INVITED PAPER

SPIRAL: Extreme Performance Portability

This paper provides an end-to-end discussion of the SPIRAL system, its domain-specific languages, and code generation techniques.

By FRANZ FRANCHETTI, *Senior Member IEEE*, TZE MENG LOW, *Member IEEE*, DORU THOM POPOVICI, *Student Member IEEE*, RICHARD M. VERAS, *Member IEEE*, DANIELE G. SPAMPINATO, *Member IEEE*, JEREMY R. JOHNSON, *Senior Member IEEE*, MARKUS PÜSCHEL, *Senior Member IEEE*, JAMES C. HOE, *Fellow IEEE*, AND JOSÉ M. F. MOURA, *Fellow IEEE*

ABSTRACT | In this paper, we address the question of how to automatically map computational kernels to highly efficient code for a wide range of computing platforms and establish the correctness of the synthesized code. More specifically, we focus on two fundamental problems that software developers are faced with: performance portability across the ever-changing landscape of parallel platforms and correctness guarantees for sophisticated floating-point code. The problem is approached as follows: We develop a formal framework to capture computational algorithms, computing platforms, and program transformations of interest, using a unifying mathematical formalism we call operator language (OL). Then we cast the problem of synthesizing highly optimized computational kernels for a given machine as a strongly constrained optimization problem that is solved by search and a multistage rewriting system. Since all rewrite steps are semantics preserving, our approach establishes equivalence between the kernel specification and the synthesized program. This approach is implemented in the SPIRAL system, and we demonstrate it with a selection of computational kernels from the signal and image processing domain, software-defined radio, and robotic vehicle control. Our target platforms range from mobile devices, desktops, and server multicore processors to large-scale high-performance and supercomputing systems, and we demonstrate performance comparable to expertly hand-tuned code across kernels and platforms.

KEYWORDS | Automatic performance tuning; performance engineering; program generation; program synthesis; SPIRAL

Manuscript received May 24, 2018; revised September 18, 2018; accepted September 19, 2018. Date of current version October 25, 2018. This material is based on research sponsored by DARPA under Agreements FA8750-12-2-0291, FA8750-16-2-0033, HR0011- 13-2-0007, DOI Grant NBCH1050009, and ARO Grant W911NF0710416. This work was supported by the National Science Foundation (NSF) under Awards 0325687, 0702386, and 0931987; by the U.S. Office of Naval Research (ONR) under Grant N000141110112; and by Intel, Nvidia, and Mercury. This work was supported in part by the Department of Defense under Contract FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM17-0413]. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. *(Corresponding author: Franz Franchetti.)*

F. Franchetti, **T. M. Low**, **D. T. Popovici**, **D. G. Spampinato**, **J. C. Hoe**, and **J. M. F. Moura** are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: franzf@ece.cmu.edu).

R. M. Veras is with the Center for Computation and Technology, Louisiana State University, Baton Rouge, LA 70803 USA.

J. R. Johnson is with the College of Computer and Informatics, Drexel University, Philadelphia, PA 19104-2875 USA.

M. Püschel is with the Department of Computer Science, ETH Zürich, 8092 Zürich, Switzerland.

Digital Object Identifier 10.1109/JPROC.2018.2873289

I. INTRODUCTION

Computer architects are experimenting with ever more complex systems containing manycore processors, graphics processors, field-programmable gate arrays (FPGAs), and a range of speculation techniques to keep Moore's law on track and to keep systems within their power envelope. This enormous growth of computing power is a boon to scientists; however, it comes at a high cost: the development of efficient and correct computing applications has become increasingly more difficult and complex. Already on a single central processing unit (CPU), the performance of an inefficient implementation can be 10–100 times slower than the fastest code written by an expert. Thus, significant effort has to be invested by highly sophisticated programmers to attain the desired performance on modern platforms that include multiple CPUs or accelerators.

0018-9219 © 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

At the heart of this effort is the inherent tension between performance, software abstraction, and code maintainability as programmers have to constantly develop for the latest release of the newest platform.

Current tools such as advanced compiler frameworks and automatic performance tuning (autotuning) systems allow portability across a wide range of platforms and algorithms while also attaining reasonable performance. However, automatically achieving near-optimal performance for performance-relevant mathematical operations has been shown to be difficult. Part of the reason for this difficulty is that programs do not exactly capture the desired input–output behavior. They are often over-specified either due to language requirements or programmer decisions while writing code. As such, the expert programmer often has to hand-code their highly optimized implementation with low-level machine-specific instructions (assembly) in order to attain appropriate high performance when high-performance library code is not available.

In this paper, we present a complete overview of the SPIRAL system. SPIRAL is a program and library generation/synthesis and autotuning system that translates rule-encoded high-level specifications of mathematical algorithms into highly optimized/library-grade implementations for a large set of computational kernels and platforms. The system has been developed over the last 20 years and is freely available as open source under a BSD-style license. SPIRAL formalizes a selection of computational kernels from the signal and image processing domain, software-defined radio (SDR), numerical solution of partial differential equations, graph algorithms, and robotic vehicle control, among others. SPIRAL targets platforms spanning from mobile devices, to desktop and server multicore processors, and to large high-performance and supercomputing systems, and it has demonstrated performance comparable to expertly hand-tuned code across a variety of kernels and diversity of platforms. To maximize portability and to leverage the work of backend compiler developers, when producing software SPIRAL usually targets vendor compilers such as the Intel C compiler or IBM's XL C compiler, or widely available compilers such as the GNU C compiler or LLVM, and only rarely generates assembly code directly.

A. Contributions

This paper presents an end-to-end description of the current status of the SPIRAL system and its underlying program generation methodology. We detail:

- the formal framework to describe computational kernels;
- the machine model used to target both known and novel, yet-unknown machines;
- the constraint solving system that derives a search space of candidate programs for a given platform and computational kernel;
- the autotuning approach that yields high performance;
- correctness guarantees and applied verification methods;
- results across a range of platforms for a range of computational kernels.

High-level algorithm representation of linear signal transforms using the signal processing language (SPL) and rule trees, aspects of SIMD vectorization for SSE and SMP parallelization using OpenMP, search and autotuning, as well as performance modeling are discussed in the previous SPIRAL overview paper [1], but will be discussed as necessary to make this paper self-contained.

B. Synopsis

Section II describes the high level and conceptual aspects of SPIRAL. Section III introduces hardware, algorithm, and program transformation abstractions in a unified framework. Section IV discusses how term rewriting, constraint solving, and domain-specific language (DSL) compilers interact to synthesize optimized software implementations and hardware designs, as well as co-optimize combined hardware/software systems. Section V demonstrates with select results the quality of SPIRAL generated software and hardware. Section VI describes current work aiming at extending its capabilities and how to obtain the newly released open-source version of SPIRAL. Finally, Section VII offers a summary and conclusion.

II. OVERVIEW

The key observation in the SPIRAL system is that the mathematics underlying computational kernels changes slowly and provides a well-developed language to describe algorithms, while target computer platforms change frequently. Consider some of the basic mathematical and physical concepts that underlie commonly used computational kernels: geometry (Euclid, 300 BC [2]), Gaussian Elimination (unknown Chinese scholars, 179 AD [3]), equations of motion (Newton, 1687 [4]), and the fast Fourier transform (FFT, Gauss, 1805 [5]). Further, consider how FFTs have evolved since Gauss: rediscovered in 1965 by Cooley and Tukey [6], further FFT variants in subsequent years, formalized in matrix form extended and popularized in 1992 by Van Loan [7].

In contrast, consider the release timeline of classes of processors that have been targeted by SPIRAL: single core x86 CPU with cache in personal computers (mid-1990s), multicore CPU (Pentium D, 2005), GPGPU (GeForce 8800, 2006), manycore CPU (Xeon Phi, 2011). Even if we were to restrict the discussion to Intel CPUs, the rate at which new versions and microarchitectures are released is staggering [8]. Moreover, the range of platforms successfully targeted by SPIRAL spans orders of magnitude in peak performance: mobile and embedded devices (ARM CPUs and multicores), desktop and server class CPUs (up to tens of cores and gigabytes to terabytes of shared memory), accelerators (graphics processors, Xeon PHI, FPGAs), and large parallel machines such as BlueGene/L/P/Q and the K computer with up to almost 800,000 cores.

A. Goal and Approach

1) Goal: The goal of SPIRAL is to provide performance portability for well-defined, ubiquitously needed computational kernels across a wide range of continuously changing computational devices. Specifically, SPIRAL aims to automatically generate an implementation that satisfies the functional specification of a given problem on a given platform. The implementation should rival the performance that the best human expert programmer can achieve. Further, SPIRAL aims to provide evidence of correctness for this generated implementation. This problem statement addresses questions of programmability, performance portability, and rapid prototyping. SPIRAL builds on Johnson *et al.*, methodology that connected the mathematics of fast Fourier transforms to programs and computer architectures [9]–[11].

2) Approach: The SPIRAL solution is as follows: 1) develop a formal framework to capture computational algorithms, computing platforms, and program transformations of interest through a unifying mathematical formalism we call *operator language* (OL); and 2) cast the problem of synthesizing highly optimized computational kernels for a given machine as a tightly constrained optimization problem that is solved by a multistage rewriting system that uses semantics-preserving operations. This approach allows us to formally prove the equivalence between the kernel specification and the synthesized program.

The formal system has three main components.

- Algorithms such as the famous Cooley–Tukey FFT algorithm are captured in OL, which encompasses a family of DSLs that capture various aspects and refinements of specifications and algorithms for computational kernels. The top level DSL (called tagged OL, tOL) captures the input–output behavior of kernels (i.e., their semantics). A midlevel DSL (called operator language, OL, which extends SPL) captures the computation as data flow graph. At an even lower level, a DSL called Σ-OL (pronounced Sigma-OL) captures folded data flow graphs and can be interpreted as providing loop abstractions. Finally, a DSL called `icode` (an abstract internal code representation) captures a small subset of C in the form of abstract syntax trees (ASTs) and can be given pure functional semantics for a relevant subset of programs.
- Hardware such as a multicore CPU, FPGA, or GPU is modeled through OL expressions that can be implemented efficiently on the given hardware. The idea is that algorithms that can be composed exclusively (or mainly) from these expressions can be mapped efficiently to the associated hardware.
- Program transformations such as loop tiling, fission, and fusion are captured as rewriting rules coupled with a tagging mechanism. Rewriting rules transform OL/Σ-OL expressions into more efficient expressions that can be composed from the hardware-based OL expressions. The tagging mechanism facilitates the introduction of higher level program transformations such as parallelism.

These three components of the system are used to construct a space of suitable programs for a given OL specification that can be mapped efficiently to SPIRAL's machine model of a given hardware instance. Conceptually, this is done by intersecting a space of programs that run efficiently on the given hardware with a space of algorithms that implement the desired computational specification, subject to applicable program transformations. The result is a search space in which every point is an algorithm for the given specification that runs efficiently on the given hardware. Finally, an autotuning system traverses this search space and uses a DSL compiler to translate the points in the search space into high-performance implementations.

This approach of mapping a specification to highly optimized implementations across a wide range of machines solves the forward problem: mapping computations to architectures. The inverse problem of finding the best architecture for a given specification, i.e., algorithm/hardware co-optimization can be solved by iteratively solving multiple forward problems while traversing the architecture space.

3) Success: The SPIRAL approach and associated systems have been developed over the last two decades. The initial focus was on linear transform algorithms including the FFT. The first basic approach of mapping a mathematical DSL to efficient C or Fortran code using a search mechanism for optimization was presented in [12]–[14]. The approach was then expanded to include a larger set of linear transforms and range of computing platforms [1], [15], thus offering a program generation solution for what is later identified as the spectral dwarf in Berkeley's 7 dwarfs/11 motifs classification [16]. The focus of this paper is the work of the last decade in which we expanded the SPIRAL approach to a much larger scope of computational kernels, while earlier work before 2005 is discussed in [1].

Using the systematic rewriting approach, SPIRAL has demonstrated, over the last two decades, the automatic generation of expert-level performance code across a wide range of microarchitectures. Specifically, SPIRAL has successfully targeted modern CPUs with multilevel caches [1], [12], [13], [17], [18], multiple cores [19], [20], SIMD vector instructions [21]–[29], and multisocket systems with large main memory [30], fixed-point arithmetic [31], fused multiply–add instructions [32], modulo arithmetic [33]), GPUs [34]–[36], DSPs [37], the Cell BE [34], [38], Larrabee and Xeon PHI [39], FPGAs [39]–[45], clusters [47], up to 128k cores on BlueGene/L/P/Q [47]–[49], the K computer, and in presilicon settings (IBM Cell BE [38], BlueGene/L, Intel AVX and Xeon Phi [39]).

The original focus of SPIRAL was linear transforms [1], [15] such as the discrete Fourier transform [20], [51], linear filters [52], and the discrete wavelet transform [53], and is described in detail in [1]. Since then, the approach and the associated DSLs have been expanded to a range of kernels [54], [55] including the image formation algorithm in SAR [56], components of JPEG 2000 [57], Viterbi decoders [58], SDR [58], [59], matrix multiplication [54], and quantum chemistry kernels [61]. An entire generator devoted to small-scale linear algebra applications was built in [28], [62], and [63]. Support for some multigrid applications [64] and Poisson [61] solvers was also introduced, and we synthesized sensor fusion and control code kernels [65].

Rewriting in SPIRAL handles the basic mapping to a given target platform, but leaves a space of candidate alternatives for further tuning. To navigate this space, SPIRAL uses various search methods [1], models [66], but also machine learning approaches [65]–[68].

SPIRAL can be used as a low-level backend code generation tool and kernel generator for polyhedral compiler infrastructures [71], [72] and as a programming tool for special purpose hardware [73]. The formal framework of SPIRAL lends itself to mathematical correctness arguments [65], [74]. SPIRAL was used to produce parts of Intel's MKL and IPP libraries (it generated 1 million lines of code for IPP's `IPPgen` module) [75], [76], codelets for the BlueGene/L/P/Q fastest Fourier transform in the west (FFTW) version [48], [77], and Mercury's Scientific Algorithms Library (SAL) [78]. A more principled design and implementation of SPIRAL using modern language features including pattern matching, staging, and embedded DSLs was studied in [79] and [80] and provides a SPIRAL prototype implemented in Scala. An experimental Haskell version was also developed in this context [81]. Current work includes extending SPIRAL to support graphs and sparse matrices algorithms, as well as proving the correctness of SPIRAL's program generation toolchain within the Coq system.

4) Limitations: Generating code that is competitive with the best human-written code across a wide range of platforms is inherently a difficult task. This task is made even more complex when one requires the system to be extensible. Within SPIRAL, we simplify this task by restricting the problem domain to domains and algorithms that can be described in terms of recursive specifications. In addition, possible variations and developer choices must be extracted as free parameters. The specification needs to be encoded as a rewrite rule in a term rewriting system in a DSL with mathematical semantics, and all relevant algorithmic and parametric degrees of freedom need to be expressed. Developing such a specification can be a hard research problem, even for well-understood mathematical kernels, as evidenced by the development time line of SPIRAL's FFT capabilities [15]. As listed above, we have shown that it is possible to capture other domains beyond FFTs, but the effort in each case was considerable. Furthermore, adding a new hardware feature computational paradigm (such as vector instructions or multiple cores as they appeared), or program transformation, also requires encoding this knowledge in SPIRAL's formal system. Surprisingly, this tends to be an easier task than adding a new kernel.

While the SPIRAL developers have strived for complete coverage of the supported architecture and kernel/application space, a number of the results detailed above were demonstrated as one-off solutions. At one end of the spectrum, the FFT is most completely supported across all platforms targeted by SPIRAL. At the other end of the spectrum, some coding and communication kernels are only prototypically supported for a single platform [57].

SPIRAL's internal languages are DSLs with mathematical semantics. A mathematics-style notation lends itself to capturing algorithm and program optimization rules concisely, e.g., as done in a book. However, the mathematical notation and its implementation in the dated computer algebra system it was built on still poses a considerable hurdle to adoption. Therefore, we often wrapped up a part of SPIRAL's capabilities in easy-to-use parameterized generators (e.g., on the web at `www.spiral.net`) or as a command line tool in a compiler tool chain [71], [72]. We have also exposed part of SPIRAL's SPL/OL language as a MATLAB tensor library with high-dimensional map, reduce, reshape, and rotate operations in the style of hierarchically tiled arrays (HTAs) [82], [83].

Writing OL specifications is complicated. The specification needs to capture the exact semantics of the computational kernel without introducing superfluous information that may obscure the mathematical specification. Machine abstractions need to be structurally accurate, and standard program transformations need to be recoded as OL rewrite rules. SPIRAL's implementation in GAP (groups, algorithms, and programming, version 3) [84] adds to the complexity of writing specifications. At this writing, we have had success with web interfaces that expose only part of SPIRAL's capabilities and with an experimental MATLAB frontend that implements a subset of OL and a C library frontend that extends the FFTW interface [85]. Building SPIRAL and its DSLs within a powerful multiparadigm language such as Scala can increase maintainability and extensibility [79], [80]. Further, we are pursuing efforts to enable SPIRAL as a just-in-time (JIT) compiler. We hope that these advancements will make SPIRAL more accessible to general programmers.

B. Related Work

We now discuss the most important approaches and projects of the various technologies that are related to the SPIRAL project.

1) Performance Libraries: Mathematical performance libraries provide a uniform interface to key functionality across multiple platforms, but have been optimized

by hand for each target platform to obtain performance. Prominent examples include the Intel Math Kernel Library (MKL) [75] and Integrated Performance Primitives (IPP) [76], AMD's Core Math Library (ACML) [86], Cray's Scientific Library (SciLib) [87], IBM's Engineering and Scientific Subroutines Library (ESSL) [88], Mercury Computing's Scientific Algorithms Library (SAL), the Basic Linear Algebra Subroutines (BLAS) [87]–[90], and LAPACK [93]. The BLAS-like Library Instantiation Software (BLIS) [94] is a framework for instantiating a set of functions larger than BLAS from a set of microkernels. SPIRAL originated as a tool to help automate FFT kernels for some of these libraries, and SPIRAL-generated kernels can be found in the Intel MKL, IPP, and Mercury's SAL.

2) Compilers: Polyhedral compiler frameworks such as PetaBricks [95], CHiLL [96], R-Stream [97], PLuTo and PTile [98], Polly [99], [100], and the Polyhedral Parallel Code Generator (PPCG) [101] have their strength in regular and dense computations on standard parallel machines and accelerators, and extensions to sparse polyhedral computations have been investigated. Other approaches include annotation-based compilation [102] and the concept of telescoping languages [103].

Many HPC high-level languages follow the partitioned global address space (PGAS) paradigm to provide portability across distributed memory machines. The historically most important examples are HPF and FortranD [104], [105]. Chapel [106], X10 [107], UPC [108], and Co-Array Fortran [109] are other example of languages in this space. Systems such as PEAK [110], PetaBricks [95], Sequoia [111], CHiLL [96], the polyhedral infrastructures Pluto [98], Primetile [112], CLooG [113], as well as the GNU C interactive compilation interface (GCC ICI) and Milepost GCC [114] use autotuning and machine learning. SPIRAL captures many of the program transformation techniques used by these languages and systems, and its internal representation allows for extracting and capturing information that usually has to be derived by analysis or provided via annotations.

LLVM, Open64, and the GNU compiler suite are open compilers designed to be retargeted to a wide range of instruction set architectures (ISAs). These compilers have backends that can accept new ISAs and that can support novel instructions. SPIRAL is leveraging the intrinsics interface and vector data type abstractions provided by these open compilers as well as commercial compilers such as the Intel C++ compiler and IBM's XL C compiler to provide portable code generation across a wide range of SIMD vector extensions of CPU ISAs such as x86, POWER, and ARM.

3) Language Extensions: OpenCL, CUDA, OpenMP, and OpenACC extend C or FORTRAN with language constructs and/or pragmas to annotate the source code with information and instructions for the parallelizing/offload compiler. These language extensions require powerful high-level optimizing compilers to generate highly efficient code. SPIRAL utilizes them as backend compilers to abstract hardware details and attain better portability, but performs all necessary high-level transformations itself.

4) High-Level Synthesis: Vivado HLS and BlueSpec [115] translate higher-level language programs into hardware blocks (IP blocks) for FPGAs so that users are freed from some of the tedious work required when directly specifying hardware in Verilog or VHDL. SPIRAL directly targets Verilog and experimentally targets Vivado HLS to enable quick porting to novel FPGA architectures.

5) Code Generators and Autotuners: The autotuning community is the home of a number of influential projects that include code generators and/or autotuning systems for numerical software. Important autotuning projects include the adaptive FFT library FFTW [116], [117], the dense numerical linear algebra project ATLAS [118], [119], the sparse matrix–vector multiplication library OSKI [120], [121], and the quantum chemistry tensor contraction system TCE [122]. SPIRAL is firmly rooted in this community. General autotuning infrastructures that can be used independently of the optimization target include ActiveHarmony [123], Orio [124], and ppOpen-AT [125].

6) Generative Programming for Performance: Generative programming has gained considerable interest [126,127,128,129,130]. The basic goal is to reduce the development, maintenance, and analysis of software. Among the key tools, DSLs provide a compact representation that raises the level of abstraction for specific problems and hence enables the manipulation of programs [129]–[132]. C++, Haskell, MetaOCaml, and Scala are often used as host languages to embed DSLs [135]. The SEJITS [136] specializer specializes kernels to low-level implementations. The Delite [137] framework offers a set of optimized parallel patterns to DSLs that can be implemented on top of it. Other approaches are based on multistaging frameworks such as lightweight modular staging (LMS) [79], [137], [138] and Terra [139].

Other examples of DSL-based approaches are query compilers [140], [141] based on a stack of DSLs with progressive lowering of abstractions. Stencil code generators include [71], and Lift [142], which combines a high-level functional data parallel language with a system of rewrite rules that encodes algorithmic and hardware-specific optimization choices.

7) DSLs for HPC Libraries: The build-to-order (BTO) BLAS [143], [144] is a domain-specific compiler for matrix computations. BTO focuses on memory-bound computations (BLAS 1 and 2 operations) and relies on a compiler for vectorization. Cl1ck [145], [146] implements the formal linear algebra methods environment (FLAME) [147] methodology for automatically deriving algorithms for higher-level linear algebra functions [148] given as mathematical equations. The supported functions are mostly those covered by the LAPACK library and the generated

algorithms rely on the availability of a BLAS library. DxTer [149] transforms blocked algorithms such as those generated by Cl1ck and applies transformations and refinements to output high-performance, distributed-memory implementations. The CLAK compiler [150] finds efficient mappings of matrix equations onto building blocks from high-performance libraries such as BLAS and LAPACK.

8) DSLs for Matrix and Stencil Optimization: Another generative approach is adopted by Eigen [151], uBLAS [152], the Matrix Template Library (MTL) [153], STELLA [154], Halide [155], [156], and the Tensor Algebra Compiler (TACO) [157], among others. They use C++ expression templates to optimize the code at compile time. Optimizations include loop fusion, unrolling, and SIMD vectorization. The HTAs [82], [83], which offer data types with the ability to dynamically partition matrices and vectors, automatically handle situations of overlapping areas. HTAs' goal is to improve programmability by reducing the amount of code required to handle tiling and data distribution in parallel programs, leaving any optimization to the programmer (or program generator).

9) Frameworks and Computer Algebra Systems: Systems such as Sketch [158] and Paraglide [159] automatically synthesize software according to a specification. Rewriting systems are reviewed in [160]. Logic programming is discussed in [161]. An overview of functional programming can be found in [162]. SPIRAL does not employ SAT solvers but solves a specialized constraint programming problem through term rewriting.

Python, R, Julia, MATLAB, Java, and C++, and frameworks such as Caffe [163], Theano [164], and TensorFlow [165] are commonly used by data scientists to express graph analytics and machine learning algorithms. Computer algebra systems such as Maple [166], YACAS [167], and Mathematica [168], interactive numerical systems such as MATLAB [169], as well as interactive theorem proving systems based on higher order logic [170] and the R system for statistical computing [171] provide interactive capabilities to solve complex problems in engineering, mathematics, logic, and statistics. SPIRAL is built on top of the computer algebra system GAP and uses many of these concepts.

