The Diospyros-generated matrix multiply is 6.4$times$ faster than the naive loop nest, with performance within 12% of the expert kernel (Figure 3a). Figure 3a also demonstrates that highly-optimized code such as Nature can perform poorly on small kernels due to the control overhead of the parametrized unrolling. The Diospyros-generated 2D convolutions are 21.1–28.6× faster than the naive baseline and 7.5–6.9× times faster than the Nature kernel (Figure 3b).



(a) Matrix multiplication.



(b) 2D convolution.

Figure 3. Simulated running time in cycles for two benchmarks. *Naive* is a naive loop nest, *Naive (fixed size)* is a loop nest with fixed bounds, *Nature* is a vendor-supplied library function, *Diospyros (first)* is the first solution found by our system, *Diospyros (fastest)* is the best solution found by our system, and *Expert* is hand-tuned vectorized code (where available).

However, our evaluation shows Diospyros's initial scalability limitations (**Q2**). Diospyros finds initial implementations of the matrix multiply after 12 seconds and the 3×3 and 2×2 convolution in 5 minutes. However, Diospyros's current implementation does not find a solution to a 4×4 and 2×2 convolution within 10 hours. We address ongoing work in Section 5.1.

4.3 Case Study: 2D Convolution

Convolution is ubiquitous computational primitive in computer vision. The kernel takes an input matrix and a filter matrix, and produces an output matrix where each element is the weighted sum of the input and a element-wise filter multiplication. A naive implementation of a convolution uses a deep loop nest, iterating over both the output matrix's and the filter's rows and columns (Figure 1a).

The Nature library's 2D convolution makes extensive use of vector intrinsics for loads, stores, and arithmetic operations. However, because the sizes of the matrices are dynamically provided, the implementation is limited to using general vectorization techniques such as loop unrolling. The implementation must include outer loops over the unrolling factor, with additional loops for boundary conditions when the unrolling factor does not evenly divide the matrix sizes.

In contrast, Diospyros is able to exploit the fixed matrix sizes to enable aggressive data movement to keep vector lanes full even in cases where the register width does not evenly divide the loop boundaries. In fact, the Diospyros-generated code contains no control flow at all because it is able to fully unroll the deep loop nest into straightline code.

Given a sketch and concrete fixed data sizes of 3×3 for the input, 2×2 for the filter, and 4×4 for the output, and vector width 4, Diospyros generates a solution that is 28.6× faster than the baseline at an `-O3` optimization level (Section 4.2). Figure 1b demonstrates a section of the C/C++ code with intrinsics generated by Diospyros. Similar snippets are repeated with different shuffles and partitions 15 times, for a total of 61 lines of completely straight-line C/C++ code (excluding variable declarations).

While an expert might find an even faster solution with intrinsics, doing so would require manually determining data shuffles to enable full vector lanes—there is no obvious way to vectorize this kernel when the vector width does not evenly divide the matrix sizes. Diospyros thus enables near-expert level vectorized performance, without manually specifying irregular data movement.

5 Future Directions

Diospyros generates competitive code for small numerical kernels on a Tensilica DSP. We have more work to do, however, to make the system scale to more complex programs, to make it easier to use, and to target other DSP hardware.

5.1 Scalability

A fundamental challenge in the development of synthesis-aided compilers is scaling them up to larger programs [12]. We see three avenues for improving scalability: decomposing compilation into smaller synthesis problems, improving solving time during synthesis, and parallelizing the search.

To scale to large benchmarks, we want to partition the target program into smaller problems that can be solved independently. For example, many DSP kernels have a (nested) loop structure, and experts extract performance by hand-unrolling the loop. Rather than rediscovering this unrolled structure during synthesis, we could instead perform the synthesis in two steps: first, synthesize a single iteration of the loop body, then synthesize an unrolled version that maximizes reuse but does not otherwise alter data movement. This approach may be suboptimal because it does not explore the entire search space, but will be necessary to solve larger problems. Decomposition also enables parallelism, which synthesizers are otherwise lackluster at exploiting.

5.2 Programmability

To use the current version of Diospyros, programmers need to specify both a reference implementation and a program sketch in our DSL. Constructing a sketch requires significant domain-specific knowledge and effort, although that effort can be reused across many instantiations of the same problem (e.g., across different matrix sizes).

To ease adoption of Diospyros, we are exploring symbolic *lifting* to automatically extract specifications and sketches from existing C/C++ programs. We plan to leverage existing binary analysis tools [5] and provide programmers with a pragma annotation to automatically extract an annotated program region for synthesis.

5.3 Target Independence

Diospyros aims to generate code for multiple target architectures using a portable, high-level specification. Targeting multiple architectures requires us to build an intermediate representation (IR) that can abstractly reason about target specific constraints. For example, the Hexagon DSP [7] exposes only hard-coded data reorganization primitives while Tensilica cores allow for arbitrary shuffles. Furthermore, different architectures might have different cost models for different instructions. Currently, Diospyros exposes a flexible `vec-shuffle` primitive that can arbitrarily move data. We plan to extend Diospyros to target multiple architectures and build a robust IR.

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] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*.

[3] 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)*.

[4] Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS*.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[6] Cadence Design Systems, Inc. [n.d.]. Tensilica Customizable Cores.

[7] Lucian Codrescu. 2015. Architecture of the Hexagon™ 680 DSP for mobile imaging and computer vision. In *2015 IEEE Hot Chips 27 Symposium (HCS)*.

[8] Franz Franchetti and Markus Püschel. 2008. Generating SIMD Vectorized Permutations. In *Proceedings of the International Conference on Compiler Construction*.

[9] 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*.

[10] 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)*.

[11] 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)*.

[12] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[13] 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)*.

[14] 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)*.

[15] 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)*.