III. ALGORITHM AND HARDWARE ABSTRACTION

We now discuss the different OL abstractions within SPIRAL that we use to capture specifications, algorithms, algorithmic degrees of freedom, hardware capabilities, and program transformations. The key idea is to capture this information into a single formal system that combines multiple rewriting systems with constraint solving and automatic performance tuning. Algorithms are captured symbolically as data flow graphs that are expanded recursively by identity rules. The target hardware is modeled by the set of all data flow graph fragments it can execute efficiently and a grammar that describes all programs that can be built from these fragments. Rewriting rules are essentially program transformations that map an algorithm to more efficient algorithms while preserving correctness.

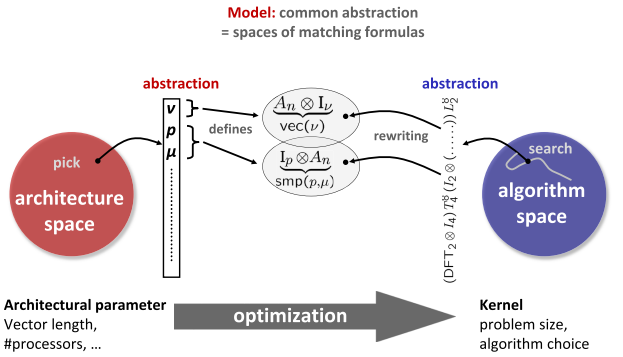


Fig. 1. ***SPIRAL's approach: The architecture space (red circle, left), the algorithm space (blue circle, right), and program transformations (shown as space in between) are abstracted in a joint formal framework.***

The overall approach is shown in Fig. 1. The architecture space (red circle, left), the algorithm space (blue circle, right), and program transformations (shown as space in between) are abstracted in a joint formal framework. Abstracting the three components in a compatible way allows SPIRAL to impose architecture requirements on algorithms and utilize the necessary program transformations.

In this section, we first discuss the algorithm abstraction, followed by the formalization of data layout and program transformations, and finally the hardware abstraction. In the next section (Section IV), we will discuss how these abstractions interact to implement rewriting, constraint solving, and autotuning in SPIRAL's code synthesis process.

A. Algorithm Abstraction

1) Specification: In SPIRAL, the top-level objects are specifications of computational kernels. A kernel is a function that performs a mathematical operation on its input data to produce its output. Kernels are modeled as parameterizable mathematical operators that map vectors to vectors. SPIRAL operators are stateless (any state would have to be an explicit parameter and matching result value). Higher-dimensional data are linearized and sparsity is abstracted as discussed below. In general, operators can take multiple-input vectors and produce multiple-output vectors. We allow a range of mathematical base types for the vectors, including fields ($\mathbb{R}$, $\mathbb{C}$, $\mathrm{GF}(k)$), rings ($\mathbb{Z}$, $\mathbb{Z}_n$ with n not prime), and semi-rings (e.g., min/sum semiring, etc. [172]). Operators act as problem specifications in our formal system.

For instance, the scalar product/dot product is mathematically a bilinear operator and defined in SPIRAL as

$$< .,. >_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}; (x, y) \mapsto x \cdot y. \qquad (1)$$

Note that we annotate the operator symbol by its vector length n, which will allow us to more concisely describe algorithms and transformations. A specification such as (1) explains unambiguously the input–output behavior of the operator (its semantics) but does not describe *how* the operator application is to be computed. Digital signal processing examples of operators defined in SPIRAL are the discrete Fourier transform (DFT)

$$\mathrm{DFT}_n : \mathbb{C}^n \to \mathbb{C}^n; x \mapsto \left[\omega_n^{ij}\right]_{i,j} x \tag{2}$$

with $\omega_n = \sqrt[n]{-1}$ a primitive nth root of 1. SPIRAL defines more than 50 linear digital signal processing transforms [1] and a number of bilinear and nonlinear transforms [54].

Beyond linear transforms, SPIRAL defines numerical linear algebra operations such as the circular convolution [54]

$$\mathrm{Conv}_n : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n; (x, y) \mapsto x \circledast y = \sum_{i=0}^{n-1} x_i y_{(n-i) \bmod n} \tag{3}$$

and matrix–matrix multiply [54]

$$\mathrm{MMM}_{k,m,n} : \mathbb{R}^{m \times k} \times \mathbb{R}^{k \times n} \to \mathbb{R}^{m \times n}; (A, B) \mapsto AB \tag{4}$$

which are bilinear as the scalar product. In SPIRAL, we implicitly use the isomorphism $\mathbb{R}^{m \times n} \cong \mathbb{R}^{mn}$ to abstract away tensor rank and convert all higher rank objects to vectors.

Examples of nonlinear operators defined in SPIRAL include polynomial evaluation and infinity norm

$$\mathrm{P}_n : \mathbb{R}^{n+1} \times \mathbb{R} \to \mathbb{R}; (a, x) \mapsto \sum_{i=0}^{n} a_i x^i, \tag{5}$$

$$\|.\|_\infty^n : \mathbb{R}^n \to \mathbb{R}; x \mapsto \|x\|_\infty \tag{6}$$

and checking if a point $x \in \mathbb{R}^n$ is inside a polytope given by a matrix A of polytope face normal vectors and a vector b of polytope face displacements

$$\mathrm{Inside}_{A,b}^n : \mathbb{R}^n \to \mathbb{Z}_2; x \mapsto Ax - b < (0, \ldots, 0). \tag{7}$$

Equation (7) can be used to implement geofencing for unmanned aerial vehicles (UAVs) [74]. A more complicated nonlinear example is the statistical z-test, given by

$$\mathrm{zTest}_{\mu,\alpha}^n : \mathbb{R}^n \to \mathbb{R}; x \mapsto \frac{\bar{x} - \mu}{\sigma(x)/\sqrt{n}} < \Phi^{-1}(1 - \alpha/2) \tag{8}$$

where μ is the population mean, $\bar{x}$ is the sample mean, $\sigma(x)$ is the sample standard deviation, and Φ^{-1} is the inverse error function.

Beyond the kernels shown in (1)–(8) we have modeled many more kernels as OL operators: the Viterbi decoder [58], polar formatting synthetic aperture radar (SAR) [56], Euler integration, statistical tests [65], wavelet transforms and JPEG2000 image compression [53], [57], the multigrid V cycle [64], quantum chemistry kernels used in ONETEP [61], the operations needed in the physical layer of SDR [60], a range of dense linear algebra kernels [63], and others.

Unambiguously declaring the input–output behavior of a kernel as function of all parameters is the first step required for program generation with SPIRAL, and developing the exact specification of a kernel is often a hard research problem in particular for higher-level operations such as SAR imaging, Viterbi decoders, and the multigrid V cycle, which require many algorithm and implementation choices. Top level operators often require a number of helper operators to be defined to express algorithms cleanly through breakdown rules, as discussed next.

2) Algorithms: In SPIRAL, algorithms describe how an operation given by a specification is turned into a computation directly or through other operators. For instance, an FFT is an algorithm to compute the DFT (which is a specification), usually through smaller DFTs. Similarly, computing a circular convolution via DFT and pointwise multiplication in the frequency domain or evaluating a polynomial via the Horner scheme are considered algorithms.

More formally, algorithms break down operators into (usually smaller) other operators in the divide-and-conquer style or iterative algorithms. Such algorithmic decompositions are modeled as breakdown rules in SPIRAL's rewriting system and may expose algorithmic degrees of freedom. Recursive application of breakdown rules yields a fully specified algorithm that is expressed as a rule tree. The rules explain how a rule tree is translated into a flat data flow graph. The DSL OL is used to capture these data flow graphs. OL programs may contain iterators (similar to map and reduce/fold in functional languages).

3) Operator Language: SPIRAL uses the OL to represent algorithmic breakdown rules and data flow graphs. We often refer to OL programs as OL formulas. OL consists of atomic operators such as (1)–(8) and others including auxiliary ones. Higher-order functions (operations) construct operator expressions, i.e., new operators from others.

Linear operators play an important role in SPIRAL. Originally, SPIRAL's language was called signal processing language (SPL) [13] and focused on linear operators, composition, and the Kronecker product to describe linear signal transforms and their algorithms (like the FFT) [1]. OL generalizes SPL to allow for multiple inputs and operator nonlinearity. In OL, matrices are interpreted as linear operators, and matrix multiplication is interpreted as

composition of linear operators. Thus, OL is a generalization of SPL and any SPL expression can be cast in OL.

Important linear operators in SPIRAL include the identity matrix I_m and the stride permutation matrix L_n^{mn}, which permutes the elements of the input vector as $in + j \mapsto jm + i,\ 0 \le i < m,\ 0 \le j < n$. If the vector x is viewed as an $n \times m$ matrix, stored in row-major order, then L_m^{mn} performs a transposition of this matrix. Also important are diagonal matrices, e.g., to describe the twiddle matrix T_n^{mn} needed in the most commonly used FFTs.

Higher-order functions create new OL operators from existing OL operators. Important higher-order functions included within SPIRAL include function composition $\circ$, the Cartesian product $\times$, the direct sum $\oplus$, and the Kronecker product or tensor product $\otimes$

$$(A \circ B)(x) = A(B(x)), \tag{9}$$
$$(A \times B)(x, y) = A(x) \times B(y), \tag{10}$$
$$(A \oplus B)(x \oplus y) = A(x) \oplus B(y), \text{and} \tag{11}$$
$$(A \otimes B)(x \otimes y) = A(x) \otimes B(y). \tag{12}$$

The Kronecker product of matrices A and B is defined as

$$A \otimes B = [a_{k,\ell}B], \quad \text{for } A = [a_{k,\ell}]. \tag{13}$$

It replaces every entry $a_{k,\ell}$ of A by the matrix $a_{k,\ell}B$. The Kronecker product binds weaker than matrix product but stronger than matrix addition. For scalar operations such as +, -, ·, /, >, =, ≠, <, etc., we define the infix operations

$$(A \diamond B)(x) = A(x) \diamond B(x) \quad \text{for} \quad \diamond \in \{+ - \cdot /, <, \ldots\}. \tag{14}$$

In (9)–(14), we assume that the operators A and B are of arity (1,1), i.e., that they have one input and one output vector. Generalization to higher arities (multiple input or output vectors) is technically complex but conceptually straightforward [54]. SPIRAL applies the isomorphism between $\mathbb{R}^{m+n}$ and the direct sum and Cartesian product of $\mathbb{R}^m$ and $\mathbb{R}^n$

$$\mathbb{R}^m \times \mathbb{R}^n \cong \mathbb{R}^m \oplus \mathbb{R}^n \cong \mathbb{R}^{m+n}$$

as needed in type unification to simplify handling of higher arities.

More recently, an additional class of OL operators that model functional programming constructs such as `map` and `fold` [65], [74] was introduced (into SPIRAL) in OL

$$\text{Map}_{f_i(.)}^n : \mathbb{R}^n \to \mathbb{R}^n; \bigoplus_{i=0}^{n-1} x_i \mapsto \bigoplus_{i=0}^{n-1} f_i(x_i), \tag{15}$$

$$\text{Fold}_{f_i(.,.),z}^n : \mathbb{R}^n \to \mathbb{R}; x_0 \oplus x_r \mapsto \begin{cases} f_0(x_0, z), & n = 1 \\ f_0(x_0, \ \text{Fold}_{f_{i+1}(.,.),z}^{n-1}(x_r)). \end{cases} \tag{16}$$

This additional class of operators is introduced so that SPIRAL-generated code can be formally verified using standard techniques common to proving the correctness of functional languages.

4) Breakdown Rules: Algorithms and their degrees of freedom are expressed in SPIRAL as breakdown rules. A breakdown rule is a rewriting rule that matches an OL operator (e.g., specifying a problem) and replaces the matched expression with a more complicated expression made up of simpler/smaller OL operators. The pattern matching performed by the rule encodes possible constraints on the kernel (e.g., a subset of input sizes to which it is applicable). For instance, the general radix Cooley–Tukey FFT algorithm is expressed as breakdown rule [1], [7]

$$\text{DFT}_{mn} \to (\text{DFT}_m \otimes \text{I}_n)\text{T}_n^{mn}(\text{I}_m \otimes \text{DFT}_n)\text{L}_m^{mn}. \tag{17}$$

In (17), DFT_{mn} is the nonterminal and is translated into a right-hand side that consists of an SPL/OL expression that contains new nonterminals DFT_m and DFT_n. The left-hand side DFT_{mn} encodes the constraint that the size of the DFT be a composite number and therefore can be factored into m and n. Further, the new DFT nonterminals created by (17) are of smaller size than the original nonterminal. Thus recursive application of (17) will terminate and we need to provide a terminal rule that translates DFT_2 into an atomic SPL/OL operator (the butterfly)

$$\text{DFT}_2 \to \text{F}_2 \quad \text{with} \quad \text{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{18}$$

k-dimensional multi-dimensional DFTs (MDFTs) are broken down into lower dimensional DFTs through tensor products. For instance, a complete decomposition into one-dimensional DFTs is

$$\text{DFT}_{m_1 \times \cdots \times m_k} \to \text{DFT}_{m_1} \otimes \ldots \otimes \text{DFT}_{m_k}. \tag{19}$$

As an example, a DFT_8 operator can be fully expanded into an SPL/OL formula by applying (17) twice and (18) thrice to get

$$\text{DFT}_8 \to (\text{F}_2 \otimes \text{I}_4)\text{T}_4^8\big(\text{I}_2 \otimes (\text{F}_2 \otimes \text{I}_2)\text{T}_2^4(\text{I}_2 \otimes \text{F}_2)\text{L}_2^4\big)\text{L}_2^8. \tag{20}$$

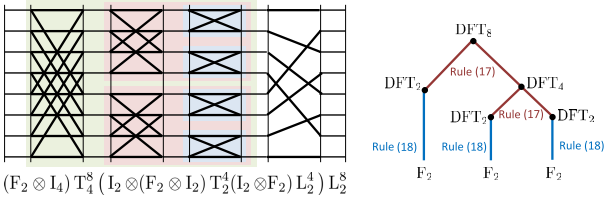


Fig. 2. ***Data flow graph, OL formula, and rule tree for DFT_8 as expanded in (116) (see Fig. 7).***

Table 1 Spiral OL Breakdown Rules for a Multigrid Solver $\text{MGSolvePDE}_{n,\omega,r,m}$ for an $n \times n$ Discretized 2-D Poisson Equation With Dirichlet Boundary Conditions and Parameters ω, r, and m. The Solver Uses a Richardson Smoother With Parameter ω and r Iterations and Injection as Restriction Operator. It Performs m Multigrid Cycles [64].

$$\text{MGSolvePDE}_{n,\omega,r,m} \to [\,\mathrm{I}_{n^2} \mid 0_{n^2}\,] \cdot \left(\prod_{i=0}^{m-1} \text{MGCycle}_{n,\omega,r} \right) \cdot \begin{bmatrix} 0_{n^2} \\ \mathrm{I}_{n^2} \end{bmatrix} \tag{21}$$

$$\text{MGCycle}_{n,\omega,r} \to \text{CGC}_{n,\omega,r} \cdot \text{Richardson}_{n,\omega,r} \tag{22}$$

$$\text{CGC}_{n,\omega,r} \to \begin{bmatrix} \text{CoarseError}_{n,\omega,r} \\ 0_{n^2} \mid \mathrm{I}_{n^2} \end{bmatrix} \tag{23}$$

$$\text{CoarseError}_{n,\omega,r} \to \text{Interpolate}_n \cdot \text{Scatter}_n \cdot \text{Solve}_{n,\omega,r} \cdot \text{Gather}_n \cdot \text{Residual}_n \tag{24}$$

$$\text{Interpolate}_n \to \text{Tridiag}_n\left(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2\right) \otimes \text{Tridiag}_n\left(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2\right) \tag{25}$$

$$\text{Scatter}_n \to \mathrm{S}_{\mathbb{I}_{(n-1)/2} \to \mathbb{I}_n;\, i \mapsto 2i+1} \otimes \mathrm{S}_{\mathbb{I}_{(n-1)/2} \to \mathbb{I}_n;\, i \mapsto 2i+1} \tag{26}$$

$$\text{Solve}_{n,\omega,r} \to \begin{cases} \frac{1}{4}\,\mathrm{I}_1, & n = 1 \\ \left[\mathrm{I}_{((n-1)/2)^2} \mid 0_{((n-1)/2)^2}\right] \cdot \text{MGCycle}_{(n-1)/2,\omega,r} \cdot \begin{bmatrix} 0_{((n-1)/2)^2} \\ \mathrm{I}_{((n-1)/2)^2} \end{bmatrix}, & n > 1 \end{cases} \tag{27}$$

$$\text{Gather}_n \to \mathrm{G}_{\mathbb{I}_{(n-1)/2} \to \mathbb{I}_n;\, i \mapsto 2i+1} \otimes \mathrm{G}_{\mathbb{I}_{(n-1)/2} \to \mathbb{I}_n;\, i \mapsto 2i+1} \tag{28}$$

$$\text{Residual}_n \to \left[(\text{Tridiag}_n(1, -2, 1) \otimes \mathrm{I}_n) + (\mathrm{I}_n \otimes \text{Tridiag}_n(1, -2, 1)) \mid \mathrm{I}_{n^2} \right] \tag{29}$$

$$\text{Richardson}_{n,\omega,r} \to \prod_{i=0}^{r-1} \begin{bmatrix} \text{ResidueLaplace}_{n,\omega} & \omega\,\mathrm{I}_{n^2} \\ 0_{n^2} & \mathrm{I}_{n^2} \end{bmatrix} \tag{30}$$

$$\text{ResidueLaplace}_{n,\omega} \to \left(\text{Tridiag}_n(\omega, 1/2 - 2\omega, \omega) \otimes \mathrm{I}_n \right) + \left(\mathrm{I}_n \otimes \text{Tridiag}_n(\omega, 1/2 - 2\omega, \omega) \right) \tag{31}$$

Table 2 OL Breakdown Rules for Polar Formatting Synthetic Aperture Radar (SAR) [27], [51], [56], Formalizing Algorithms From [173] and [174].

$$\text{PDFT}_N^{\sigma} \to \text{DFT}_N \mathrm{S}_{\sigma} \tag{32}$$

$$\text{PDFT}_{kmn}^{\sigma \otimes km} \to (\text{DFT}_m \otimes \mathrm{I}_{kn})\, \mathrm{T}_n^{mn}\, \mathrm{L}_m^{kmn} (\text{PDFT}_{kn}^{\sigma \otimes k} \otimes \mathrm{I}_m) \tag{33}$$

$$\text{SAR}_{s,\alpha} \to \text{DFT}_{m_2 \times n_2}\, \text{2D-Intp}_{(k,r,m,n,\alpha_r,\alpha_a)}^{m_1 \times n_1 \to m_2 \times n_2} \tag{34}$$

$$\text{2D-Intp}_{(k,r,m,n,\alpha_r,\alpha_a)}^{m_1 \times n_1 \to m_2 \times n_2} \to \left(\text{Intp}_{(k,r,m,n,\alpha_a(i))}^{n_1 \to n_2} \otimes_i \mathrm{I}_{m_2} \right) \left(\mathrm{I}_{n_1} \otimes_i \text{Intp}_{(k,r,m,n,\alpha_r(i))}^{m_1 \to m_2} \right) \tag{35}$$

$$\text{Intp}_{(k,r,m,n,w)}^{u \to v} \to \left(\mathrm{I}_\ell \otimes_j \text{Intp}_{(k,w(j))}^{m \to n} \right) (\mathrm{I}_\ell \otimes^r \mathrm{I}_m) \tag{36}$$

$$\text{Intp}_{(k,(b,s))}^{m \to n} \to \mathrm{G}_{(b,s)}^{km \to n}\, \text{iDFT}_{km}\, \mathrm{S}_{\langle 0, k-1 \rangle \otimes (m/2)} \text{DFT}_m \tag{37}$$

$$\text{iDFT}_{km}\, \mathrm{S}_{\langle 0, k-1 \rangle \otimes (m/2)} \to \text{iPDFT}_{km}^{\langle 0, k-1 \rangle \otimes (m/2)} \tag{38}$$

The first application of (17) allows for a choice of $(m, n) = (4, 2)$ or $(m, n) = (2, 4)$, and for larger sizes requiring multiple rule applications there is a considerable degree of freedom to expand DFT_N into a fully expanded SPL/OL formula. The dataflow that is represented by the formula is shown in Fig. 2(left). Such algorithmic degrees of freedom are one source of the optimization space that is leveraged by SPIRAL. SPIRAL defines more than 50 linear transform nonterminals such as the DFT and more than 200 breakdown rules such as (17) [1].

Tables 1 and 2 show breakdown rule sets for more complex algorithms. Table 1 shows the set of OL breakdown rules for a multigrid solver $\text{MGSolvePDE}_{n,\omega,r,m}$ for an $n \times n$ discretized 2-D Poisson equation with Dirichlet boundary conditions and parameters ω, r, and m. The solver uses a Richardson smoother with parameter ω and r iterations and injection as restriction operator. It performs m multigrid cycles [64]. Note the interconnected set of auxiliary operators needed.

Table 2 shows the OL breakdown rule set for polar-formatting synthetic aperture radar (SAR [51], [56]), formalizing the algorithms from [173] and [174].

5) Nonlinear OL Breakdowns: The step from linear operators to non-linear operators requires the use of OL constructs in breakdown rules. For instance, matrix–matrix multiplication (MMM) as defined in (4) is a bilinear operator. Optimized MMM implementations utilize multilevel cache and register blocking [91], and blocking of MMM can be described as

$$\text{MMM}_{k,m,n} \to \text{MMM}_{k/b_1, m/b_2, n/b_3} \otimes \text{MMM}_{b_1, b_2, b_3}. \tag{39}$$

In (39), the multilinear tensor product captures the recursiveness of MMM in the same fashion that (17) captures the recursiveness of the DFT and (19) captures the higher dimensional structure of the MDFT.

Functional programming-style operators are equally decomposed by rewrite rules that express them as para-

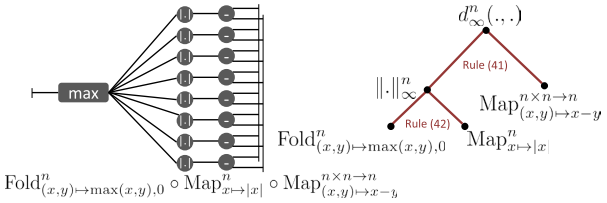


Fig. 3. ***Data flow graph, OL formula, and rule tree for $d_\infty(.,.)$ as expanded in (43).***

meterized OL expressions. For instance, the Chebyshev distance (distance induced by the infinity norm)

$$d_\infty(u,v)=\|u-v\|_\infty,\quad u,v\in\mathbb{R}^n \tag{40}$$

is captured by the rules

$$d^n_\infty(.,.)\to\|.\|^n_\infty\circ\mathrm{Map}^{n\times n\to n}_{(x,y)\mapsto x-y} \tag{41}$$

$$\|.\|^n_\infty\to\mathrm{Fold}^n_{(x,y)\mapsto\max(x,y),0}\circ\mathrm{Map}^n_{x\mapsto|x|}. \tag{42}$$

Note that we annotated the OL definition of $d_\infty(u,v)$ with dimensionality n, leading to the OL nonterminal $d^n_\infty(u,v)$ for the Chebyshev distance. The nonterminal $d^8_\infty(.,.)$ can be fully expanded into an SPL/OL expression (a functional style formula) by applying (41) followed by (42), leading to

$$d^8_\infty(.,.)\to\mathrm{Fold}^8_{(x,y)\mapsto\max(x,y),0}\circ\mathrm{Map}^8_{x\mapsto|x|}\circ\mathrm{Map}^{8\times8\to8}_{(x,y)\mapsto x-y}. \tag{43}$$

Fig. 3 shows the corresponding data flow graph and rule tree.

Similarly, polynomial evaluation (5) is expressed as the breakdown rule

$$\mathrm{P}_n(.,.)\to<.,.>_n\circ\left(\mathrm{I}_{n+1}\times\left(\mathrm{Map}^{n+1}_{x\mapsto x^i}\circ\begin{bmatrix}1&\cdots&1\end{bmatrix}^T\right)\right). \tag{44}$$

$\mathrm{P}_n(.,.)$ is an arity (2,1) operator. The cross product $\times$ passes the first argument to the identity matrix and the second argument to the Map construct. The linear operator $\begin{bmatrix}1&\cdots&1\end{bmatrix}^T$ takes the second input of $\mathrm{P}_n(a,x)$ (the scalar x) and replicates it for $n+1$ times to form a vector. Then $\mathrm{Map}^{n+1}_{x\mapsto x^i}$ computes all exponents of x from 0 to n, yielding the vector $(1,x,x^2,\ldots,x^n)$. Note that the map function $f_i(x)=x^i, i=0,\ldots,n$ depends on the index of the mapped element. Furthermore, I_{n+1} forwards the coefficient vector $a=(a_n,\ldots,a_0)$ [the first input of $\mathrm{P}_n(.,.)$] to the scalar product $<.,.>_n$, which performs the pointwise multiplications a_ix^i for $i=0,\ldots,n$ and then performs the final sum, yielding the result $\mathrm{P}_n(a,x)=\sum_{i=0}^n a_ix^i$.

The scalar product is a special case of MMM

$$<.,.>_n\to\mathrm{MMM}_{n,1,1} \tag{45}$$

and thus can be decomposed via (39). The check for being inside a polytope (7) [74] is expressed as an OL breakdown rule by

$$\mathrm{Inside}^n_{A,b}(.)\to\mathrm{Fold}^n_{(x,y)\mapsto(x<0)\wedge y,\mathrm{true}}\circ\mathrm{Map}^n_{x\mapsto x-b}\circ\mathrm{MMM}_{n,1,n}(A,.). \tag{46}$$

Inside (46) the matrix–vector product Ax required by (7) is expressed as degenerate partially evaluated MMM $\mathrm{MMM}_{n,1,n}(A,.)$. The operator $\mathrm{Map}^n_{x\mapsto x-b}$ subtracts the right-hand side b from the result, and $\mathrm{Fold}^n_{(x,y)\mapsto(x<0)\wedge y,\mathrm{true}}$ checks that all entries of the resulting vector are negative. Finally, the OL rule set for the z-test (8) for a sample size of n at α confidence interval is shown in Table 3.

Note that all breakdown rules are point-free: operators are expressed through other operators but the input and output is not part of the expression. This property is important for data flow optimization (discussed next) and program generation.

B. Program Transformations as Data Flows

In the previous section, we discussed how specifications are expanded into fully expanded (terminated) OL expressions that represent data flow graphs of the algorithm. In this section, we discuss how SPIRAL performs data flow optimizations that change the geometry and thus locality and shape of parallelism in the data flow graph.

1) Approach: A human developer typically starts from a given algorithm/data flow graph, and then modifies it to better match the underlying hardware. SPIRAL takes a different approach: The data flow graph is made to fit the targeted hardware while it is derived recursively. This is achieved by a two-pronged approach: 1) program/data flow transformations are also cast as breakdown rules and are treated equivalently to algorithmic breakdown rules; and 2) hardware properties are encoded as constraints on these breakdown rules.

In this section, we discuss how program transformations are encoded as breakdown rules. Section III-C discusses how hardware is modeled through breakdown rules and

Table 3 The Statistical z-Test Represented as a Set of OL Breakdown Rules [74].

$$\mathrm{Mean}_n\to\mathrm{Map}^1_{x\mapsto x/n}\circ\mathrm{Fold}^n_{(a,b)\mapsto(a+b),0} \tag{47}$$

$$\mathrm{Variance}_n\to\left(\mathrm{Fold}^n_{(a,b)\mapsto(a+b),0}\circ\mathrm{Map}^n_{x\mapsto x^2}\right)-\left(\mathrm{Map}^1_{x\mapsto x^2}\circ\mathrm{Mean}_n\right) \tag{48}$$

$$\mathrm{zTest}_{n,\alpha}\to\left(\mathrm{Mean}_n\,/\,\mathrm{Map}^1_{x\mapsto\sqrt{x}}\circ\mathrm{Variance}_n\,/n\right)<\Phi^{-1}(1-\alpha/2), \tag{49}$$

constraints on them, and Section IV discusses how the resulting constraint problem is solved.

2) Memory Abstraction: Many performance-relevant hardware constraints translate into the requirement that certain contiguous data blocks are held in a certain class of memory, and these data are moved in contiguous blocks of certain sizes. Recall that operators map vectors to vectors. Vectors impose a natural neighborhood relation: for a vector x the component x_i is neighbor of components x_{i-1} and x_{i+1}. A subvector $(x_i, \ldots, x_{i+k-1})^T$ of length k is contiguous and thus naturally provides the concept of a block.

Therefore, we impose the interpretation of vectors that OL formulas operate on as being mapped to physical storage contiguously, i.e., the memory for a vector is viewed as array (in C notation) `vector_t X[n`, and x_i is mapped to `X[i`. Any different data layout needs to be made explicit through permutation operators as part of the OL formula, not by informally interpreting the vector data layout differently. For example, block cyclic data distribution becomes an explicit permutation in an OL formula.

Partitioning a vector into equally-sized blocks treats the vector as vector of vectors. Blocking may happen recursively (blocks of blocks). For constant block size at each recursion level, this interpretation implies that data vectors are seen as linearized tensors (high-dimensional matrices), which explains the importance of the tensor product and Kronecker product in SPIRAL's formal framework. This implies that blocking higher-dimensional objects (e.g., matrices) introduces explicit permutations, since subblocks of higher-dimensional blocks contiguous in memory are themselves not contiguous any more. We use this approach to model structural hardware features that require data locality for performance or correctness reasons, as discussed in Section III-C.

3) The Role of Permutations: SPIRAL's memory abstraction implies that data required to compute an operator $A_n(.)$ are in the local memory (if there are multiple address spaces) or in the smallest level of the memory hierarchy (e.g., in the L1 cache for standard CPUs). If n is too large (i.e., the memory is too small to hold the necessary data), then the operator needs to be recursively decomposed until this condition holds. In this respect, SPIRAL's memory model is related to the idea of cache oblivious algorithms [175]. SPIRAL implies that a data access that crosses memory regions that are contiguous needs to be explicitly marked in the OL data flow abstraction.

Permutations are used to capture both data reordering and communication. They are expressed as permutation matrices that are parameterized by the actual permutation (function). The most prominent example in SPIRAL is the stride permutation L^{mn}_n introduced in Section III-A.

4) Data Flow Representation: With the above convention of how vectors are mapped to memory we can now describe locality enhancing optimizations as breakdown rules for OL expressions.

First we discuss the meaning of tensor/Kronecker products with identity matrices in the context of data flow graphs. For a linear operator $A_m \in \mathbb{R}^{m\times m}$ and $B_n \in \mathbb{R}^{n\times n}$ the Kronecker product $A_m \otimes B_n$ defined in (13) is a matrix of size $mn \times mn$. For $A_m = \mathrm{I}_m$ the tensor product becomes a block-diagonal matrix

$$\mathrm{I}_m \otimes B_n = \begin{bmatrix} B_n & & \\ & \ddots & \\ & & B_n \end{bmatrix}. \tag{50}$$

The operator $\mathrm{I}_m \otimes B_n$ applies the same operator B_n on contiguous subvectors of length n, i.e.,

$$(\mathrm{I}_m \otimes B_n)(x_0 \oplus \ldots \oplus x_{m-1}) = (B_n x_0) \oplus \ldots \oplus (B_n x_{m-1})$$

and can be seen as a "parallel operation." The "flipped" tensor product is a block matrix where each block is a constant-diagonal matrix

$$A_m \otimes \mathrm{I}_n = \begin{bmatrix} a_{0,0}\mathrm{I}_n & \ldots & a_{0,m-1}\mathrm{I}_n \\ \vdots & \ddots & \vdots \\ a_{m-1,0}\mathrm{I}_n & \ldots & a_{m-1,m-1}\mathrm{I}_n \end{bmatrix}. \tag{51}$$

Equation (51) applies the operator A_m to subvectors of length n and can be seen as a "vector operation." A common interpretation is that for a linearized $m \times n$ input matrix $x \in \mathbb{R}^{mn}$ the operation $(\mathrm{I}_m \otimes B_n)x$ applies the operator B_n to the "rows" of x while $(A_m \otimes \mathrm{I}_n)x$ applies the operator A_m to the "columns" of x. The tensor product is separable: it can be factorized into a "row pass" and "column pass" in two ways, written as breakdown rule

$$A_m \otimes B_n \rightarrow \begin{cases} (A_m \otimes \mathrm{I}_n)(\mathrm{I}_m \otimes B_n) \\ (\mathrm{I}_m \otimes B_n)(A_m \otimes \mathrm{I}_n) \end{cases}. \tag{52}$$

For instance, (19) together with (52) captures a large class of multidimensional FFT algorithms including the row-column algorithm, as well as the slab and pencil decomposition. Further, the identity matrix is the tensor product of identity matrices

$$\mathrm{I}_{mn} \rightarrow \mathrm{I}_m \otimes \mathrm{I}_n \tag{53}$$

which can be used to represent loop tiling. The following breakdown rule describes the use of (53) to tile the mn iterations of applying the operator A into m iterations of n computations of A:

$$I_{mn} \otimes A \rightarrow I_m \otimes (I_n \otimes A).$$

A generalization that allows the blocks B_n to be different in the occurring entries (but not in size) is expressed by

$$\mathrm{I}_m \otimes_i B_n^{(i)} = \bigoplus_{i=0}^{m-1} B_n^{(i)}. \tag{54}$$

The definitions in this section generalize to rectangular matrices $A_{k\times\ell} \in \mathbb{R}^{k\times\ell}$ and $B_{m\times n} \in \mathbb{R}^{m\times n}$.

5) Data Layout Transformations: The most important permutation operator in SPIRAL's framework is the stride permutation L_m^{mn}. As discussed in Section III-A, it can be viewed as transposing a linearized matrix: it blocks a vector of length mn into m vectors of length n, treats the vector-of-vectors as a matrix of size $m \times n$, transposes the matrix (new size is $n \times m$), reinterprets the matrix as n vectors of length m, and finally linearizes the vector-of-vectors into a single vector of length mn. Note that the reshaping steps are virtual: under SPIRAL's matrix/tensor linearization rules the data flow graph of a matrix transposition and of a stride permutation are the same.

The stride operation derives its importance from the fact that it commutes the tensor product and thus enables formal derivation of loop transformations such as tiling, blocking, loop interchange, and loop merging/fusion/fission. Written as breakdown rule

$$A_m \otimes B_n \rightarrow \mathrm{L}_m^{mn}(B_n \otimes A_m)\mathrm{L}_n^{mn}. \tag{55}$$

Furthermore, matrix transposition can be blocked into a transposition of blocks followed by or preceded by a transposition of the blocks [9]

$$\mathrm{L}_{km}^{k^2mn} = \left(\mathrm{I}_k \otimes \mathrm{L}_m^{kmn}\right)\left(\mathrm{L}_k^{k^2} \otimes \mathrm{I}_{mn}\right)\left(\mathrm{I}_k \otimes \mathrm{L}_k^{kn} \otimes \mathrm{I}_m\right). \tag{56}$$

Stride permutations have multiplicative and transposition/inversion properties, which yield the identities

$$\left(\mathrm{L}_m^{mn}\right)^T \rightarrow \mathrm{L}_n^{mn} \tag{57}$$
$$\left(\mathrm{L}_m^{mn}\right)^{-1} \rightarrow \mathrm{L}_n^{mn} \tag{58}$$
$$\mathrm{L}_{km}^{kmn} \rightarrow \mathrm{L}_k^{kmn}\mathrm{L}_m^{kmn} \tag{59}$$
$$\mathrm{L}_n^{kmn} \rightarrow \mathrm{L}_{kn}^{kmn}\mathrm{L}_{mn}^{kmn} \tag{60}$$
$$\mathrm{L}_n^{kmn} \rightarrow \left(\mathrm{L}_n^{kn} \otimes \mathrm{I}_m\right)\left(\mathrm{I}_k \otimes \mathrm{L}_n^{mn}\right) \tag{61}$$
$$\mathrm{L}_{km}^{kmn} \rightarrow \left(\mathrm{I}_k \otimes \mathrm{L}_m^{mn}\right)\left(\mathrm{L}_k^{kn} \otimes \mathrm{I}_m\right). \tag{62}$$

Tensor products of stride permutations with identity matrices can be seen as block transpositions, transposition of blocks, or tensor rotations. For instance, $\mathrm{I}_k \otimes \mathrm{L}_m^{mn}$ is a block diagonal matrix of size $kmn \times kmn$ with blocks of size $mn \times mn$. Similarly, the operation $\mathrm{L}_m^{mn} \otimes \mathrm{I}_k$ is expressing the reordering of packets of size k. Both tensor products can be seen as rotations of a linearized rank-3 tensor in the xy or yz plane. As an aside, the tensor product of two stride permutations is rotating a linearized rank-4 tensor simultaneously in the xy and zu plane that also can be seen as an operation for a block matrix that simultaneously performs a transposition of blocks and transposition within blocks.

6) Nonlinear and Higher-Arity Data Flows: So far, we have only discussed the tensor product of linear operators. The generalization to multilinear is standard in the field of multilinear algebra and tensor calculus, and we use the standard definitions. For instance, (39) describes recursive blocking of MMM as tensor decomposition of the operator $\mathrm{MMM}_{k,m,n}$. Generalizations of rules (52)–(62) exist for the multilinear tensor product and generalize to the iterative sum (54).

SPIRAL also defines a generalization of (50) and (51) for a tensor product of an identity matrix and a nonlinear operator [54]. For the simple case of $B_n : \mathbb{R}^n \rightarrow \mathbb{R}^n$ the generalization is straightforward: the operator $\big(\mathrm{I}_m \otimes B_n(.)\big)(.)$ applies the operator $B_n(.)$ on contiguous subvectors of length n, i.e.,

$$\big(\mathrm{I}_m \otimes B_n(.)\big)(x_0 \oplus \ldots \oplus x_{m-1}) = B_n(x_0) \oplus \ldots \oplus B_n(x_{m-1}). \tag{63}$$

Using (63) we can break down (15) and (16) to capture program transformations for the Map and Fold operators

$$\mathrm{Map}_{f_i(.)}^{mn} \rightarrow \mathrm{I}_m \otimes_j \mathrm{Map}_{f_{jn+i}(.)}^{n}, \quad \text{and} \tag{64}$$
$$\mathrm{Fold}_{f_i(.,.),z}^{mn} \rightarrow \mathrm{Fold}_{f_{in}(.,.),z}^{m} \circ \left(\mathrm{I}_m \otimes_j \mathrm{Fold}_{f_{jn+i}(.,.),z}^{n}\right). \tag{65}$$

For more general situations and arities beyond $(1, 1)$ these definitions and breakdown rules become unwieldy and for brevity we will not introduce them in this paper.

The relatively small set of identities and breakdown rules introduced in this section gives rise to a large possible space of data layout and blocking transformations that can be exploited in locality optimizations and work hand-in-hand with algorithmic breakdown rules. Together with hardware parameters (discussed next) they are powerful enough to enable SPIRAL to derive efficient data flow graphs for a wide range of problems and platforms.

C. Hardware Abstraction

We now discuss how hardware is abstracted in SPIRAL's formal framework.

1) Approach: The framework is designed to capture the major current architectural and microarchitectural features that require data flow optimizations for performance, including 1) SIMD vector instruction sets, SIMT, and VLIW architectures [22], [25]; 2) cache memories (multiple cache levels and various topologies) [20]; 3) scratch pads (explicitly managed local storage) and multiple address spaces (CPU/device memory) [38]; 4) shared memory parallelism (multicore, manycore, and hyperthreading) [19], [20]; 5) distributed memory parallelism

(message passing) [47], [49]; and 6) streaming parallelism (processor arrays and FPGAs) [40], [45], [46]. We expect that also most future hardware paradigms are composed from these features. SPIRAL's hardware model provides a formal way to enumerate "good programs" for a target platform. This is achieved by modeling the hardware in SPIRAL's rewrite system through constrained terminal rules (the rule tree expansion stops), or as constraints on OL breakdown rules. SPIRAL does not use a detailed predictive model (as, e.g., [176] for MMM) but captures structurally which kind of computations work or do not work well. Further autotuning search (or learning) is then used to select which program runs fastest.

Hardware or architecture features that do not require direct changes to the data flow but may influence its parameters are not modeled at the formal level but handled in the backend compiler (see Section IV-B). These features include ISA details such as fused multiply–add instructions, special data type support (floating-point, double-double, fixed-point, finite field arithmetic), and microarchitectural features such as the number of physical and named registers, topology of execution pipelines, and others.

2) Tags: A hardware constraint on an OL operator is captured by a tag. Tags are symbolic annotations that are used in rewrite rules to control pattern matching. Tags carry parameters that provide high-level structural information of the hardware feature they describe. Operators or operator expressions can be tagged. For instance

$$\underbrace{\mathrm{I}_n \otimes A_n}_{\mathrm{vec}(\nu)}, \quad \underbrace{\mathrm{DFT}_n}_{\mathrm{smp}(2)\,\mathrm{vec}(4)}, \quad \text{and} \quad \underbrace{\mathrm{MMM}_{k,m,n}}_{\mathrm{mpi}(16)\,\mathrm{vec(avx\text{-}double)}}$$

specify a general operator expression $\mathrm{I}_n \otimes A_n$ to be ν-way SIMD vectorized, a DFT_n to be 2-way parallelized for shared memory and 4-way SIMD vectorized, and an $\mathrm{MMM}_{k,m,n}$ to be parallelized for 16 MPI processes and vectorized for the double-precision AVX instruction set, respectively. These examples show that an operator can be tagged with multiple tags (order matters: e.g., $\mathrm{smp}(2)\,\mathrm{smp}(4)$ captures a different nested parallelization from $\mathrm{smp}(4)\,\mathrm{smp}(2)$), and that tags can be generic (4-way SIMD vectorization) or specific (SIMD vectorization for AVX double-precision). Specific tags imply generic tags (AVX double precision implies 4-way SIMD vectorization, and 16 MPI processes imply 16-way distributed memory parallelism). Other hardware features are also captured by tags: for instance, a tag for ν-way streaming on FPGAs used below is given by $\mathrm{fpga}(\nu)$.

3) Tagged Breakdown Rules: Tagged breakdown rules propagate tags and constrain algorithmic or data flow optimization breakdown rules so that they become compatible with the target hardware. For instance, the propagation rule

$$\underbrace{A \circ B}_{\mathrm{tag}} \rightarrow \underbrace{A}_{\mathrm{tag}} \circ \underbrace{B}_{\mathrm{tag}} \quad \text{for } \mathrm{tag} \in \{\mathrm{par}(.),\, \mathrm{vec}(.), \ldots\} \tag{66}$$

states that tagging and function composition commutes for certain tags (to parallelize a sequence of operators, parallelize each operator independently and have barriers at the end of each parallel operator).

An example of a parallelization rule that expresses that the outermost loop should be parallelized is given by

$$\underbrace{\mathrm{I}_p \otimes A_n}_{\mathrm{tag}} \rightarrow \mathrm{I}_p \otimes_{\mathrm{tag}} A_n \quad \text{for } \mathrm{tag} \in \{\mathrm{smp}(p),\, \mathrm{mpi}(p),\, \mathrm{fpga}(n)\}. \tag{67}$$

The rule states that a loop of p iterations with loop body $A_n(.)$ can be perfectly parallelized across p processors by running one instance of $A_n(.)$ on each processor. Similarly, on an FPGA a block implementing $A_n(.)$ as a combinational data path can be invoked every cycle with a new input vector. Thus, $\mathrm{I}_p \otimes A_n$ is executed over p cycles as a pipelined IP block of streaming width n.

The symbol $\otimes_{\mathrm{tag}}$ is a tagged OL operation that carries the information that the loop resulting from this tensor product is parallel to the further stages in the rewriting system. Rule (67) "drops" the tag, i.e., there is no tagged OL object left on in the right-hand side of the rule—only a tagged OL operation. The removal of the tags and replacing them with a specific tagged OL operations restricts the applicable breakdown rules to a smaller, possibly more optimized, subset of breakdown rules that simplifies the implementation of the generator. Note that rule (53) may need to be applied first to make (67) applicable.

An example of a vectorization or streaming rule is given by

$$\underbrace{A_n \otimes \mathrm{I}_\nu}_{\mathrm{tag}} \rightarrow A_n \otimes_{\mathrm{tag}} \mathrm{I}_\nu \quad \text{for } \mathrm{tag} \in \{\mathrm{vec}(\nu),\, \mathrm{fpga}(\nu)\}. \tag{68}$$

It states that for any A_n, the construct $A_n \otimes \mathrm{I}_\nu$ can be implemented using vector instructions by promoting any scalar operations in A_n to ν-way vector operations. For example, the scalar operation `a+b` is replaced by `vec_add(a,b)`. SSE code for $\mathrm{F}_2 \otimes_{\mathrm{vec}(4)} \mathrm{I}_4$ is given by the following code snippet:

```
void F2xI4_SSE4x32f(__m128 *Y, __m128 *X) {
    Y[0] = _mm_add_ps(X[0], X[1]);
    Y[1] = _mm_sub_ps(X[0], X[1]);
}
```

Similarly, on an FPGA for a pipelined block A_n of streaming width $w = 1$ (i.e., which takes one input per cycle over n cycles), $A_n \otimes \mathrm{I}_\nu$ can be implemented with streaming width $w = \nu$ (consuming ν inputs per cycle) by replicating the logic for A_n for ν times.

Table 4 Advanced Vectorization and Parallelization Rules [19], [20], [22], [25].

$$\underbrace{\mathrm{L}^{mn}_{m}}_{\mathrm{vec}(\nu)} \to \underbrace{\left(\mathrm{L}^{mn/\nu}_{m} \otimes \mathrm{I}_{\nu}\right)\left(\mathrm{I}_{mn/\nu^2} \otimes \mathrm{L}^{\nu^2}_{\nu}\right)\left(\left(\mathrm{I}_{n/\nu} \otimes \mathrm{L}^{m}_{m/\nu}\right) \otimes \mathrm{I}_{\nu}\right)}_{\mathrm{vec}(\nu)}, \quad \nu \mid m,n \tag{71}$$

$$\underbrace{\left(\mathrm{I}_{m} \otimes A^{n\times n}\right)\mathrm{L}^{mn}_{m}}_{\mathrm{vec}(\nu)} \to \underbrace{\left(\mathrm{I}_{m/\nu} \otimes \left(\mathrm{L}^{n\nu}_{\nu}(A^{n\times n} \otimes \mathrm{I}_{\nu})\right)\right)\left(\mathrm{L}^{mn/\nu}_{m/\nu} \otimes \mathrm{I}_{\nu}\right)}_{\mathrm{vec}(\nu)}, \quad \nu \mid m \tag{72}$$

$$\underbrace{\left(\mathrm{I}_{k} \otimes \left(\left(\mathrm{I}_{m} \otimes A^{n\times n}\right)\mathrm{L}^{mn}_{m}\right)\right)\mathrm{L}^{kmn}_{k}}_{\mathrm{vec}(\nu)} \to \underbrace{\left(\mathrm{L}^{km}_{k} \otimes \mathrm{I}_{n}\right)\left(\mathrm{I}_{m} \otimes \left(\left(\mathrm{I}_{k} \otimes A^{n\times n}\right)\mathrm{L}^{kn}_{k}\right)\right)\left(\mathrm{L}^{mn}_{m} \otimes \mathrm{I}_{k}\right)}_{\mathrm{vec}(\nu)}, \quad \nu \mid k,n \tag{73}$$

$$\underbrace{\mathrm{L}^{mn}_{m}}_{\mathrm{mpi}(p)} \to \underbrace{\left(\mathrm{I}_{p} \otimes \mathrm{L}^{mn/p}_{m/p}\right)\left(\mathrm{L}^{p^2}_{p} \otimes \mathrm{I}_{mn/p^2}\right)\left(\mathrm{I}_{p} \otimes (\mathrm{L}^{n}_{p} \otimes \mathrm{I}_{m/p})\right)}_{\mathrm{mpi}(p)}, \quad p \mid m,n \tag{74}$$

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathrm{smp}(p,\mu)} \to \underbrace{\left(\mathrm{L}^{mp}_{m} \otimes \mathrm{I}_{n/p}\right)\left(\mathrm{I}_{p} \otimes (A_m \otimes \mathrm{I}_{n/p})\right)\left(\mathrm{L}^{mp}_{p} \otimes \mathrm{I}_{n/p}\right)}_{\mathrm{smp}(p,\mu)}, \quad \mu \mid n/p \tag{75}$$

Often, a tag is "pushed down" recursively

$$\underbrace{\mathrm{I}_{m} \otimes A_n}_{\mathrm{vec}(\nu)} \to \mathrm{I}_{m} \otimes \underbrace{A_n}_{\mathrm{vec}(\nu)} \tag{69}$$

which states that if the kernel A_n can be vectorized, then just loop over it. A shorthand notation for parallel tags is $\otimes_{\|}$ and for vector tags is $\vec{\otimes}$.

4) Architecture-Aware Tiling: Rules (61) and (62) are stride permutation factorizations that capture tilings of matrix transposition. They break the transposition into two passes: one pass that transposes blocks and one pass that transposes within blocks. To model the target platform, the size constraints of the architecture are imposed on the factorizations. For instance, the rule

$$\underbrace{\mathrm{L}^{n\nu}_{n}}_{\mathrm{vec}(\nu)} \to \left(\mathrm{I}_{n/\nu} \otimes \underbrace{\mathrm{L}^{\nu^2}_{\nu}}_{\mathrm{vec}(\nu)}\right)\left(\mathrm{L}^{n}_{n/\nu} \vec{\otimes} \mathrm{I}_{\nu}\right) \tag{70}$$

describes how stride permutations are performed efficiently on SIMD vector architectures such as Intel's SSE and AVX or the AltiVec and VSX extensions supported by PowerPC and POWER: it breaks the stride permutation into one stage that is performed solely with vector operations (captured by the tagged operation $\vec{\otimes}$) and one stage that performs small in-register transpositions (explained below).

More involved vectorization data flow transformations are captured by rules such as (71)–(73) in Table 4, which all ensure that data are accessed in SIMD vectors of size ν and the only intravector operation is $\mathrm{L}^{\nu^2}_{\nu}$, which will be handled below.

Parallelization for shared memory and message passing systems is captured by rules such as (74) and (75). Rule (74) factorizes a parallel transpose into local transposes and a big all-to-all communication with maximum packet size, while (75) performs a loop tiling transformation that ensures that packets of size μ are transferred between p processors, which avoids false sharing.

While the above discussion focused on SIMD vectors and parallelism on shared memory and message passing architectures, the underlying concepts can be generalized. Any hardware that requires coalesced access at a packet level is modeled similar to SIMD vectors, and any explicit or implicit messaging is modeled similar to the MPI rules. The focus in this section was on the stride permutation since it plays a central role in reshaping linearized high-dimensional data structures. However, other permutation families such as permutations that are linear on mixed-radix digits exhibit similar internal structures and can be handled analogously.

5) Architecture-Specific Templates: Rules such as (68) and (75) explain generically (for all operators A) how to vectorize or parallelize the respective data flow pattern. However, these rules may introduce irreducible operators such as $\mathrm{L}^{\nu^2}_{\nu}$ in case of (70) and $\mathrm{L}^{rs}_{r} \otimes \mathrm{I}_{u}$ in case of (74). SPIRAL contains a template library for such constructs that explains how to implement them for each supported platform. Specifically, for SIMD vector architectures, SPIRAL requires code fragments that implement

$$\mathrm{L}^{\nu^2}_{\nu}, \quad \mathrm{L}^{2\nu}_{2}, \quad \mathrm{L}^{2\nu}_{\nu}, \quad \mathrm{L}^{\nu^2/4}_{\nu/2} \otimes \mathrm{I}_2, \quad \mathrm{I}_{\nu/2} \otimes \begin{bmatrix} a & b \\ -b & a \end{bmatrix} \tag{76}$$

for vectors of primitive data types such as reals and integers. These constructs are parameterized by the vector length ν, abstract the details of the particular architecture, and provide support for data shuffling and complex arithmetic, e.g., required by T^{mn}_{n}. Code for (76) is stored in SPIRAL's internal representation (`icode`, discussed in Section IV-B) and inserted upon program generation. The code templates can be written by hand but also, in certain cases, be generated automatically from the ISA as explained below. Fig. 4 shows a code example for L^{16}_{4} implemented in Intel SSE 4-way `float`.

```
void L_16_4_SSE4x32f(__m128 *Y, __m128 *X) {
    __m128 t3, t2, t1, t0;
    t0 = _mm_unpacklo_ps(X[0], X[1]);
    t2 = _mm_unpacklo_ps(X[2], X[3]);
    t1 = _mm_unpackhi_ps(X[0], X[1]);
    t3 = _mm_unpackhi_ps(X[2], X[3]);
    Y[0] = _mm_movelh_ps(t0, t2);
    Y[1] = _mm_movehl_ps(t2, t0);
    Y[2] = _mm_movelh_ps(t1, t3);
    Y[3] = _mm_movehl_ps(t3, t1);
}
```

Fig. 4. *Code example for L_4^{16} implemented in Intel SSE 4-way `float`.*

The rule system ensures that as long as constructs (76) can be implemented efficiently, the whole data flow graph will be implemented efficiently. Sometimes algorithms require additional irreducible OL operators to be supported, and then users need to supply the necessary templates for the targeted architectures. Examples include the cyclic shift permutation that rotates vector components and the reduction operator that sums up all vector components. Once such a new operator template is provided, it is usable for any algorithm.

For distributed memory systems with message passing, the construct

$$P \otimes \mathrm{I}_\mu \tag{77}$$

for an arbitrary permutation P needs to be implemented as an explicit collective communication operation. The minimum is to provide an efficient implementation for

$$\underbrace{\mathrm{L}_p^{p^2} \otimes \mathrm{I}_\mu}_{\mathrm{mpi}(p)} \tag{78}$$

but other specialized permutations (that for instance capture MPI communicators) may be provided to obtain better performance. Similarly, FPGA implementations of streaming permutations or double buffering across multiple address spaces as well as computation off-loading requires the definition of irreducible operators and the supply of code templates for them [30], [43].

6) Automatic Derivation of Code Templates: For certain hardware structures or instruction sets, it can be hard to derive implementations for irreducible operators such as (76) by hand. For example, efficient implementations for SIMD vectorized blocks can be hard to derive by hand as vector lengths get longer and instruction set peculiarities prevent straightforward implementations. SPIRAL uses multiple approaches to automate the derivation of code snippets that implement (76). A matrix factorization-based search method expresses the semantics of SIMD shuffle operations as rectangular matrices and searches for short sequences that implement the targeted operations [26] by stacking and multiplying them. For AVX and Larrabee with 256- and 512-bit vectors, the search space becomes too large, and thus we developed a super-optimization-based approach [39].

For the efficient implementation of stride permutations (and related so-called linear permutations) on FPGAs we developed an automatic (and optimal) approach based on solving a particular matrix factorization problem [177], [178]. For general permutations, a method based on decomposing Latin squares is used [44].

IV. PROGRAM GENERATION

In this section, we discuss SPIRAL's approach to program generation and autotuning. First, we discuss the approach to 1) setting up and solving the constraint problem which produces structurally-optimized OL formulas for a given target platform; and 2) autotuning to select from the identified OL formulas. Next, we explain the DSL compilation process that translates the formulas into programs. Finally, we discuss correctness guarantees and formal correctness proofs.

A. Constraint Solving and Autotuning

Fig. 5 introduces the overall approach, depicted for the problem specification of DFT_8 and the hardware specification "AVX 2-way `_Complex double`" (complex arithmetic applied to 4-way `double`). Hardware rules (left, red), algorithm rules (right, blue), and program transformation rules (center, gray) together define a search space (multi-colored oval). The given problem specification and hardware target give rise to the solution space (black line), which is a subspace of the overall search space.

Each point in the solution space (black dot) represents a data flow graph given as a rule tree encoding the sequence

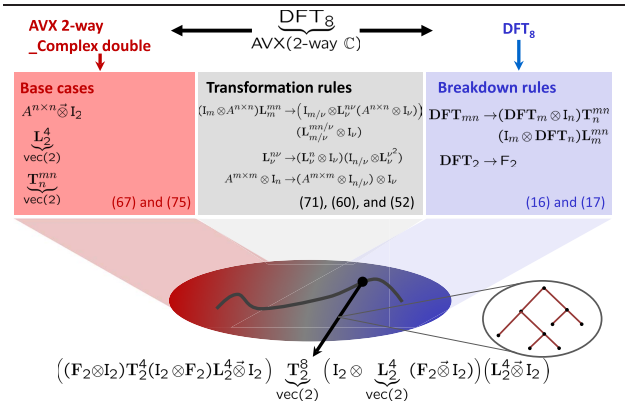


Fig. 5. ***Program generation as a constraint problem. Hardware rules (left, red), algorithm rules (right, blue), and program transformation rules (center, gray) together define a search space (oval). A given problem specification (DFT_8) and hardware constraint (Intel AVX in 4-way double precision mode implementing 2-way complex arithmetic) give rise to the solution space (black line). Each point in the solution space is represented as sequence of rule applications captured as rule tree that translates uniquely into a OL formula, which is compiled to efficient code by SPIRAL's rules-based DSL compiler. Walking the solution space enables autotuning.***

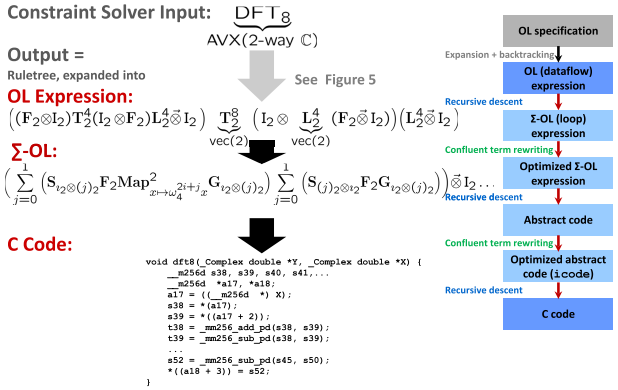


Fig. 6. ***SPIRAL's DSL compiler translates the output of the constraint solver (an OL expression) into high-performance C code through a multilevel rewriting system that intersperses multiple layers of recursive descent followed by confluent term rewriting.***

of rule applications, translated into an OL formula. The rule tree/OL formula represents an algorithmic solution for the specified problem and is optimized for the given target architecture. SPIRAL translates the OL formula via a multistage rewrite system (see Section IV-B and Fig. 6) into optimized code. Automatic performance tuning is implemented by measuring the performance of the candidate solutions so as to search for a fast solution. Note that solving the constraint problem dramatically reduces the search space (i.e., valid and optimized OL formulas for a desired problem specification) to already structurally-optimized candidates. This ensures that the generated code can be computed efficiently on the target architecture. Applying autotuning then factors in more complicated hardware/software interactions that cannot be easily modeled.

We now detail the components of our approach.

1) Hardware Space: As discussed in Section III-C, SPIRAL models hardware through the set of operators and OL formulas that can be mapped efficiently on it. Different aspects of a target platform (e.g., SIMD vector instructions, multiple cores, deep memory hierarchy, and accelerators) are modeled via a set of associated tags (one tag per hardware feature). Hardware features are modeled as constraints on breakdown rules.

This setup implicitly defines a space of programs that can be mapped efficiently on a given hardware platform, where the tags constrain the applicable breakdown rules so that only efficient expressions are enumerated. To enumerate this space of efficient programs, we start with the set of all (small) operators that we can directly implement through terminals, e.g., A_n in (67). We then build the expression on the right-hand side of the rule for all A_n, and apply the rule in reverse, deriving the untagged left-hand side from the right-hand side. In the case of (67), this would be the set

$$\{\mathrm{I}_p \otimes A_n : A_n \in (\mathbb{R}^n \to \mathbb{R}^n)\}.$$

The set of expressions now is substituted into all right-hand sides of all rules where it matches, e.g., A and B in (66), and the process is repeated. This procedure constructs the exponentially growing set of all efficient expressions given by a rule set, thus forming a space of parallel programs.

Formally, a rule set gives rise to a grammar of all efficient formulas, which can be expressed in Backus–Naur form (BNF). For instance, (66) and (67) restricted to tag $=\mathrm{mpi}(p)$ plus (78) give rise to the BNF for efficient MPI OL formulas $\langle \mathrm{mpiol} \rangle$

$$\langle \mathrm{mpiol} \rangle ::= \langle \mathrm{mpiol} \rangle \circ \langle \mathrm{mpiol} \rangle \mid \mathrm{L}_p^{p^2} \otimes \mathrm{I}_\mu \mid \mathrm{I}_p \otimes \langle \mathrm{uol} \rangle \tag{79}$$

$$\langle \mathrm{uol} \rangle ::= \text{any untagged OL formula} \tag{80}$$

parameterized by all untagged OL formulas $\langle \mathrm{uol} \rangle$. The BNF (79)–(80) guarantees that any formula

$$\underbrace{\langle \mathrm{mpiol} \rangle}_{\mathrm{mpi}(p)}$$

will be fully expanded by the rule set (66), (67), and (78). The parameter μ in (79) is a suitable MPI packet size.

In general, a target platform may have multiple hardware features (e.g., MPI and a SIMD ISA). In this case, the rule sets are combined to yield a smaller, more constrained space of OL formulas.

2) Algorithm Space: Similarly, for a set of algorithmic breakdown rules, all OL formulas that implement the encoded algorithms can be characterized through a BNF. For instance, the set of all multidimensional two-power FFTs computable via the Cooley–Tukey FFT algorithm and the row-column algorithm for higher-dimensional DFTs [rules (17)–(19)] is given by the BNF

$$\langle \mathrm{mdft} \rangle ::= \langle \mathrm{mdft} \rangle \otimes \langle \mathrm{mdft} \rangle \mid \langle \mathrm{dft} \rangle \otimes \langle \mathrm{mdft} \rangle \mid \langle \mathrm{mdft} \rangle \otimes \langle \mathrm{dft} \rangle \mid \langle \mathrm{dft} \rangle \otimes \langle \mathrm{dft} \rangle \tag{81}$$

$$\langle \mathrm{dft} \rangle ::= (\langle \mathrm{dft} \rangle \otimes \mathrm{I}_n)\mathrm{T}_n^{mn}(\mathrm{I}_m \otimes \langle \mathrm{dft} \rangle)\mathrm{L}_m^{mn} \mid \mathrm{F}_2. \tag{82}$$

Note that the size of the operands within an OL formula has to be compatible. This compatibility requirement also extends to the SPL/OL rules, where the size of the formulas on the right-hand sides has to match that of those on their respective left-hand sides.

As for the hardware formula space, the algorithm formula space is parameterized by the set of rewrite rules used to construct it. Different sets of rules can be used to break down a given specification and may restrict the supported problem sizes. For instance, a rule set to break down the DFT can consist of a number of combinations of the base rule (18) and recursive rules such as the Cooley–Tukey FFT (17), the Good Thomas Prime Factor Algorithm, Rader's Algorithm, Bluestein's Algorithm,

Partial Diagonalization, etc. [1], [17]. The FFT rule set can be a subset of a rule set for the SAR algorithm (see Table 2) or convolution. Complex algorithms require the union of multiple rule sets, and care needs to be taken to ensure the rules are compatible.

3) Program Transformations: Rules that encode program transformations or data layout transformations [e.g., (52), (57)–(62), and (71)–(75) in Table 4] define alternative but equivalent data flows for OL formulas, and thus alternative algorithms. Thus, when enumerating all OL formulas given by a rule set, after expanding formulas using a rule, the set of all equivalent data flows needs to be constructed. This is done by applying the program transformation and data layout transformation rules. For instance, using (55)–(58) in (82) and the fact that $\mathrm{DFT}_n = \mathrm{DFT}_n^T$, the BNF construct

$$(\langle \mathrm{dft} \rangle \otimes \mathrm{I}_n)\mathrm{T}_n^{mn}(\mathrm{I}_m \otimes \langle \mathrm{dft} \rangle)\mathrm{L}_m^{mn}$$

is expanded into a set of four equivalent formulas

$$\{(\langle \mathrm{dft} \rangle \otimes \mathrm{I}_n)\mathrm{T}_n^{mn}(\mathrm{I}_m \otimes \langle \mathrm{dft} \rangle)\mathrm{L}_m^{mn} \tag{83}$$

$$\mathrm{L}_m^{mn}(\mathrm{I}_n \otimes \langle \mathrm{dft} \rangle)\mathrm{L}_n^{mn}\mathrm{T}_n^{mn}(\mathrm{I}_m \otimes \langle \mathrm{dft} \rangle)\mathrm{L}_m^{mn} \tag{84}$$

$$(\langle \mathrm{dft} \rangle \otimes \mathrm{I}_n)\mathrm{T}_n^{mn}\mathrm{L}_m^{mn}(\langle \mathrm{dft} \rangle \otimes \mathrm{I}_m) \tag{85}$$

$$\mathrm{L}_m^{mn}(\mathrm{I}_n \otimes \langle \mathrm{dft} \rangle)\mathrm{T}_n^{mn}(\langle \mathrm{dft} \rangle \otimes \mathrm{I}_m)\}. \tag{86}$$

This expresses the four variants of the general radix Cooley–Tukey FFT algorithm: decimation-in-time (DIT) (83), decimation-in-frequency (DIF) (86), four-step (85), and six-step (84) as equivalency relation on OL formulas.

4) Characterizing the Solution Space: Now, we have formally defined a space of programs running efficiently on a given target platform as the set of all OL formulas that can be constructed from a properly chosen rule set, and we have defined the associated BNF. Furthermore, we have defined a space of OL formulas that implements a given problem specification via a properly chosen rule set, and again we have defined the associated BNF. Thus, we can state that all programs that are solving the given specification and run efficiently on the given target hardware are characterized by the intersection of the two spaces. For instance, all 1-D DFTs that can be implemented well using MPI on p processors are given by the intersection of the set of all formulas given by the MPI BNF (79)–(80) with all formulas implementing a multidimensional DFT via BNF (81)–(82) and alternative algorithms due to data flow transformations and the DFT transposition property (83)–(86)

$$\{\langle \text{mpi-mdft} \rangle\} = \{\langle \text{mpiol} \rangle\} \cap \{\langle \text{mdft} \rangle\}. \tag{87}$$

While the architecture formula space ($\{\langle \text{mpiol} \rangle\}$) and the implementation formula space ($\{\langle \text{mdft} \rangle\}$) are very large (exponentially growing with problem size, and possibly infinite even for small problem sizes if no restrictions are placed on formulas), the intersected space is usually not too large and for all practical problems tractable.

5) Traversing the Solution Space: While (87) abstractly characterizes the space of all solutions, it does not allow us to construct solutions efficiently. However, any point in the solution space is reachable by a sequence of rule applications, starting with the initial specification, and ending when no rule is applicable any longer. Algorithm and data layout transformation rules expand the space, but only hardware rules can terminate the expansion. This construction implicitly intersects the algorithm space with the hardware space and implies that backtracking and constraint solving needs to be applied to find feasible solutions since not all expansions lead to fully expanded (terminated) OL formulas. The sequence of recursive rule application leading to a solution is encoded as rule tree, and every rule tree can uniquely be translated into a corresponding OL formula (see Fig. 2); the converse is not true. Care needs to be taken to ensure that the rule system contains no cycles leading to infinite loops when trying to expand a specification. Currently, developers need to design rule systems that ensure termination. Automatic checking is left to future work.

There are a number of methods to traverse or explore the search space that control the order of rule expansion and candidate generation [1], [67], [179]: 1) full enumeration is achieved by recursively applying all rules and parameterizations at each expansion; 2) random expansion picks a random rule and random parameterization at each expansion; 3) dynamic programming picks the "best" rule and parameterization according to some metric at each expansion; and 4) genetic search maintains a population of expansions and performs mutation and cross breeding to derive a new population. While these methods originally were defined for recursive search spaces, SPIRAL uses them as constraint solvers by enforcing backtracking if an expansion cannot terminate.

6) Controlling the Solution Space: The search space implicitly defines a binary metric for OL formulas to be optimized for a target platform: Either a formula is in the search space (then it can be implemented efficiently), or it is not in the search space (then it cannot be implemented efficiently on the hardware with respect to our definition of efficiency). The metric can be made more gradual by assigning a fitness score between 0 and 1 and by defining a partial order of rule sets that capture various aspects and optimization methods for a target platform. For instance, the "high efficiency" rule set only allows for a limited number of very efficient constructs while a "lower efficiency" rule set allows for more and less efficient constructs. We then find the most stringent rule set that allows us to find solutions for a given specification and target architecture, which is equivalent to maximizing the fitness score of a formula.

7) Automatic Performance Tuning: The framework developed in this section allows us to traverse the solution space and evaluate each point of the solution space with a number of metrics (runtime performance being the most important). This can be used to perform automatic performance tuning (also called autotuning): Any of the previously discussed traversal/search algorithms produces candidate solutions in the search space, which by construction are efficient on the target platform. Then, the candidates are measured (usually runtime) to provide a numeric score. The score is used to guide the search algorithm. Since we have a partial order of search spaces and a number of metrics at various code representation levels, faster (and less accurate) methods can be used to quickly find good candidates, and then more aggressive search can be used locally to find the very best solutions. In practice, we often use a combination of genetic search, dynamic programming, line search, and exhaustive search, partitioning the parameters into system parameters that are optimized once and specifying parameters that are optimized for each specification.

SPIRAL allows defining arbitrary performance metrics as long as it can evaluate arbitrary OL formulas to a scalar. Example metrics we have used include runtime and power, energy and energy-delay product, static analysis outputs such as instruction count or register pressure, and profiling information and performance monitor counters. Metrics can be used at any abstraction level in SPIRAL: some analyze OL formula properties while other metrics analyze intermediate representations or profile the code (discussed in Section IV-B).

B. DSL Compiler

In Section IV-A, we discussed SPIRAL's approach to derive OL formulas that represent efficient programs for a given specification on a given target architecture. These OL formulas can be viewed as flat or structured data flow graphs. Structure is enforced by parentheses that require particular evaluation order while associative operators allow arbitrary evaluation order in the absence of parentheses. In this section we explain how SPIRAL translates a data flow graph that guarantees an efficient implementation relative to the simple hardware model into the actual efficient implementation on the target machine. This is done via a multilevel rewrite system and a stack of DSLs. The rule sets of the rewrite system are configurable and the DSLs are extensible to enable encoding of human performance engineering knowledge.

1) Overall Structure: Fig. 6 shows the overall structure of SPIRAL's DSL compiler. It translates the output of the constraint solver (an OL expression) into high-performance C code through a multilevel rewriting system that intersperses multiple layers of recursive descent followed by confluent term rewriting. The distinct abstraction layers are discussed as follows:

- rule trees represent the solution of the constraint problem given by problem specification and target architecture as a sequence of rule applications (see Section IV-A);
- OL formulas represent the data flow graphs derived from applying the rules as prescribed by the rule tree;
- Σ -OL formulas make loop iterations and data gathering/scattering explicit;
- icode is an internal abstract code representation that is restricted and can be mapped to C or Verilog.

A rule tree and thus its associated OL formula is compiled to code by applying two types of rule sets at every abstraction level: first recursive descent is used to translate the higher-level abstraction to the next-lower level abstraction by recursively walking the tree and locally translating the constructs via context-free rewrite rules. Second, the translated expression is reduced to normal form via a confluent term rewriting step. Individual rule sets can be complex sequences of merged basic rule sets, and developers need to exercise caution to ensure confluence and avoid infinite loops. SPIRAL uses both mathematical data types ($\mathbb{R}$, $\mathbb{C}$, intervals over reals, Galois fields, etc.) and machine data types (`double`, `float`, `int`, `__m128d`, etc.), and rewrite rules are used to lower from mathematical to machine data types.

Throughout the translation process, the lower level representations retain the properties ensured by the higher level abstractions, such as locality and being implementable by certain instruction sets. It would be hard to prove these properties directly on the lower-level abstractions, but the properties hold by construction. Σ-OL expressions are folded data flow graphs that represent operator expressions constructed with iterative higher-order operators that capture loops, folds, and maps. `icode` programs that are derived from OL expressions can be viewed as C programs, abstract syntax trees, pure functional programs, lambda functions, or mathematical operators.

Finally, we present here the rewrite process as a procedure consisting of well-separated stages. However, as implemented by SPIRAL the stages may operate asynchronously on different parts of the algorithm: some subexpressions may already have been lowered to Σ-OL or `icode` while other subexpressions are still unexpanded nonterminals, rule trees, or OL formulas. This flexibility allows SPIRAL to cache partial solutions or store partially generated code that can be retargeted to a smaller set of target architectures with less code synthesis time. In addition, rule trees have a unique translation to OL expressions and thus are used above synonymously.

We now discuss the Σ-OL and `icode` abstraction levels and the associated rewrite systems.

2) Σ-OL: In Section IV-A, data flow optimizations are encoded as rewrite rules and together with algorithmic breakdown rules allow SPIRAL to expand a specification into an optimized data flow graph represented as an OL formula. The OL formula encodes dependencies (encoded

by $\circ$) and execution order (encoded by nested parenthesis). It uses iterators such as Map, Fold, and $\otimes$ to encode repetitions in the data flow graph. We now explain how these iterators are lowered to a representation that makes iterations and iteration order explicit.

This is achieved in the next layer in SPIRAL's formal system, using a DSL called Σ-OL, which provides a representation that is closer to the final imperative description of the computation. Iterators are "broken" into a representation that makes loops explicit and enables loop-level optimization such as loop merging, software pipelining, and loop iteration reordering. Σ-OL is a superset of OL that introduces additional constructs. All optimizations on Σ-OL are again expressed as rewriting rules. However, there is one major difference between OL rewriting and Σ-OL rewriting: OL rewriting requires only relatively simpler pattern matching but needs to handle alternatives and support search. Σ-OL rewriting requires powerful pattern matching but no search. SPIRAL's Σ-OL rewriting system, which is described next, has an elaborate mechanism to manage, combine, and stage rule sets.

To introduce Σ-OL, we first consider $\mathrm{Map}^n_{f_i(.)}$. It applies the function $f_i(.)$ to all input vector components to produce its output vector. We define the n-dimensional standard basis vector with a one at entry i and zero everywhere else as

$$\mathrm{e}^n_i = (\delta_{ij})_{j=0,\ldots,n-1} \in \mathbb{R}^n, \quad \delta_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j. \end{cases} \tag{88}$$

In (88), δ_{ij} is the Kronecker delta. We view (88) as a linear operator (and thus an OL operator) that is multiplied to an input vector from the left to extract its ith component. Now we can decompose (as a rewriting rule)

$$\mathrm{Map}^n_{f_i(.)} \to \sum_{i=0}^{n-1} \mathrm{e}^n_i(.) \circ f_i(.) \circ \left(\mathrm{e}^n_i\right)^T(.). \tag{89}$$

Equation (89) captures the fact that $\mathrm{Map}^n_{f_i(.)}$ can be implemented as a loop that applies a function to each entry of the input vector by translating $\mathrm{Map}^n_{f_i(.)}$ into an explicit loop that extracts vector element x_i, computes $f_i(x_i)$, and then stores the result at location i of the output vector. Map can be easily expressed as iterative sum since all operations are independent.

The reduction operation described by $\mathrm{Fold}^n_{f_i(.,.),z}$ requires a bit more formalism. As an example, we decompose

$$\mathrm{Fold}^n_{(x,y)\to x\sqcup y,z} \to \bigsqcup_{i=0}^{n-1} \left(\mathrm{e}^n_i\right)^T(.) \tag{90}$$

for an arbitrary associative binary operator $\sqcup$ that has a neutral element z (returned for $n = 0$) and an iterative version $\bigsqcup_{i=0}^{n-1}$. Practically relevant reduction operators $\sqcup$ are min, max, $\cdot$, or $+$. As in (89), $\left(\mathrm{e}^n_i\right)^T(.)$ in (90) extracts the element x_i from the input vector x, and the iterative operator reduces all elements x_i by its definition

$$\bigsqcup_{i=0}^{n-1} x_i = x_0 \sqcup x_i \sqcup \ldots \sqcup x_{n-1}. \tag{91}$$

3) Tensor Products and $\boldsymbol{\Sigma}$ -OL: The tensor product $\mathrm{I}_m \otimes A_n$ and the direct sum $\bigoplus_{i=0}^{n-1} A^{(i)}_n$ are generalizations of $\mathrm{Map}^n_{f_i(.)}$. To lower tensor products to Σ-OL, we first define a gather and scatter operator

$$\mathrm{G}_f = \sum_{i=0}^{n-1} \mathrm{e}^n_i \left(\mathrm{e}^N_{f(i)}\right)^T \tag{92}$$

$$\mathrm{S}_f = \sum_{i=0}^{n-1} \left(\mathrm{e}^N_{f(i)}\right)^T \mathrm{e}^n_i \tag{93}$$

parameterized by an index mapping function

$$f : \mathbb{I}_n \to \mathbb{I}_N$$

where $\mathbb{I}_k = \{0, 1, \ldots, k-1\}$ is the integer interval from 0 to $k-1$. We define basic index mapping functions

$$\imath_n : \mathbb{I}_n \to \mathbb{I}_n; i \mapsto i \tag{94}$$

$$(j)_n : \mathbb{I}_1 \to \mathbb{I}_n; i \mapsto j \tag{95}$$

$$\ell^{mn}_m : \mathbb{I}_{mn} \to \mathbb{I}_{mn}; i \mapsto \left\lfloor \tfrac{i}{n} \right\rfloor + m(i \bmod m) \tag{96}$$

that have the properties

$$\mathrm{G}_{\imath_n} = \mathrm{I}_n \tag{97}$$

$$\mathrm{S}_{(j)_n} = \mathrm{e}^n_j \tag{98}$$

$$\mathrm{G}_{\ell^{mn}_m} = \mathrm{L}^{mn}_m. \tag{99}$$

Finally, we define the tensor product of index mapping functions for functions $f : \mathbb{I}_m \to \mathbb{I}_M$ and $g : \mathbb{I}_n \to \mathbb{I}_N$

$$f \otimes g : \mathbb{I}_{mn} \to \mathbb{I}_{MN} : i \mapsto Nf\left(\left\lfloor \tfrac{i}{n} \right\rfloor\right) + g(i \bmod n). \tag{100}$$

The definitions give rise to the key compatibility condition

$$\mathrm{G}_{f\otimes g} = \mathrm{G}_f \otimes \mathrm{G}_g \tag{101}$$

$$\mathrm{G}_{f\times g} = \mathrm{G}_f \times \mathrm{G}_g \tag{102}$$

$$\mathrm{G}_{f\circ g} = \mathrm{G}_g \mathrm{G}_f \tag{103}$$

and the identity $\mathrm{S}_f = \mathrm{G}^T_f$. We now can translate tensor products to Σ-OL expressions using

$$\mathrm{I}_m \otimes A_n \to \sum_{j=0}^{m-1} \mathrm{S}_{(j)_m \otimes \imath_n} A_n \mathrm{G}_{(j)_m \otimes \imath_n} \tag{104}$$

$$A_m \otimes \mathrm{I}_n \to \sum_{j=0}^{n-1} \mathrm{S}_{\imath_n \otimes (j)_m} A_m \mathrm{G}_{\imath_n \otimes (j)_m}. \tag{105}$$

As in Section III-A, generalizations to multi-arity operators exist but are unwieldy.

4) Tagged Σ-OL: Tags used for OL operations and objects are also supported at the Σ-OL level to carry hardware information down to the program generation system. For instance

$$\mathrm{G}_f \vec{\otimes} \mathrm{I}_\nu$$

will not be broken down further as it will be implemented using vector operations. On Intel's SSE, for example, it will be implemented eventually with `_mm_movaps` as a permutation/gather of SIMD vectors of type `__m128`. Similarly, tagged parallel tensor products become tagged parallel iterative sums that are eventually implemented using OpenMP or MPI. For FPGAs, software pipelining is an annotation that tags both OL and Σ-OL formulas appropriately. The tagging does not interfere with the rewriting rules we present. It may be explicitly utilized by rules and carries hardware information through the system.

5) Loop Merging: So far we have described the recursive descent approach and rule set that converts OL formulas into Σ-OL formulas. Next we discuss how these formulas are optimized. OL constructs such as Map, Fold, and tensor products imply a traversal through their input data set. Thus, merging rules such as

$$\mathrm{Map}^n_f \circ \mathrm{Map}^n_g \to \mathrm{Map}^n_{f\circ g} \tag{106}$$
$$\mathrm{Fold}^n_{f,z} \circ \mathrm{Map}^n_g \to \mathrm{Fold}^n_{(x,y)\mapsto f(g(x),y),z} \tag{107}$$

can be seen as loop merging rules. For instance, (107) is used to optimize the expression derived from (41)–(42) for the Chebyshev distance (40), and leads to the transformation sequence

$$d_\infty(u,v) \to \mathrm{Fold}^n_{(x,y)\mapsto \max(x,y),0} \circ \mathrm{Map}^{n\times n\to n}_{(x,y)\mapsto |x-y|} \tag{108}$$
$$\to \mathrm{Fold}^{n\times n\to n}_{((x_1,x_2),y)\mapsto \max(|x_1-x_2|,y),0}. \tag{109}$$

Similarly, rules for G_f and S_f exist and optimize away copy/reorder loops

$$\mathrm{G}_f \circ \mathrm{Map}_g \to \mathrm{Map}_{g\circ f} \circ \mathrm{G}_f \tag{110}$$
$$\mathrm{G}_f \circ \mathrm{G}_g \to \mathrm{G}_{g\circ f}. \tag{111}$$

For (106)–(111) to be able to simplify the Σ-OL formulas fully, rules

$$\left(\sum_{i=0}^{n-1} A_i\right) \circ B \to \sum_{i=0}^{n-1} A_i \circ B, \quad C \circ \left(\sum_{i=0}^{n-1} A_i\right) \to \sum_{i=0}^{n-1} C \circ A_i$$

with $B \in \{\mathrm{G}, \mathrm{Map}\}$ and $C \in \{\mathrm{S}, \mathrm{Map}\}$ are needed to propagate the gather, scatter, and map operations into the inner sum/loop. Generalizations for reductions/Fold also exist.

6) Indexing Simplification: The final set of rewrite rules that is needed simplifies compositions of index mapping functions and encodes complex integer equalities. Important examples [17] are

$$\ell^{mn}_m \circ \left((j)_m \otimes \imath_n\right) \to \imath_n \otimes (j)_m \tag{112}$$
$$\left((j)_m \otimes \imath_n\right) \circ f_n \to (j)_m \otimes f_n \tag{113}$$
$$\left(\imath_n \otimes (j)_m\right) \circ f_n \to f_n \otimes (j)_m. \tag{114}$$

These identities mirror OL identities such as (55) at the Σ-OL level. While tensor products can be broken into iterative direct sums or sums of gather/compute/scatter kernels, stride permutations and their identities cannot easily be handled by decomposition into sums alone. The gather/scatter rewriting rules together with the index simplification rules address this problem and allow to merge stride permutations as indexing patterns into gather/compute/scatter loops. They are used to simplify the Σ-OL expressions such as

$$(\mathrm{I}_m \otimes A_n)\mathrm{L}^{mn}_m \to \sum_{j=0}^{m-1} \mathrm{S}_{(j)_m \otimes \imath_n} A_n \mathrm{G}_{\imath_n \otimes (j)_m}. \tag{115}$$

The gather and scatter functions in (115) capture the addressing pattern due to the loop implementing the tensor product and translate the data reorganization captured by the stride permutation into reindexing in the gather operation. The index simplification rule set allows this simplification to be performed across multiple recursion levels.

We now apply the Σ-OL optimization stage to our example, DFT_8. Applying the full Σ-OL rule system to (20) yields (116) shown in Fig. 7. Note that the resulting expression is an imperfectly nested sum and all summands are normalized to the shape

$$\mathrm{S}_f \circ A \circ \mathrm{G}_g$$

where A, f, and g are parameterized by loop variables. The gather and scatter functions capture all reindexing due to multiple levels of stride permutations, and all twiddle factor scaling is pulled into the inner-most loop. This is an instance of the normal form that SPIRAL achieves for well-optimized OL formulas: the Σ-OL rewrite succeeds because the OL level rewrites guarantee it by manual construction. Future work will address theoretical guarantees and automation for rule set management.

7) Termination: A final intermediate step before translating Σ-OL to `icode` is the termination of all remaining gather, scatter, map, and fold using (89)–(93). The resulting expressions are composed of the minimal set

$$\left(\sum_{j=0}^{3} \mathrm{S}_{\imath_2\otimes(j)_4}\mathrm{F}_2\,\mathrm{Map}^2_{x\mapsto\omega_8^{4i+j}x}\,\mathrm{G}_{\imath_2\otimes(j)_4}\right)\left(\sum_{j=0}^{1}\left(\sum_{i=0}^{1}\mathrm{S}_{(j)_2\otimes\imath_2\otimes(i)_2}\mathrm{F}_2\,\mathrm{Map}^2_{x\mapsto\omega_4^{i+2j}x}\,\mathrm{G}_{\imath_2\otimes(i)_2}\right)\left(\sum_{i=0}^{1}\mathrm{S}_{(i)_2\otimes\imath_2}\mathrm{F}_2\mathrm{G}_{\imath_2\otimes(i)_2\otimes(j)_2}\right)\right) \tag{116}$$

Fig. 7. *Fully expanded and optimized $\sum$-OL expression for DFT_8 as decomposed in (20). ω_n is the primitive nth root of unity, used to compute the twiddle factors. All factors in (116) are linear and thus matrices; therefore, we drop the $\circ$ operator.*

of Σ-OL operators and operations needed to express OL expressions: standard basis vectors, scalar n-ary operators, and iterative operations (sum/union and reductions). An example is the final expression for (40) derived by terminating (109)

$$\max_{i=0,\ldots,n-1}\left((x,y)\mapsto|x-y|\right)\circ\left(\left(\mathrm{e}_i^n\right)^T\times\left(\mathrm{e}_i^n\right)^T\right). \tag{117}$$

This enables the design of a small rule-based compiler for the whole Σ-OL language, which we discuss next.

C. Backend Compiler

We now discuss the lowest abstraction level in SPIRAL, the internal code representation called `icode`, the translation form Σ-OL to `icode`, `icode` optimization, and unparsing (pretty printing).

1) Code Representation: SPIRAL utilizes a simple iterative language to capture C code and language extensions. The language represents 1) values and data types; 2) arithmetic and logic operations; 3) constants, arrays and scalar variables; and 4) assignments and control flow. While the language is very general, the code generated inside the SPIRAL system has strong properties. An `icode` program that was derived by SPIRAL from an operator specification is at the same time 1) a program given as abstract syntax tree; 2) a program in a restricted subset of C; 3) an OL operator over mathematical numbers and machine numbers; 4) a pure functional program; and 5) a lambda expression.

icode is arbitrarily extensible to represent instruction set extensions such as Intel's SSE and AVX and the necessary data types. Instructions are modeled as functions and a compiler that implements an intrinsic function interface and the necessary data types are assumed. icode represents C pragmas needed for OpenMP, OpenACC and similar systems, as well as library calls to math and communication libraries. Instruction set-specific strength reduction is implemented through SPIRAL's rewriting system. We have implemented higher-level data types and the respective operations such as complex arithmetic, interval arithmetic, Galois fields, and soft floating-point and fixed-point arithmetic.

2) Program Generation: Translation from terminated Σ-OL to `icode` is done via a recursive descent pass. Applied to the terminated normal form of Σ-OL, only a few rules are needed for program generation. Fig. 8 shows the simplified rule set. We use an operator $\mathrm{Code}(.)$ to invoke the compilation process, and provide a rule set to recursively translate Σ-OL expressions to `icode`. The translation rules need to cover the translation of 1) operator composition into a sequence of operator implementations (118); 2) loop implementation of iterative operations (119); 3) basis vector operations into reads and assignments of array elements (120)–(121); and 4) scalar operator evaluation into their `icode` implementation (122).

The full compilation rule system used by SPIRAL is highly configurable and much more complex. It handles code generation for tagged operations and irreducible OL operators into specially tagged `icode` instructions. For instance, a tagged parallel iterative sum is translated into a tagged parallel for loop. Users can supply templates for OL constructs that require implementation "tricks," and multiple implementations per construct can be provided. For optimization, we may not terminate constructs such as gather, scatter, or small matrices but translate these Σ-OL constructs directly to `icode`.

3) Backend Optimization: After translation to `icode`, SPIRAL's backend compiler optimizes the generated code and transforms it into the correct shape for the target (C/Fortran/Verilog) compiler. The core is a powerful and highly configurable basic block compiler. No loop-level optimization is required, since all higher level optimizations such as parallelization, vectorization, or streaming have been lifted to the OL and Σ-OL level. The main basic block optimizations are [13]: 1) loop unrolling; 2) array scalarization; 3) constant folding; 4) copy propagation; and 5) common subexpression elimination.

In addition, strong support for architecture-specific strength reduction is provided to enable backend optimizations for a wide range of complex instruction sets with

$$\mathrm{Code}\left(y=(A\circ B)(x)\right)\rightarrow\{\texttt{decl}(t)\ \mathrm{Code}\left(t=B(x)\right)\ \mathrm{Code}\left(y=A(t)\right)\} \tag{118}$$

$$\mathrm{Code}\left(y=\left(\bigsqcup_{i=0}^{n-1}A_i\right)(x)\right)\rightarrow\{y:=\vec{z}_{\sqcup}\ \texttt{for}(i=0..n-1)\ \mathrm{Code}\left(y_{\sqcup}=A_i(x)\right)\} \tag{119}$$

$$\mathrm{Code}\left(y=\left(\mathrm{e}_i^n\right)^T(x)\right)\rightarrow y[0]:=x[i] \tag{120}$$

$$\mathrm{Code}\left(y=\mathrm{e}_i^n(x)\right)\rightarrow\{y=\vec{0},y[i]:=x[0]\} \tag{121}$$

$$\mathrm{Code}\left(y=f(x)\right)\rightarrow y[0]:=\mathrm{Code}(f)(x[0]) \tag{122}$$

Fig. 8. *Rule-based translation from terminated Σ-OL to `icode`. $\vec{z}_{\sqcup}$ is the zero vector relative to the operator $\sqcup$.*

```
opts := CopyFields(
    SpiralDefaults,
    IntelC99Mixin,
    breakdownRules := rec(
        DFT := [ DFT_CT, DFT_Base ] ));
rt := RandomRuleTree(rt, opts);
c := CodeRuleTree(rt, opts);
PrintCode("dft8", c, opts);
```

Fig. 9. Spiral *script to generate C using C99 complex arithmetic (Intel C++ compiler syntax) for* DFT_8 *using breakdown rules (17) and (18).*

idiosyncratic behaviors. The compiler also implements an extensible type system that supports both abstract types and machine types, type unification, and type-specific optimization. The production compiler is implemented in GAP using both a complex rule system and procedural components. A high assurance branch developed for correctness guarantees [65] is implemented solely using rewriting rules.

The same `icode` language is used for C compilers, C-derived dialects such as OpenCL and CUDA, and for combinational logic (Verilog). The SPIRAL backend compiler can be configured to utilize various equivalent C idioms (e.g., `X[i` versus `*(X+i))` and optimizations such as index recomputation versus induction to best match the capabilities of the target C compiler. This is made necessary by the high variance in the capabilities of C compilers we have observed, in particular when targeting experimental hardware and early prototypes. The SPIRAL backend compiler also is a valuable standalone tool [180] as it can be used as an interactive and scriptable backend code generation system.

4) Unparsing: The final `icode` needs to be unparsed (pretty-printed) for the target compiler. Again, the SPIRAL unparsing system is highly configurable to be easily adaptable to target compiler idiosyncracies. There is a large variation across compilers regarding pragmas, hardware-specific language extensions, implementation level of various C dialects (ANSI C, C99, etc), support for C++ idioms in C, and built-in functions. Traditionally, these variations are handled through the C preprocessor. However, since SPIRAL has a fully programmable unparsing system, most of these issues are handled in the unparser, and C preprocessor macros are used sparingly, e.g., for productivity and experimental reasons.

As an example, see the code generated from (116) by the SPIRAL script shown in Fig. 9, shown in Fig. 10. The script specifies the kernel DFT_8, the breakdown rules (17) and (18), and C99 complex arithmetic (Intel C++ compiler syntax). The final C code for (117) is shown in Fig. 11. SPIRAL implements a high-performance interval arithmetic backend that uses Intel's SSE2 two-way double precision vectors to encode intervals. The infimum is stored with changed sign, and the rounding mode is set to round-to-infinity to avoid frequent reconfiguration of the FPU.

```
_Complex double *D3, *D4;

void dft8(_Complex double *Y,
          _Complex double *X) {
    static _Complex double T1[8], T2[4];
    _Complex double s5, s6, s7, s8, s3, s4;
    for(int i5=0; i5<=1; i5++) {
        for(int i9=0; i9<=1; i9++) {
            s5 = X[2*i5+i9];
            s6 = X[2*i5+i9+4];
            T2[2*i9] = s5+s6;
            T2[2*i9+1] = s5-s6;
        }
        for(int i8=0; i8<=1; i8++) {
            s7 = D4[i8]*T2[i8];
            s8 = D4[i8+2]*T2[i8+2];
            T1[4*i5+i8] = s7+s8;
            T1[4*i5+i8+2] = s7-s8;
        }
    }
    for(int i4 = 0; i4 <= 3; i4++) {
        s3 = D3[i4]*T1[i4];
        s4 = D3[i4+4]*T1[i4+4];
        Y[i4] = (s3+s4);
        Y[i4+4] = (s3-s4);
    }
}
```

Fig. 10. *Final C code for* DFT_8 *as expanded in (116), unparsed using C99 complex arithmetic.* $D3$ *and* $D4$ *have been initialized with the proper complex values of* ω_i^n.

Denormal floating-point numbers are treated as zeroes as usual in the Intel SSE high-performance mode [65]. We show the resulting code in Fig. 12.

5) Targeting FPGAs: When targeting FPGAs, SPIRAL can be configured in two ways. In the first approach, an SPL/OL program representing the data flow is outputted together with basic block definitions using `icode` unparsed in Verilog. Then, a separate Java-based tool chain implemented outside SPIRAL is responsible for implementing the preshaped data flow graphs efficiently in Verilog [41]–[45], [179]–[181].

In the second approach, currently under development [184], `icode` is unparsed for high-level synthesis using the OpenCL language or C/C++ annotations, e.g., as supported by Vivado HLS. In this case, SPIRAL performs lower-level optimizations using the standard tool pipeline including the Σ-OL and `icode` abstraction layers and then relies on third-party backend tools for the final implementation on the FPGA. While in this case the FPGA design generation is using the same infrastructure as soft-

```
double cheb_dist(double *x, double *y, int n) {
    double r=0.0;
    for(int i=0; i<n; i++)
        r = max(abs(x[i]-y[i]), r);
    return r;
}
```

Fig. 11. *Final C code for* $d_\infty^n(.,.)$ *as expanded in (117).*

```
#include <smmintrin.h>
#include <float.h>
#define EPS (DBL_MIN+DBL_MIN)

__m128d cheb_dist(double *x, double *y, int n) {
  __m128d r, xi, yi, nxx, mx, mn, absi;
  // rounding mode: round to infinity
  unsigned _xm = _mm_getcsr();
  _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);
  // initialize r=[0.0, 0.0]
  r = _mm_set1_pd(0.0);
  for(int i=0; i<n; i++) {
    // xi = roundup([-(x[i]-EPS), x[i]+EPS])
    xi = _mm_addsub_pd(_mm_set1_pd(EPS),
         _mm_set1_pd (x[i]));
    // yi = roundup([-(y[i]-EPS), y[i]+EPS]
    yi = _mm_addsub_pd(_mm_set1_pd(EPS),
         _mm_set1_pd (y[i]));
    // xi_sub_yi = roundup(xi - yi)
    xi_sub_yi = _mm_add_pd(x,
      _mm_shuffle_pd(y, y, _MM_SHUFFLE2(0,1)));
    // absi = roundup(abs(xi_sub_yi));
    nxx = _mm_shuffle_pd(xx, xx, _MM_SHUFFLE2(0,1));
    mx = _mm_max_pd(xx, nxx);
    mn = _mm_min_pd(xx, nxx);
    absi = _mm_shuffle_pd(mn, mx,
             _MM_SHUFFLE2(1,0));
    // r = roundup(max(absi, r))
    r = _mm_shuffle_pd(_mm_min_pd(absi, r),
        _mm_max_pd(absi, r), _MM_SHUFFLE2(1,0));
  }
  // restore rounding mode
  _mm_setcsr(_xm);
  // infinum supremum sign
  r = _mm_xor_pd(_mm_set_pd(-0.0, 0.0), r);
  // return interval containing d_inf(x[],y[])
  return r;
}
```

Fig. 12. ***Final C SSE 4.1 code for $d_\infty^n(.,.)$ as expanded in (117), implemented using interval arithmetic [65].***

ware generation, the rules and hardware abstractions are FPGA-specific instances and `icode` is interpreted as a state machine, not as a sequential or parallel program.

6) Profiling: The autotuning component of SPIRAL requires profiling of the generated code. The standard approach is to time the generated code on the target architecture and use the execution time as a metric. However, more general profiling may be required. Across SPIRAL related projects, we have used measured power and energy, accuracy of the computation, code size, statistics derived from the generated assembly, and many other metrics. This is enabled by SPIRAL's configurable and extensible profiling system.

7) Optimization Performance: The offline optimization time for SPIRAL to generate code can range from less than a second for small code fragments to a day or more of search. The tuning time depends on the size and type of the problem, the performance of the backend compiler, potential job submission systems, and the desired optimization and autotuning level. The implementation on SPIRAL in the computer algebra system GAP is not optimized but favors productivity over performance.

8) Correctness: The SPIRAL system provides strong correctness guarantees for the generated code. For linear and multilinear operators, any internal representation can be converted to the equivalent matrix that specifies the operation, and the matrices can be compared. SPIRAL supports exact arithmetic for a range of special cases that cover signal processing algorithms. For problem sizes for which the matrix representation is too large to be constructed, the matrix can be constructed by evaluating the program representation for all basis vectors. Finally, probabilistic checks that evaluate multiple random algorithms on random inputs can be used.

For nonlinear operators, establishing correctness is a hard problem. Problem-specific test procedures that check invariants or can compute the output exactly can be used [58]. A formal approach used for high-assurance code generation is to show that the final code is derived by a chain of semantics-preserving rewrites [65]. In this case, it needs to be established that all rewriting rules are correct, either by formal methods or probabilistic approaches.

In practice, correctness is achieved without full formal methods guarantees. During the code generation process hundreds of thousands to millions of rule applications happen, and it is very rare that a rule is buggy without introducing easily observable erroneous behavior. Finally, SPIRAL uses unit testing of rules and automatic recursive counterexample production (SPIRAL automatically finds the smallest expression that results in an observed bug) to aid debugging of the rule system. This is implemented using the full power of the GAP computer algebra system, and allows for locally applying correctness checks during rule application to verify that the left-hand side and right-hand side are equivalent at every step.

D. SPIRAL-Based Approach to Small-Scale Linear Algebra

So far in this section, we have provided evidence of the expressiveness of OL and how it could be used to describe well-defined linear and nonlinear mathematical expressions. However, the point-free nature of OL, meaning that input and output quantities are not explicitly represented within an expression, yields a more complex and less conventional representation of linear algebra computations. Consider, for instance, the symmetric rank-k update

$$S = A^T A + S, \quad A \in \mathbb{R}^{k\times n}, S \in \mathbb{R}^{n\times n} \tag{123}$$

where S is a symmetric matrix. In OL, the expression above would be formulated assuming implicit, linearized input and output matrices and the information that matrix S and the one obtained by computing $A^T A$ are symmetric would have to be embedded in the computation rather then being considered a feature of input and output quantities. In the remainder of this section, we will introduce LGen, a code generator for small-scale linear algebra designed after SPIRAL, and we will show how the approach proposed with this paper is not limited by the choice of the input mathematical abstraction. Small-scale linear algebra

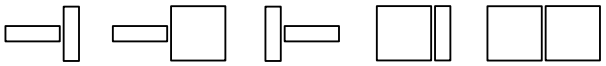


Fig. 13. *The five ν-BLACs used to vectorize matrix multiplication. Matrices are $\nu \times \nu$ and vectors $\nu \times 1$ or $1 \times \nu$.*

computations are common in various domains including computer vision (e.g., stereo vision algorithms), graphics (e.g., geometric transformations), and control systems (e.g., optimization algorithms and Kalman filters).

1) Beyond a Point-Free Notation: The LGen code generator [28], [62] translates basic linear algebra computations (BLACs) such as (123) into C code, optionally vectorized with intrinsics. Differently from SPIRAL, the highest level of abstraction adopts a MATLAB-like notation where a computation is defined using addition, multiplication, scalar multiplication, and transposition over scalars, vectors, and (possibly structured) matrices of small, fixed sizes.

LGen is designed after SPIRAL as it generates code using two intermediate compilation phases. During the first phase, the input expression is optimized at the mathematical level. These optimizations include multilevel tiling, loop merging and exchange, and matrix structure propagation. During the second phase, the mathematical expression obtained from the first phase is translated into a C-intermediate representation where additional code-level optimizations such as loop unrolling and scalar replacement are applied. Finally, since different tiling decisions lead to different code versions of the same computation, LGen uses autotuning to select the fastest version for the target microarchitecture.

2) Tiling: LGen allows for multiple levels of tiling. Tiling is expressed by left- and right-multiplying a matrix by gather matrices as defined in (92). For instance, assuming A is 3×3, then the top left 2×2 submatrix can be extracted using gather matrices as

$$A(0:1,0:1) = GAG^T, \quad G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Note that in OL this operation would be equivalent to the application of $G \otimes G^T$ to matrix A linearized.

More formally, in LGen, we define the gather operator as

$$\begin{aligned} g = [f,p] : \mathbb{R}^{m \times n} &\to \mathbb{R}^{k \times \ell}, \\ A \mapsto Ag = A[f,p] &= G_f A G_p^T \\ &= A(f.[0:k-1], p.[0:\ell-1]) \end{aligned}$$

where $f : \mathbb{I}_k \to \mathbb{I}_m$ and $p : \mathbb{I}_\ell \to \mathbb{I}_n$ are index mapping functions and $f.a$ is a pointwise application of function f to the elements of array a. Similarly, LGen's input language also defines the scatter operator as the dual of the gather with the purpose of inserting submatrices into larger matrices.

3) Vectorization: If vectorization is enabled, the innermost level of tiling decomposes an expression into so-called ν-BLACs, a concept close in spirit to the one of architecture specific templates discussed in Section III-C. ν-BLACs are basic operations on $\nu \times \nu$ matrices and vectors of length ν, where ν is the target SIMD vector length. The four basic operations define 18 of them [28] and they only need to be preimplemented once for a given vector ISA together with vectorized data access building blocks for handling leftovers and structured matrices [62]. For example, Fig. 13 shows the five ν-BLACs for matrix multiplication.

4) Matrix Structure Representation: The cost of computing a BLAC can be significantly reduced if the matrices have structure. For example, multiplying a lower triangular matrix by an upper triangular one requires only half of the total amount of instructions necessary to multiply two general matrices, as shown in Fig. 14. Further, the storage scheme of a structured matrix must be taken into account to ensure correct access to the data. For example, adding a symmetric matrix to a general one may require different access patterns for the two matrices.

LGen uses tools from the polyhedral framework [113] to mathematically describe every input and output matrix of an input BLAC, including those produced by intermediate operations. This information is later used to synthetize the iteration space for the entire expression. For instance, Fig. 14 shows how the synthetized iteration space for a multiplication between a lower and an upper triangular matrix (Fig. 14(b)) differs from the one obtained when no structural assumptions on the inputs are made (Fig. 14(a)). The approach is flexible and can be extended to describe a variety of structures.

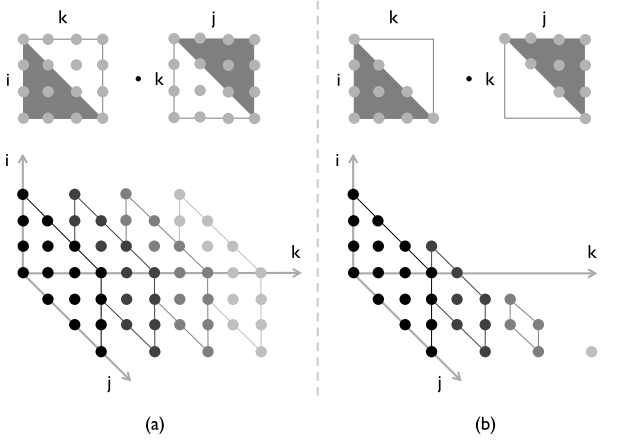


Fig. 14. *Iteration spaces of a matrix multiplication between a lower and upper triangular matrix (a) with and (b) without redundant zero computations. Variables i, j, and k are induction variables and every dot in the two tridimensional spaces represents a single multiply–add operation required to compute the whole expression.*

5) Small-Scale Linear Algebra Programs: Finally, LGen and the Cl1ck algorithm generator [145], [146] were combined and extended into SLinGen, a program generator for linear algebra applications such as the Kalman filter [63]. SLinGen's input can contain a sequence of BLACs and higher level operations including linear systems solvers and the Cholesky factorization. Algorithms for computing the latter kind of operations are automatically synthetized using formal methods techniques while ensuring their correctness [148]. Similarly to LGen, SLinGen generates single-source C functions optionally vectorized for a target ISA.

E. Beyond Software Synthesis

The main focus of the paper so far has been to discuss the end-to-end pipeline SPIRAL uses to generate and autotune highly efficient code across a wide range of platforms for a class of computational kernels. We now discuss applications of SPIRAL beyond this core capability.

1) High-Assurance Code Generation: Our focus in work related to the DARPA HACMS program was to leverage SPIRAL's correctness guarantees to enable the generation of high-assurance code for vehicles and robots [65]. We cast a number of self-consistency algorithms based on statistics [185] and functional redundancy as OL operators and used SPIRAL to generate correct-by-construction high-performance implementations for these algorithms. We enabled SPIRAL to synthesize high-performance interval arithmetic using SIMD vector extensions to provide guarantees for floating-point code, and enabled SPIRAL as a code generation backend for the *KeYmaera* hybrid theorem prover [186].

2) HW/SW Co-Optimization: We targeted hardware/software co-optimization and cosynthesis in the DARPA DESA and PERFECT programs. Since SPIRAL is able to quickly and automatically produce software for a wide range of target platforms, this enables us to pursue hardware/software co-optimization [41]. First, we define a parameterized hardware template that gives rise to a hardware space. Then, we run nested optimization, where the outer loop varies the hardware template parameters while the inner loop invokes SPIRAL for the current instance. A number of metrics can be used (e.g, area, power, and performance).

3) Hardware Microcode: In the DARPA PERFECT program, we used SPIRAL to synthesize the microcode to program the memory controllers of 3-D stacked memory [73]. SPIRAL was used to derive efficient data flows for in-memory data reorganization, and then to derive the configuration data for multiple state machines implementing permutation networks in the base layer of a 3-D memory chip.

4) Backend Tool: SPIRAL can be used as a low-level backend code generation tool, kernel generator, and iterative compiler. The lower layers of the SPIRAL infrastructure serve as a standalone tool that enables a wide range of code generation and program transformation techniques in an interactive and scriptable environment. We have demonstrated the use of the backend as a kernel generator for polyhedral compiler infrastructures [71], [72] and as a code generator for kernels used in power grid Monte Carlo simulations [187] and Boolean satisfiability [188].

V. RESULTS

We now discuss a selection of representative results. We organize the results along three dimensions.

- Machine size: We show SPIRAL's capability to generate efficient code from small, embedded machines through desktop/server class machines up to the largest HPC/supercomputing systems.
- Machine type: We show SPIRAL's capability to target CPUs, GPUs, manycores, and FPGAs.
- Kernel and application type: We show SPIRAL's capability to generate code for FFTs, material science kernels, image reconstruction, and SDR.

Unless noted otherwise, FFT performance results are given in pseudo-Gflop/s, where an operation count of $5n \log_2 n$ is assumed for DFT_n. This is the standard metric used by the community [117].

A. Machine Size

We show performance results for the small/embedded form factor (ARM CPU), desktop/server-class form factor (single Intel CPU), and large scale supercomputer (128k cores on ANL's BlueGene/Q).

1) Embedded CPU: In the embedded form factor we present results on the ARM Juno Board R1 implementing ARM's big.LITTLE architecture. We run the code on a single big 1.1-GHz ARM A57 core. Fig. 15 shows performance

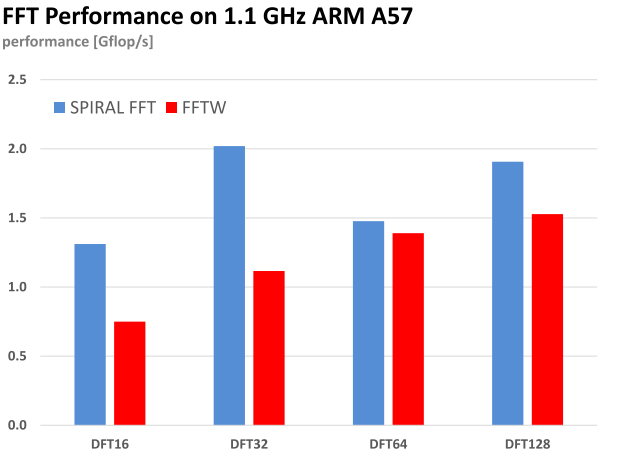


Fig. 15. *FFT kernels (codelets) on 1.1-GHz ARM A57.*

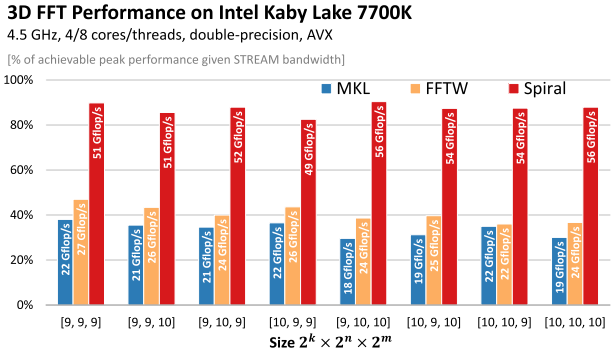


Fig. 16. ***Large FFT on Intel [30].* Spiral *implements a scratchpad-style double-buffering scheme for better memory performance.***

results for small 2-power complex FFT kernels, and compares our results to FFTW. SPIRAL's FFT kernels reach up to 2 Gflop/s and are slightly faster than FFTW.

2) Desktop/Server CPU: Fig. 16 shows performance of large 3-D FFTs on an Intel Kaby Lake 7700K single socket systems [30]. We compare our implementation to MKL 2017.0 and FFTW 3.3.6. All libraries are compiled with OpenMP, AVX and SSE enabled. On the Intel architectures, we use `MKL_DYNAMIC`, `MKL_NUM_THREADS`, and `KMP_AFFINITY` to control the number of threads and the placement of the threads within the MKL library. We compiled all code with the Intel C/C++ compiler version 2017.0 with the `-O3` flag. We show performance for 3-D FFTs of size 512^3 to 1024^3, which have a memory requirement of 32–64 GB. Our implementation runs at 49–56 Gflop/s and achieves 80%–90% of practical peak (the limit imposed by memory bandwidth), whereas MKL and FFTW achieve at most 47%. Our approach uses the bandwidth and the cache hierarchy more efficiently and therefore outperforms MKL and FFTW by almost 3x.

3) HPC/Supercomputing: To show large-scale results, we run the HPCC Global FFT benchmark on ANL's BlueGene/P supercomputer [49]. We used BlueGene/P configurations from one node card (32 quadcore nodes or 128 cores) up to 32 racks (32k quadcore nodes or 128k cores), with one process per node. We used the IBM UPC and IBM's XL C compiler with options `-O3 -qarch=440d`. Fig. 17 summarizes the performance results. We run a UPC+ESSL baseline benchmark on the IBM T.J. Watson BlueGene/P system for up to eight racks. We run our SPIRAL-generated library from one node card to two racks on the T.J. Watson machine and on ANL's "Intrepid" from four racks to 32 racks. The SPIRAL-generated Global FFT generally outperforms the UPC+ESSL baseline which shows that 1) SPIRAL's automatically generated node libraries offer performance competitive with ESSL; and 2) the memory traffic savings obtained by merging data scrambling with the node-libraries improves performance. Finally, the SPIRAL-generated Global FFT reaches 6.4 Tflop/s on 32 racks of "Intrepid." The winning 2008 ANL HPC Challenge Class I [189] submission reported 5-Tflop/s Global FFT performance on the same machine. Thus, the combination of algorithmic optimization and library generation improved the Global FFT on "Intrepid" by 1.4 Tflop/s or 28%.

B. Machine Type

Next, we show results across machine types. We show representative examples for CPUs, accelerators, and FPGAs.

1) Multicore CPU: In the previous section, we showed CPU results from the embedded form factor to server class and HPC multicore CPUs. The results span multiple architectures (ARM, POWER, and x86) and multicore parallelism (single core to 16 cores).

2) Manycore/GPU: We show performance results for the NVIDIA 480GTX Fermi architecture with 480 cores grouped into 15 multiprocessors, 1.5 GB of GPU DRAM main memory and memory peak bandwidth of 177 GB/s. Each core's clock is 1.4 GHz, and each core can perform one fused multiply–add operation per cycle, leading to 1.3-TFlop/s peak performance [36]. Fig. 18 shows performance of batched 1-D power of two single precision floating point FFTs of the corresponding size. SPIRAL's performance is comparable to the well-optimized batch FFT library function of CUFFT that is part of CUDA 4.0 for most of the points in the plot.

3) Field-Programmable Gate Arrays: We show a summary of FPGA results obtained on Xilinx Virtex-6 XC6VLX760 from [44], [46], and [177]. We utilize SPIRAL's framework together with a Verilog backend written in Java that takes as input a hardware formula and outputs synthesizable

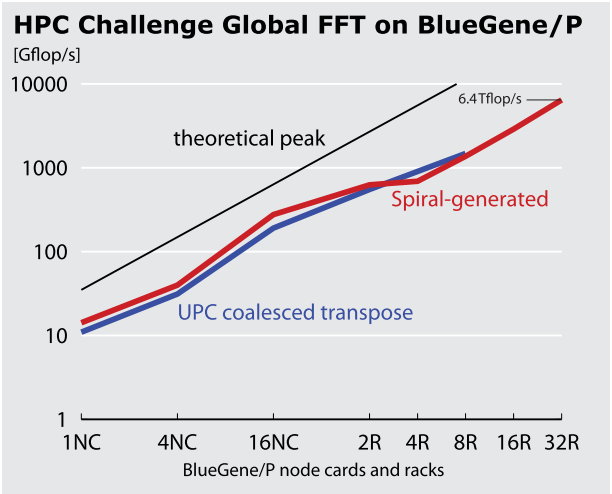


Fig. 17. ***FFT results on the BlueGene/P supercomputer "Intrepid" at Argonne National Laboratory [49]. Thirty two racks (32R) are 128k cores.***

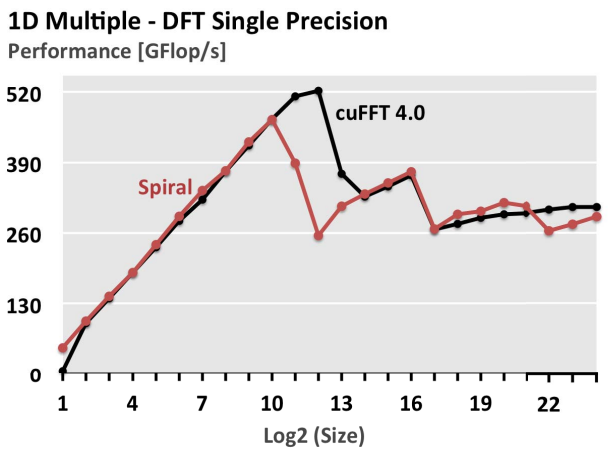


Fig. 18. ***Batch 1-D FFT on NVIDIA 480GTX Fermi GPU:* Spiral *versus CUDA 4.0 CUFFT [36].***

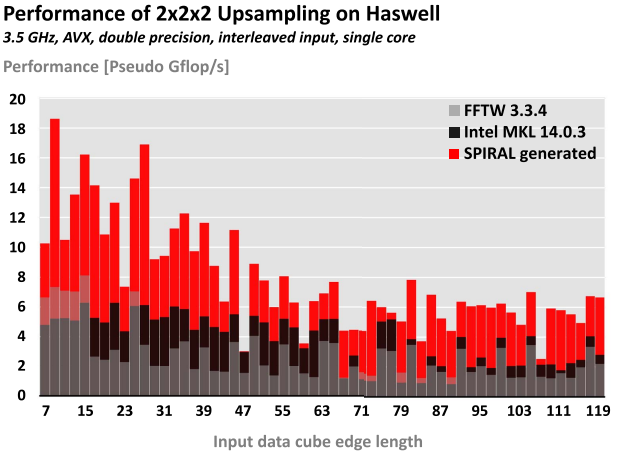


Fig. 20. ***ONETEP* $2 \times 2 \times 2$ *upsampling kernel with small odd-sized 3-D batch FFTs on 3.5-GHz Intel Haswell 4770K:* Spiral *versus FFTW and Intel MKL [61].***

register–transfer level Verilog. All FPGA synthesis is performed using Xilinx ISE version 12.0, and the area and timing data shown in the results are extracted after the final place and route are complete.

We evaluate tradeoffs and compare our FPGA FFT results with implementations from the Xilinx LogiCore IP library. Fig. 19 shows generated designs for DFT_{256}, fixed point, FPGA, throughput versus slices. The Pareto optimal designs are connected with a line. Various implementation choices (radix, reuse type) are coded using shading and bullet shape. Data labels in the plot indicate the number of block RAMs and DSP48E1 slices required. SPIRAL generated designs are competitive with Xilinx LogiCore for design points supported by LogiCore, but span a much larger design space.

C. Kernel and Application Type

Next we show results for a number of applications and kernel types. We start with FFTs, extend to FFT-based signal processing and computational science kernels, and conclude with an SDR kernel.

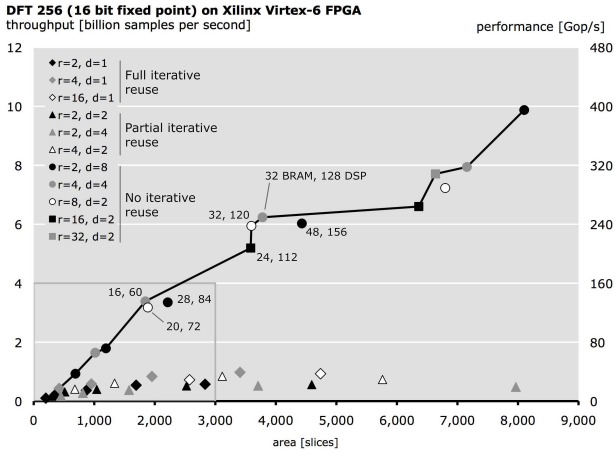


Fig. 19. ***Streaming 1-D* DFT_{256} *on Xilinx Virtex-6 XC6VLX760 FPGA:* Spiral *versus Xilinx LogiCore IP library 12.0 [46].***

1) Fast Fourier Transforms: FFT results have already been presented extensively in the previous section. We show results at a range of sizes from kernel to large 1-D and multidimensional sizes. SPIRAL supports a wide range of FFT corner cases, some of which will be discussed as building blocks below.

2) Upsampling/Convolutions: Upsampling of a multidimensional data set is an operation with wide application in image processing and quantum mechanical calculations using density functional theory. For small upsampling factors as seen in the quantum chemistry code ONETEP [190], a time-shift-based implementation that shifts samples by a fraction of the original grid spacing to fill in the intermediate values using a frequency domain Fourier property can be a good choice [191]. This kernel requires multiple stages of 3-D FFT-based convolution and interleaving. Fig. 20 shows the performance of the central $2 \times 2 \times 2$ upsampling kernel on a 3.5-GHz Intel Haswell 4770K [63]. Note that the original data cube is small (edge length between 7 and 119), odd, and may be rectangular. These unusual requirements render standard FFT libraries (Intel MKL and FFTW) suboptimal and allow SPIRAL-generated end-to-end kernels to be three times faster.

3) Image Reconstruction: Polar formatting synthetic aperture radar (SAR) [174] is an FFT-based image reconstruction algorithm. We generated SAR kernels for a 4k×4k (16 Megapixel) and 10k×10k (100 Megapixel) following [192] with SPIRAL. Both scenarios have longer slant ranges (24 and 200 km, respectively), fine resolution (0.1 and 0.3 m) and small coherent integration angles (approximately4° and 7°). In Fig. 21, we show performance results on the Intel Quad Core CPUs: the 3.0-GHz 5160, the 3.0-GHz X9560, and the 2.66-GHz

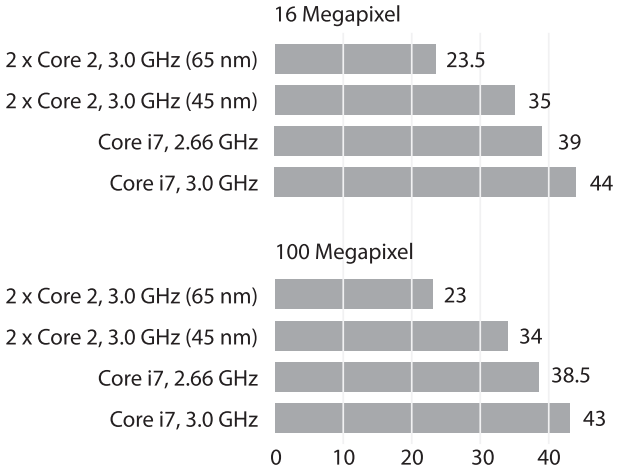


Fig. 21. ***Performance of* Spiral*-generated polar formatting SAR image formation on 3.0-GHz Intel 5160, the 3.0-GHz Intel X9560, and the 2.66-GHz Intel Core i7 920 for 16 and 100 Megapixel [56].***

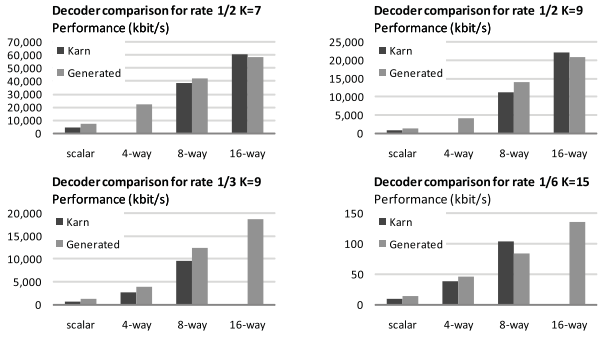


Fig. 22. ***Software Viterbi decoder on a 3-GHz Intel Core 2 Extreme X9650 for a range of codes:* Spiral *versus Karn's library [58].***

Core i7 920, using Intel's C compiler 11.0.074 with `-O3`. The code uses the SSE2 instruction set and the POSIX threading interface. The SPIRAL-generated SAR kernels achieve between 23 and 44 Gflop/s [56]. The fastest measurements are comparable to the performance obtained by [192] on a Cell BE server blade.

4) Software-Defined Radio (SDR): SPIRAL demonstrated the end-to-end generation of physical-layer software required by SDR [60], and the Viterbi decoder is a key component that needs to be well optimized and is the focus of our evaluation here. The Viterbi algorithm is a maximum-likelihood sequence decoder introduced by Andrew Viterbi in 1973 [193], and finds wide usage in communications, speech recognition, and statistical parsing. As a decoder for convolutional codes, it is used in a broad range of everyday applications and telecommunication standards including wireless communication (e.g., cellphones and satellites) and high-definition television [194].

Fig. 22 shows SPIRAL-generated Viterbi decoders compared to state-of-the-art hand-tuned decoders by Karn [195]. We use SPIRAL to generate the forward pass of Viterbi decoders and use a generic infrastructure to implement the rest of the algorithm [58]. We show performance results on a 3-GHz Intel Core 2 Extreme X9650. All code is compiled using the Intel C/C++ Compiler Version 10.1 with performance flags `-fast -fomit-frame-pointer -fno-alias`. We see that SPIRAL-generated Viterbi decoders are competitive with Karn's implementation, and we provide some kernels that are missing in Karn's library.

5) Performance Results of SPIRAL*-Based Approaches:* In Section IV-D, we have discussed the SPIRAL-related LGen compiler for small-scale linear algebra computations and its extension into the SLinGen program generator for small-scale linear algebra applications. In Fig. 23, we show performance results for code generated using both systems [64]. All results are obtained running on an Intel Sandy Bridge (AVX, 32-kB L1-D cache, 256-kB L2 cache) with Ubuntu 14.04 (Linux 3.13). All code is in double precision and the working set fits in the first two levels of cache. Fig. 23(a) shows performance for a rank-4 update ($S = AA^T + S$, $A \in \mathbb{R}^{n\times 4}$, $S \in \mathbb{R}^{n\times n}$ and symmetric). We compare LGen-generated code with: 1) Intel MKL 11.2; 2) straightforward code compiled with Intel C/C++ compiler 15; and 3) code generated by LGen without structure support. The straightforward code is scalar, handwritten, loop-based code with hardcoded sizes of the matrices. For this we use flags `-O3 -xHost -fargument-noalias -fno-alias -no-ipo -no-ip`. LGen is between 1.6x and 2.5x faster than MKL, while in general it is 1.6x faster than Intel C/C++ compiler compiled code.

Fig. 23(b) shows performance for a single iteration of the Kalman filter with state size varying between 4 and 52 (linear algebra program provided in [63]). In this case, we compare SLinGen generated code against 1) straightforward code compiled the Intel C/C++ compiler 16 and library-based implementations using 2) MKL 11.3 and 3) Eigen v3.3.4 [151]. In Eigen, we used fixed-size Map

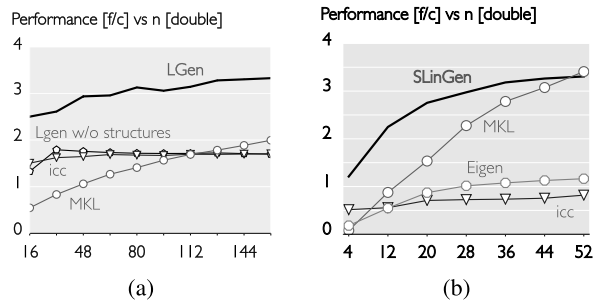


Fig. 23. ***Performance of: (a) a rank-4 update generated with LGen [62] versus MKL, Intel C compiler (icc) compiled code, and LGen disabling structure support; (b) a single iteration of the Kalman filter generated with SLinGen [63] versus MKL, Eigen, and Intel C compiler (icc) compiled code. Code tested on an Intel Core i7-2600K (Sandy Bridge microarchitecture).***

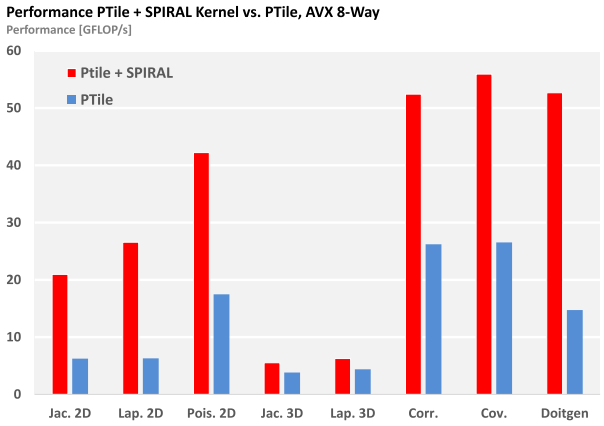


Fig. 24. ***End-to-end stencil performance on a 3.4-GHz Intel Core i7-2600K for a range of stencil kernels: PLuTo/PTile [98] together with the* Spiral *backend versus PTile plus Intel C/C++ compiler [71].***

interfaces to existing arrays, no-alias assignments, in-place computations of solvers, and enabled AVX code generation. SLinGen generates code which is on average 1.4x, 3x, and 4x faster than MKL, Eigen, and the Intel C/C++ compiler.

6) SPIRAL *as Backend Tool:* Last, we demonstrate that SPIRAL's backend compiler is a valuable standalone code generation tool. We use SPIRAL's infrastructure from `icode` downwards as a code generator for stencil kernels in the context of the polyhedral PTile system [71]. Fig. 24 shows the performance of the end-to-end stencil code on a 3.4-GHz Intel Core i7-2600K by PLuTo/PTile [98] together with the SPIRAL backend versus the kernels generated by PTile and directly sent to the C compiler. We see that the utilization of SPIRAL as kernel generator leads to substantial performance improvement (typically two to three times) across a range of stencil kernels.

VI. CURRENT WORK

The SPIRAL system is available as open source software under a permissive license from `www.spiral.net`. We now discuss current and experimental work aimed at extending SPIRAL's capabilities.

A. Targeting of New Architectures

SPIRAL is designed to be quickly retargeted to new and novel architectures. We are continually looking for new instruction sets and new processor generations to add to the library of supported platforms. This is done at various stages of the development cycle, depending on the level of access the SPIRAL team has: presilicon with functional or cycle-accurate simulators, early prototypes with limited compiler and OS support, or just-released production models.

B. Field-Programmable Gate Arrays

Targeting FPGAs for the full suite of kernels/applications supported by SPIRAL is an open problem and a target of current research. This builds on previous work targeting FFTs [43] and aims at current generation FPGAs with OpenCL and HLS (high-level synthesis) support [184].

C. Cross-Library-Call Optimization

We are extending SPIRAL's capabilities beyond the current set of applications where SPIRAL can optimize across the traditional boundaries of library calls. We have demonstrated this for convolution operations [61] and image processing kernels that combine spectral and numerical linear algebra functionality [196] and are developing this capability further in the ExaScale FFT system FFTX [85].

D. SPIRAL as JIT

This paper focuses on code generation for fixed-size kernels (the size is known at code synthesis time). A branch of SPIRAL called *Autolib* [18] is able to automatically generate full autotuning libraries for signal processing kernels and matrix multiplication [197]. We are currently developing a JIT code synthesis approach for SPIRAL that aims at bringing this beyond linear signal processing kernels and matrix multiplication.

E. MATLAB Frontend

To ease the adoption of SPIRAL and its mathematical notation to describe algorithms (OL and Σ-OL as used in this paper), we have developed a prototype MATLAB frontend that exposes a subset of OL as MATLAB tensors and operations on tensors (map, rotate, fold, reshape). This is ongoing research that builds on the idea of HTAs [82].

F. Graph Algorithms

We are currently extending SPIRAL to support graph algorithms. The approach to view graph algorithms as linear algebra operations [172] and the resulting GraphBLAS standard [198] enables us to apply SPIRAL's formal framework. The key insight is that the function parameter in OL's gather operators can be used to abstract sparse matrix data formats. This combined with SPIRAL's general size support opens the way to abstract graph algorithms [199].

G. Formal Correctness Proofs

With OL and Σ-OL, SPIRAL defines a mathematical framework that provides mathematical correctness guarantees and a variety of testing approaches. SPIRAL is implemented in the computer algebra system GAP, which enables symbolic arithmetic, and SPIRAL supports fast interval arithmetic. Together, this allows us to provide strong end-to-end correctness guarantees for SPIRAL-synthesized code [65], [74]. We are currently working on developing a formal methods framework in the proof assistant Coq [200] that proves that the synthesized code is a refinement of the original specification [201], [202].

VII. CONCLUSION

In this paper, we presented an overview of the SPIRAL system, a sampling of results, and an indication of current and future directions. This paper summarizes research conducted over the last 13 years and builds on the previous overview paper [1]. Due to space limitations, we leave out details that can be found in the references. The paper focuses on a coherent end-to-end discussion of SPIRAL's formal system and the concepts used. For details on how specific algorithms and hardware are handled in SPIRAL, we refer the reader to the literature covering the specifics.

The key message of this paper (and the SPIRAL system and approach in general) is that it is possible to design and build a system that provides performance portability across a wide range of (past, current, and future) architectures for a set of algorithms. The system is necessarily complex but very powerful. This paper should give interested readers who want to try the SPIRAL system for themselves a starting point.

Acknowledgments

The authors would like to acknowledge the many discussions with colleagues David Padua and Manuela Veloso and the support provided in the last ten years to SPIRAL by NSF, ONR, and DARPA (in particular, the HACMS, PERFECT, and BRASS programs), the CMU Software Engineering Institute, and the DOE ExaScale program. A full list of supporting agencies and companies as well as involved individuals can be found at `www.spiral.net`.

REFERENCES

[1] M. Püschel *et al.*, "Spiral: Code generation for DSP transforms," *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, 2005.
[2] Euclid, *Elements*, c. 300 BC.
[3] "Fangcheng," in *The Nine Chapters on the Mathematical Art*, ch. 8.
[4] I. Newton, *Philosophiæ Naturalis Principia Mathematica*, 1687.
[5] M. T. Heideman, D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast Fourier transform," *Arch. History Exact Sci.*, vol. 34, no. 3, pp. 265–277, 1985.
[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.
[7] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Philadelphia, PA, USA: SIAM, 1992.
[8] Intel. *Product Specification: Processors*. [Online]. Available: www.ark.intel.com/#Processors
[9] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *Circuits Syst. Signal Process.*, vol. 9, no. 4, pp. 449–500, 1990.
[10] C. H. R. W. Johnson and J. R. Johnson, "Multilinear algebra and parallel programming," *J. Supercomput.*, vol. 5, pp. 189–217, Oct. 1991.
[11] J. R. Johnson and A. F. Breitzman, "Automatic derivation and implementation of fast convolution algorithms," *J. Symbolic Comput.*, vol. 37, no. 2, pp. 261–293, 1997.
[12] J. M. F. Moura *et al.*, "Spiral: Automatic implementation of signal processing algorithms," in *Proc. High Perform. Extreme Comput. (HPEC)*, 2000.
[13] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. Program. Lang. Design Implement. (PLDI)*, 2001, pp. 298–308.
[14] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura, "Fast automatic generation of DSP algorithms," in *Proc. Int. Conf. Comput. Sci. (ICCS)*, 2001, pp. 97–106.
[15] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*. New York, NY, USA: Springer-Verlag, 2011.
[16] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from Berkeley," Dept. Electr. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2006-183, 2006.
[17] F. Franchetti, Y. Voronenko, and M. Püschel, "Loop merging for signal transforms," in *Proc. Program. Lang. Design Implement. (PLDI)*, 2005, pp. 315–326.
[18] Y. Voronenko, F. de Mesmay, and M. Püschel, "Computer generation of general size linear transform libraries," in *Proc. Intell. Symp. Code Gener. Optim. (CGO)*, 2009, pp. 102–113.
[19] F. Franchetti, Y. Voronenko, and M. Püschel, "FFT program generation for shared memory: SMP and multicore," in *Proc. Supercomput.*, 2006, p. 51.
[20] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, "Discrete Fourier transform on multicore," *IEEE Signal Process. Mag.*, vol. 26, no. 6, pp. 90–102, Nov. 2009.
[21] F. Franchetti and M. Püschel, "A SIMD vectorizing compiler for digital signal processing algorithms," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2002, pp. 20–26.
[22] F. Franchetti and M. Püschel, "Short vector code generation for the discrete Fourier transform," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2003, p. 10.
[23] F. Franchetti and M. Püschel, "Short vector code generation and adaptation for DSP algorithms," in *Proc. Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, vol. 2, 2003, pp. 537–540.
[24] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient utilization of SIMD extensions," *Proc. IEEE*, vol. 93, no. 2, pp. 409–425, Feb. 2005.
[25] F. Franchetti, Y. Voronenko, and M. Püschel, "A rewriting system for the vectorization of signal transforms," in *High Performance Computing for Computational Science*. Berlin, Germany: Springer-Verlag, ser.Lecture Notes in Computer Science, 2006, vol. 4395, pp. 363–377.
[26] F. Franchetti and M. Püschel, "Generating SIMD vectorized permutations," in *Proc. Int. Conf. Compiler Construct. (CC)*, 2008, pp. 116–131.
[27] F. Franchetti and M. Püschel, "Generating high performance pruned FFT implementations," in *Proc. Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, 2009, pp. 549–552.
[28] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2014, pp. 117–127.
[29] T. Popovici, F. Franchetti, and T.-M. Low, "Mixed data layout kernels for vectorized complex arithmetic," in *Proc. High Perform. Extreme Comput. (HPEC)*, 2017, pp. 1–7.
[30] T. Popovici, T.-M. Low, and F. Franchetti, "Large bandwidth-efficient FFTs on multicore and multi-socket systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2018.
[31] L. J. Chang, I. Hong, Y. Voronenko, and M. Püschel, "Adaptive mapping of linear DSP algorithms to fixed-point arithmetic," in *Proc. High Perform. Embedded Comput. (HPEC)*, 2004.
[32] Y. Voronenko and M. Püschel, "Mechanical derivation of fused multiply–add algorithms for linear transforms," *IEEE Trans. Signal Process.*, vol. 55, no. 9, pp. 4458–4473, Sep. 2007.
[33] L. Meng, J. Johnson, F. Franchetti, Y. Voronenko, M. M. Maza, and Y. Xie, "Spiral-generated modular FFT algorithms," in *Proc. Parallel Symbolic Comput. (PASCO)*, 2010, pp. 169–170.
[34] F. Franchetti *et al.*, "Domain-specific library generation for parallel software and hardware platforms," in *Proc. NSF Next Gener. Softw. Program Workshop (NSFNGS)*, 2008, pp. 1–5.
[35] F. Franchetti *et al.*, "Program generation with Spiral: Beyond transforms," in *Proc. High Perform. Embedded Comput. (HPEC)*, 2008, pp. 1–2.
[36] C. Angelopoulos, F. Franchetti, and M. Püschel, "Automatic generation of FFT libraries for GPUs," in *Proc. NVIDIA Res. Summit GPU Technol. Conf.*, 2012.
[37] P. D'Alberto, M. Püschel, and F. Franchetti, "Performance/energy optimization of DSP transforms on the XScale processor," in *Proc. Int. Conf. High Perform. Embedded Archit. Compil. (HiPEAC)*, 2007.
[38] S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of fast Fourier transforms for the Cell Broadband Engine," in *Proc. Int. Conf. Supercomput. (ICS)*, 2009, pp. 26–35.
[39] D. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic SIMD vectorization of fast Fourier transforms for the Larrabee and AVX instruction sets," in *Proc. Int. Conf. Supercomput. (ICS)*, 2011, pp. 265–274.
[40] G. Nordin, P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of customized discrete Fourier transform IPs," in *Proc. Design Autom. Conf. (DAC)*, 2005, pp. 471–474.
[41] P. D'Alberto *et al.*, "Generating FPGA-accelerated DFT libraries," in *Proc. IEEE Symp. Field-Program. Custom Comput. Machines (FCCM)*, Apr. 2007, pp. 173–184.
[42] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Proc. Design Autom. Conf.*, 2008, pp. 385–390.
[43] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Linear transforms: From math to efficient hardware," in *Proc. Workshop High-Level Synth. Colocated DAC*, 2008.
[44] P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of streaming datapaths for arbitrary fixed permutations," in *Proc. Design, Autom. Test Eur. (DATE)*, 2009, pp. 1118–1123.
[45] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Hardware implementation of the discrete Fourier transform with non-power-of-two problem size," in *Proc. Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, 2010, pp. 1546–1549.
[46] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Design Automat. Electron. Syst.*, vol. 17, no. 2, 2012, Art. no. 15.
[47] A. Bonelli, F. Franchetti, J. Lorenz, M. Püschel, and C. W. Ueberhuber, "Automatic performance

optimization of the discrete Fourier transform on distributed memory computers," in *Proceedings of the International Symposium on Parallel and Distributed Processing and Applications*. Berlin, Germany: Springer-Verlag, ser. Lecture Notes In Computer Science, vol. 4330, 2006, pp. 818–832.

[48] F. Gygi *et al.*, "Large-scale electronic structure calculations of high-Z metals on the bluegene/L platform," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, 2006, p. 45.

[49] F. Franchetti, Y. Voronenko, and G. Almasi, "Automatic generation of the HPC challenge's global FFT benchmark for BlueGene/P," in *Proc. High Perform. Comput. Comput. Sci. (VECPAR)*, 2012, pp. 187–200.

[50] B. Duff, J. Larkin, M. Franusich, and F. Franchetti, "Automatic generation of 3-D FFTs," in *Proc. Rice Oil Gas HPC Workshop*, 2014.

[51] F. Franchetti and M. Püschel, "SIMD vectorization of non-two-power sized FFTs," in *Proc. Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, vol. 2, 2007, pp. II-17–II-20.

[52] A. Gačić, M. Püschel, and J. M. F. Moura, "Fast automatic software implementations of FIR filters," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, vol. 2, Apr. 2003, pp. 541–544.

[53] A. Gačić, "Automatic implementation and platform adaptation of discrete filtering and wavelet algorithms," Ph.D. dissertation, Dept. Electr. Comput. Eng., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2004.

[54] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *Proc. IFIP Working Conf. Domain Specific Lang. (DSL WC)*, 2009, pp. 385–409.

[55] Y. Voronenko, F. Franchetti, F. de Mesmay, and M. Püschel, "System demonstration of Spiral: Generator for high-performance linear transform libraries," in *Algebraic Methodology and Software Technology*. Berlin, Germany: Springer-Verlag, ser. Lecture Notes in Computer Science, vol. 5140, 2008, pp. 407–412.

[56] D. S. McFarlin, F. Franchetti, J. M. F. Moura, and M. Püschel, "High-performance synthetic aperture radar image formation on commodity multicore architectures," in *Proc. SPIE Conf. Defense Secur. Sens.*, 2009, p. 733708.

[57] H. Shen, "Generation of a fast JPEG 2000 encoder using SPIRAL," M.S. thesis, Dept. Inf. Math. Model., Lang.-Based Technol., Tech. Univ. Denmark, Kongens Lyngby, Denmark, 2008.

[58] F. de Mesmay, S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of efficient software Viterbi decoders," in *High-Performance Embedded Architectures and Compilers*. Berlin, Germany: Springer-Verlag, ser. Lecture Notes in Computer Science, vol. 5952, 2010, pp. 353–368.

[59] Y. Voronenko, V. Arbatov, C. Berger, R. Peng, M. Püschel, and F. Franchetti, "Computer generation of platform-adapted physical layer software," in *Proc. Softw. Defined Radio (SDR)*, 2010, pp. 1–8.

[60] C. Berger, V. Arbatov, Y. Voronenko, F. Franchetti, and M. Püschel, "Real-time software implementation of an IEEE 802.11a baseband receiver on Intel multicore," in *Proc. Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, 2011, pp. 1693–1696.

[61] D. T. Popovici, F. P. Russell, K. Wilkinson, C.-K. Skylaris, P. H. J. Kelly, and F. Franchetti, "Generating optimized Fourier interpolation routines for density functional theory using SPIRAL," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2015, pp. 743–752.

[62] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler for structured matrices," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2016, pp. 117–127.

[63] D. G. Spampinato, D. Fabregat-Traver, P. Bientinesi, and M. Püschel, "Program generation for small-scale linear algebra applications," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2018, pp. 327–339.

[64] M. Bolten, F. Franchetti, P. H. J. Kelly, C. Lengauer, and M. Mohr, "Algebraic description and automatic generation of multigrid methods in SPIRAL," *Concurrency Comput. Pract. Exper.*, vol. 29, no. 17, 2017, Art. no. e4105.

[65] F. Franchetti *et al.*, "High-assurance SPIRAL: End-to-end guarantees for robot and car control," *IEEE Control Syst.*, vol. 37, no. 2, pp. 82–103, Apr. 2017.

[66] B. B. Fraguela, Y. Voronenko, and M. Püschel, "Automatic tuning of discrete Fourier transforms driven by analytical modeling," in *Proc. Parallel Archit. Compilation Techn.*, 2009, pp. 271–280.

[67] B. Singer and M. M. Veloso, "Automating the modeling and optimization of the performance of signal transforms," *IEEE Trans. Signal Process.*, vol. 50, no. 8, pp. 2003–2014, Aug. 2002.

[68] B. Singer and M. Veloso, "Learning to construct fast signal processing implementations," *J. Mach. Learn. Res.*, vol. 3, pp. 887–919, Dec. 2002.

[69] F. de Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel, "Bandit-based optimization on graphs with application to library performance tuning," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2009, pp. 729–736.

[70] F. de Mesmay, Y. Voronenko, and M. Püschel, "Offline library adaptation using automatically generated heuristics," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Apr. 2010, pp. 1–10.

[71] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector SIMD architectures," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. (ICS)*, 2013, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2464996.2467268

[72] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet SIMD code generation," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2013, pp. 127–138. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462187

[73] B. Akin, J. C. Hoe, and F. Franchetti, "HAMLeT: Hardware accelerated memory layout transform within 3D-stacked DRAM," in *Proc. IEEE High Perform. Extreme Comput. (HPEC)*, Sep. 2014, pp. 1–6.

[74] T.-M. Low and F. Franchetti, "High assurance code generation for cyber-physical systems," in *Proc. IEEE Int. Symp. High Assurance Syst. Eng. (HASE)*, Jan. 2017, pp. 104–111.

[75] *Developer Guide for Intel Math Kernel Library*, Intel, Santa Clara, CA, USA, 2018.

[76] *Developer Guide for Intel Integrated Performance Primitives (Intel IPP)*, Intel, Santa Clara, CA, USA, 2018.

[77] J. Moreira *et al.*, "The Blue Gene/L supercomputer: A hardware and software story," *Int. J. Parallel Program.*, vol. 35, no. 3, pp. 181–206, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10766-007-0037-2

[78] Mercury Computing. *OpenSAL*. [Online]. Available: https://sourceforge.net/projects/opensal/

[79] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel, "Spiral in scala: Towards the systematic construction of generators for performance libraries," in *Proc. Int. Conf. Generative Program. Concepts Exper. (GPCE)*, 2013, pp. 125–134.

[80] G. Ofenbeck, T. Rompf, and M. Püschel, "Staging for generic programming in space and time," in *Proc. Int. Conf. Generative Program. Concepts Exper. (GPCE)*, 2017, pp. 15–28.

[81] G. Mainland and J. Johnson, "A Haskell compiler for signal transforms," in *Proc. Int. Conf. Generative Program. Concepts Exper. (GPCE)*, 2017, pp. 219–232.

[82] G. Almasi, L. De Rose, B. B. Fraguela, J. Moreira, and D. Padua, "Programming for locality and parallelism with hierarchically tiled arrays," in *Proc. 16th Int. Workshop Lang. Compil. Parallel Comput. (LCPC)*, 2003, pp. 162–176.

[83] G. Bikshandi *et al.*, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proc. Symp. Principles Pract. Parallel Program. (PPoPP)*, 2006, pp. 48–57.

[84] The GAP Team, University of St Andrews, Scotland. (1997). *GAP Groups, Algorithms and Programming*. [Online]. Available: www-gap.dcs.st-and.ac.uk/∼gap/

[85] F. Franchetti *et al.*, "FFTX and SpectralPack: A first look," in *Proc. PFFT Workshop HiPC*, 2018.

[86] Advanced Micro Devices. (2009). *AMD Core Math Library*. [Online]. Available: http://developer.amd.com/cpu/libraries/acml/pages/default.aspx

[87] Cray Inc. (2009). *Cray Scientific Libraries*. [Online]. Available: http://www.cray.com

[88] *Engineering and Scientific Subroutine Library for AIX, Version 4 Release 2*, 2003.

[89] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[90] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, no. 1, pp. 1–17, Mar. 1988.

[91] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[92] K. Goto. (2008). *GotoBLAS 1.26*. [Online]. Available: http://www.tacc.utexas.edu/resources/software/#blas

[93] E. Anderson *et al.*, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA, USA: SIAM, 1999.

[94] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[95] J. Ansel and C. Chan, "PetaBricks: Building adaptable and more efficient programs for the multi-core era," *XRDS, Crossroads, ACM Mag. Students—Changing Face Program.*, vol. 17, no. 1, pp. 32–37, Sep. 2010. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2010/ansel-xrds.pdf

[96] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," USC Comput. Sci., Tech. Rep. 08-897, 2008.

[97] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-Stream compiler," in *Encyclopedia of Parallel Computing*. New York, NY, USA: Springer-Verlag, 2011.

[98] U. Bondhugula and J. Ramanujam, "PLuTo: A practical and fully automatic polyhedral program optimization system," in *Proc. ACM SIGPLAN Conf. Program. Language Design Implement. (PLDI)*, 2008, pp. 1–15.

[99] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, no. 4, p. 1250010, 2012.

[100] T. Grosser and T. Hoefler, "Polly-ACC transparent compilation to heterogeneous hardware," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 1:1–1:13.

[101] S. Verdoolaege *et al.*, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400713

[102] S. Z. Guyer and C. Lin, "An annotation language for optimizing software libraries," *SIGPLAN Notices*, vol. 35, no. 1, pp. 39–52, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/331963.331970

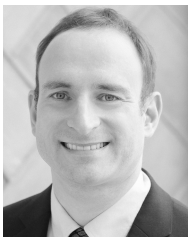
[103] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson, "Automatic type-driven library generation for telescoping languages," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, 2003, p. 51. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050201

[104] S. Benkner and H. Zima, "Compiling high

performance Fortran for distributed-memory architectures," *Parallel Comput.*, vol. 25, nos. 13–14, pp. 1785–1825, 1999.
[105] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of high performance Fortran: An historical object lesson," in *Proc. 3rd ACM SIGPLAN Conf. Hist. Program. Lang.*, 2007, pp. 7-1–7-22.
[106] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.
[107] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *Proc. 20th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2005, pp. 519–538.
[108] UPC Consortium, "UPC language specifications, V1.2," Lawrence Berkeley National Lab, Berkeley, CA, USA, Tech. Rep. LBNL-59208, 2005.
[109] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.
[110] Z. Pan and R. Eigenmann, "PEAK—A fast and effective performance tuning system via compiler optimization orchestration," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 1–43, 2008. [Online]. Available: http://www.ecn.purdue.edu/ParaMount/publications/PaEi08.pdf
[111] K. Fatahalian *et al.*, "Sequoia: Programming the memory hierarchy," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543
[112] A. Hartono *et al.*, "Parametric multi-level tiling of imperfectly nested loops," in *Proc. 23rd Int. Conf. Supercomput. (ICS)*, 2009, pp. 147–157.
[113] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proc. IEEE Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Juan-les-Pins, France, Sep. 2004, pp. 7–16.
[114] G. Fursin *et al.*, "MILEPOST GCC: Machine learning based research compiler," in *Proc. GCC Summit*, 2008, pp. 1–13.
[115] R. S. Nikhil and Arvind, "What is Bluespec?" *SIGDA Newslett.*, vol. 39, no. 1, p. 1, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1862876.1862877
[116] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 1999, pp. 169–180.
[117] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.
[118] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software (ATLAS)," in *Proc. Supercomput.*, 1998.
[119] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Comput.*, vol. 27, nos. 1–2, pp. 3–35, 2001.
[120] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, 2004.
[121] J. Demmel *et al.*, "Self-adapting linear algebra algorithms and software," *Proc. IEEE*, vol. 93, no. 2, pp. 293–312, Feb. 2005.
[122] G. Baumgartner *et al.*, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proc. IEEE*, vol. 93, no. 2, pp. 276–292, Feb. 2005.
[123] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "ActiveHarmony: Towards automated performance tuning," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, 2002, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=762761.762771
[124] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2009, pp. 1–11.
[125] T. Katagiri and D. Takahashi, "Japanese auto-tuning research: Auto-tuning languages and FFT," *Proc. IEEE*, vol. 106, no. 11, 2018.
[126] *ACM Conference on Generative Programming and Component Engineering*.
[127] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Reading, MA, USA: Addison-Wesley, 2000.
[128] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder, "Achieving extensibility through product-lines and domain-specific languages: A case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 191–214, 2002.
[129] D. Batory, R. Lopez-Herrejon, and J.-P. Martin, "Generating product-lines of product-families," in *Proc. Automat. Softw. Eng. Conf. (ASE)*, 2002, pp. 81–92.
[130] D. R. Smith, "Mechanizing the development of software," in *Calculational System Design* (NATO ASI Series), M. Broy, Ed. Amsterdam, The Netherlands: IOS Press, 1999.
[131] K. J. Gough, "Little language processing, an alternative to courses on compiler construction," *ACM SIGCSE Bull.*, vol. 13, no. 3, pp. 31–34, 1981.
[132] J. Bentley, "Programming pearls: Little languages," *Commun. ACM*, vol. 29, no. 8, pp. 711–721, 1986.
[133] P. Hudak, *Domain Specific Languages*, 1997.
[134] W. Taha, "Domain-specific languages," in *Proc. Int. Conf. Comput. Eng. Syst. (ICCES)*, 2008.
[135] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha, "DSL implementation in MetaOCaml, template Haskell, and C," in *Domain-Specific Program Generation*, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds., ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2004.
[136] B. Catanzaro *et al.*, "SEJITS: Getting productivity and performance with selective embedded JIT specialization," Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2010-23, Mar. 2010. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-23.html
[137] A. K. Sujeeth *et al.*, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4s, pp. 134:1–134:25, 2014.
[138] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs," *Commun. ACM*, vol. 55, no. 6, pp. 121–130, 2012. [Online]. Available: http://doi.acm.org/10.1145/2184319.2184345
[139] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: A multi-stage language for high-performance computing," in *Proc. Program. Lang. Design Implement. (PLDI)*, 2013, pp. 105–116. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462166
[140] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch, "How to architect a query compiler," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2016, pp. 1907–1922. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915244
[141] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2018, pp. 307–322.
[142] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with Lift," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2018, pp. 100–112.
[143] J. G. Siek, I. Karlin, and E. R. Jessup, "Build to order linear algebra kernels," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2008, pp. 1–8.
[144] T. Nelson, G. Belter, J. G. Siek, E. Jessup, and B. Norris, "Reliable generation of high-performance matrix algebra," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 18:1–18:27, 2015.
[145] D. Fabregat-Traver and P. Bientinesi, "Knowledge-based automatic generation of partitioned matrix expressions," in *Computer Algebra in Scientific Computing*, 2011, pp. 144–157.
[146] D. Fabregat-Traver and P. Bientinesi, "Automatic generation of loop-invariants for matrix operations," in *Proc. Int. Conf. Comput. Sci. Appl. (ICCSA)*, 2011, pp. 82–92.
[147] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "FLAME: Formal linear algebra methods environment," *ACM Trans. Math. Softw.*, vol. 27, no. 4, pp. 422–455, Dec. 2001.
[148] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn, "The science of deriving dense linear algebra algorithms," *ACM Trans. Math. Softw.*, vol. 31, no. 1, pp. 1–26, Mar. 2005.
[149] B. Marker, J. Poulson, D. Batory, and R. van de Geijn, "Designing linear algebra algorithms by transformation: Mechanizing the expert developer," in *High Performance Computing for Computational Science*, ser. Lecture Notes in Computer Science, vol. 7851. Berlin, Germany: Springer-Verlag, 2013, pp. 362–378.
[150] D. Fabregat-Traver and P. Bientinesi, "A domain-specific compiler for linear algebra operations," in *High Performance Computing for Computational Science*ser. Lecture Notes in Computer Science, vol. 7851. Berlin, Germany: Springer-Verlag, 2013, pp. 346–361.
[151] G. Guennebaud. (2017). *Eigen*. [Online]. Available: http://eigen.tuxfamily.org
[152] J. Walter *et al. uBLAS*. [Online]. Available: www.boost.org/libs/numeric
[153] P. Gottschling, D. S. Wise, and A. Joshi, "Generic support of algorithmic and structural recursion for scientific computing," *Int. J. Parallel Emergent Distrib. Syst.*, vol. 24, no. 6, pp. 479–503, 2009.
[154] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "STELLA: A domain-specific tool for structured grid methods in weather and climate models," in *Proc. High Perform. Comput. Netw. Storage Anal. (SC)*, 2015, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807627
[155] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2013, pp. 519–530. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462176
[156] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462176
[157] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, pp. 77:1–77:29, Oct. 2017.
[158] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," *SIGPLAN Notices*, vol. 40, no. 6, pp. 281–294, Jun. 2005.
[159] M. Vechev, E. Yahav, and G. Yorsh, "Inferring synchronization under limited observability," in *Proc. 15th Int. Conf. Tools Algorithms Construct. Anal. Syst. (TACAS)*, 2009, pp. 139–154.
[160] N. Dershowitz and D. A. Plaisted, "Rewriting," in *Handbook of Automated Reasoning*, vol. 1, A. Robinson and A. Voronkov, Eds. Amsterdam, The Netherlands: Elsevier, 2001, ch. 9, pp. 535–610.
[161] U. Nilsson and J. Maluszynski, *Logic, Programming, and PROLOG*, 2nd ed. Hoboken, NJ, USA: Wiley, 1995.
[162] A. J. Field and P. G. Harrison, *Functional Programming*. Reading, MA, USA: Addison-Wesley, 1988.
[163] Y. Jia (2014). "Caffe: Convolutional architecture for fast feature embedding." [Online]. Available: https://arxiv.org/abs/1408.5093
[164] R. Al-Rfou. (May 2016). "Theano: A Python framework for fast computation of mathematical expressions." [Online]. Available: http://arxiv.org/abs/1605.02688
[165] M. Abadi *et al.*. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*.

[Online]. Available: https://www.tensorflow.org/
[166] M. B. Monagan *et al.*, *Maple 10 Programming Guide*. Waterloo, ON, Canada: Maplesoft, 2005.
[167] A. Z. Pinkus and S. Winitzki, "YACAS: A do-it-yourself symbolic algebra environment," in *Proc. Joint Int. Conf. Artif. Intell., Automated Reasoning, Symbolic Comput.* 2002, pp. 332–336. [Online]. Available: http://portal.acm.org/citation.cfm?id=648168.750655
[168] S. Wolfram, *The Mathematica Book*, 5th ed. Wolfram Media, Aug. 2003. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1579550223
[169] C. B. Moler, *Numerical Computing With MATLAB*. Philadelphia, PA, USA: SIAM, Jan. 2004. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0898715601
[170] M. Gordon, *From LCF to HOL: A short history*. Cambridge, MA, USA: MIT Press, 2000, pp. 169–185. [Online]. Available: http://portal.acm.org/citation.cfm?id=345868.345890
[171] R. Kabacoff, *R in Action*. Manning, 2010. [Online]. Available: http://www.manning.com/kabacoff
[172] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: SIAM, 2011.
[173] J. A. Rudin, "Implementation of polar format SAR image formation on the IBM cell broadband engine," in *Proc. High Perform. Embedded Comput. (HPEC)*, 2007.
[174] W. G. Carrara, R. S. Goodman, and R. M. Majewski, *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Norwood, MA, USA: Artech House, 1995.
[175] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th Annu. Symp. Found. Comput. Sci. (FOCS)*, 1999, p. 285. [Online]. Available: http://dl.acm.org/citation.cfm?id=795665.796479
[176] K. Yotov *et al.*, "A comparison of empirical and model-driven optimization," *Proc. IEEE*, vol. 93, no. 2, pp. 63–76, 2005.
[177] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using RAMs," *J. ACM*, vol. 56, no. 2, pp. 10:1–10:34, 2009.
[178] F. Serre, T. Holenstein, and M. Püschel, "Optimal circuits for streamed linear permutations using RAM," in *Proc. Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2016, pp. 215–223.
[179] B. Singer and M. M. Veloso, "Learning to construct fast signal processing implementations," *J. Mach. Learn. Res.*, vol. 3, pp. 887–919, Dec. 2002.
[180] T. Cui and F. Franchetti, "Random walk SAT solver: Program generation and autotuning," in *Proc. The Sixth Int. Workshop Autom. Perform. Tuning (iWAPT)*, 2010.
[181] P. A. Milder, M. Ahmad, J. C. Hoe, and M. Püschel, "Fast and accurate resource estimation of automatically generated custom DFT IP cores," in *Proc. FPGA*, 2006, pp. 211–220.
[182] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Discrete Fourier transform compiler: From mathematical representation to efficient hardware," Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. #CSSI-07-01, 2007.
[183] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "FFT compiler: From math to efficient hardware," in *Proc. IEEE Int. High Level Design Validation Test Workshop (HLDVT)*, Nov. 2007, pp. 137–139.
[184] G. Xu, T. M. Low, J. Hoe, and F. Franchetti, "Optimizing FFT resource efficiency on FPGA using high-level synthesis," in *Proc. IEEE HPEC*, 2017.
[185] J. P. Mendoza, M. Veloso, and R. Simmons, "Focused optimization for online detection of anomalous regions," in *Proc. Int. Conf. Robot. Autom. (ICRA)*, Hong Kong, Jun. 2014, pp. 3358–3363.
[186] A. Platzer and J.-D. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems," in *Proc. IJCAR*, 2008, pp. 171–178.
[187] T. Cui and F. Franchetti, "Optimized parallel distribution load flow solver on commodity multi-core CPU," in *Proc. IEEE High Perform. Extreme Comput. (HPEC)*, Sep. 2012, pp. 1–6.
[188] T. Cui and F. Franchetti, "Autotuning a random walk Boolean satisfiability solver," in *Proc. Int. Workshop Autom. Perform. Tuning (iWAPT)*, 2011.
[189] P. R. Luszczek *et al.*, "The HPC challenge (HPCC) benchmark suite," in *Proc. SC*, 2006, Art. no. 213.
[190] C.-K. Skylaris, P. D. Haynes, A. A. Mostofi, and M. C. Payne, "Introducing ONETEP: Linear-scaling density functional simulations on parallel computers," *J. Chem. Phys.*, vol. 122, no. 8, p. 084119, 2005.
[191] F. P. Russell, K. A. Wilkinson, P. H. J. Kelly, and C.-K. Skylaris, "Optimised three-dimensional Fourier interpolation: An analysis of techniques and application to a linear-scaling density functional theory code," *Comput. Phys. Commun.*, vol. 187, pp. 8–19, Feb. 2015.
[192] J. Rudin, "Implementation of polar format SAR image formation on the IBM cell broadband engine," in *Proc. High Perform. Embedded Comput. (HPEC)*, 2007.
[193] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inf. Theory*, vol. 13, no. 2, pp. 260–269, Apr. 1967.
[194] A. J. Viterbi, "A personal history of the Viterbi algorithm," *IEEE Signal Process. Mag.*, vol. 23, no. 4, pp. 120–142, Jul. 2006.
[195] P. Karn. (Aug. 2007). *FEC Library Version 3.0.1*. [Online]. Available: http://www.ka9q.net/code/fec/
[196] T. M. Low, Q. Guo, and F. Franchetti, "Optimizing space time adaptive processing through accelerating memory-bounded operations," in *Proc. High Perform. Extreme Comput. (HPEC)*, 2015, pp. 1–6.
[197] F. de Mesmay, F. Franchetti, Y. Voronenko, and M. Püschel, "Automatic generation of adaptive libraries for matrix-multiplication," in *Proc. Parallel Matrix Algorithms Appl. (PMAA)*, 2008.
[198] J. Kepner *et al.*, "Mathematical foundations of the GraphBLAS," in *Proc. High Perform. Extreme Comput. (HPEC)*, 2016, pp. 1–9.
[199] T.-M. Low, V. Rao, M. Lee, T. Popovici, F. Franchetti, and S. McMillan, "First look: Linear algebra-based triangle counting without matrix multiplication," in *Proc. High Perform. Extreme Comput. (HPEC)*, 2017, pp. 1–6.
[200] *The Coq Proof Assistant Reference Manual*, 2009.
[201] V. Zaliva and F. Franchetti, "Reasoning about sparse vectors for loops code generation," in *Proc. ICFP Student Res. Competition*, 2017.
[202] V. Zaliva and F. Franchetti, "Helix: A case study of a formal verification of high performance program generation," in *Proc. 6th ACM SIGPLAN Int. Workshop Funct. High-Perform. Comput. (FHPC)*, 2018.

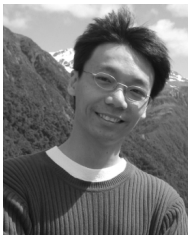
ABOUT THE AUTHORS



Franz Franchetti (Senior Member, IEEE) received the Dipl.-Ing. (M.Sc.) degree in technical mathematics and the Dr. techn. (Ph.D.) degree in computational mathematics from the Vienna University of Technology, Vienna, Austria, in 2000 and 2003, respectively.

Currently, he is a Professor at the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA. His research focuses on automatic performance tuning, program synthesis of mathematical kernels, and hardware/algorithm codesign.

Doru Thom Popovici (Student Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2018.



Tze Meng Low (Member, IEEE) received the B.S. degree in computer science and the B.S degree in economics in 2003 and the Ph.D degree in computer science in 2013 from the University of Texas Austin, Austin, TX, USA.

Currently, he is an Assistant Research Professor at the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research focuses on high-performance (hardware/software) implementations using analytical techniques and formal methods.

Richard M. Veras (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2017.

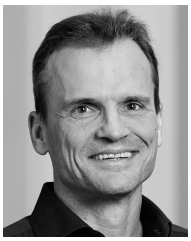
Currently, he is a Computer Systems Research Scientist at Louisiana State University, Baton Rouge, LA, USA.

Daniele G. Spampinato (Member, IEEE) received the Ph.D. degree in computer science from the Eidgenössische Technische Hochschule (ETH), Zürich, Switzerland, in 2017.

Currently, he is a Postdoctoral Researcher at the Department of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA, USA.

Jeremy R. Johnson (Senior Member, IEEE) received the B.A. degree in mathematics from the University of Wisconsin—Madison, Madison, WI, USA, in 1985, the M.S. degree in computer science from the University of Delaware, Newark, DE, USA, in 1988, and the Ph.D. degree in computer science from The Ohio State University, Columbus, OH, USA, in 1991.

Currently, he is a Professor of Computer Science and Electrical and Computer Engineering at Drexel University, Philadelphia, PA, USA. His research focuses on computer algebra, algebraic algorithms, program synthesis and verification, and high-performance computing and automated performance tuning.



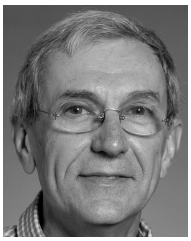
Markus Püschel (Senior Member, IEEE) received the Diploma (M.Sc.) in mathematics and the Ph.D. degree in computer science from the University of Karlsruhe, Karlsruhe, Germany, in 1995 and 1998, respectively.

Currently, he is a Professor of Computer Science at the Eidgenössische Technische Hochschule (ETH), Zürich, Switzerland, where he was the head of the department from 2013 to 2016. Before joining ETH in 2010, he was a Professor of Electrical and Computer Engineering at Carnegie Mellon University (CMU), Pittsburgh, PA, USA, where he still has an adjunct status.

Prof. Püschel received the Outstanding Research Award of the College of Engineering at Carnegie Mellon University and the main teaching awards from student organizations of both CMU and ETH.

James C. Hoe (Fellow, IEEE) received the B.S. degree in electrical engineering and computer science from the University of California Berkeley, Berkeley, CA, USA, in 1992 and the S.M. degree and the Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, in 1994 and 2000, respectively.

Currently, he is a Professor of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA, USA. His research interests include computer architecture, reconfigurable computing, and high-level hardware description and synthesis.



José M. F. Moura (Fellow, IEEE) received the Electrical Engineering degree, the M.Sc. degree in electrical engineering, and the D.Sc. degree in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA.

Currently, he is the Philip L. and Marsha Dowd University Professor at Carnegie Mellon University, Pittsburgh, PA, USA, with the Electrical and Computer Engineering Department. His research interests are in statistical signal and image processing and data science.

Prof. Moura is the 2018 President Elect of the IEEE, a Fellow of the American Association for the Advancement of Science (AAAS), a corresponding member of the Academia das Ciências of Portugal, and a member of the U.S. National Academy of Engineering.