CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Algebraic description and automatic generation of multigrid methods in SPIRAL

Matthias Bolten[1], Franz Franchetti[2*], Paul H. J. Kelly[3], Christian Lengauer[4] and Marcus Mohr[5]

[1]*Institute of Mathematics, University of Kassel, Germany*
[2]*Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA*
[3]*Department of Computing, Imperial College London, UK*
[4]*Faculty of Computer Science and Mathematics, University of Passau, Germany*
[5]*Department of Earth and Environmental Sciences, LMU Munich, Germany*

SUMMARY

SPIRAL is an autotuning, program generation and code synthesis system that offers a fully automatic generation of highly optimized target codes, customized for the specific execution platform at hand. Initially, SPIRAL was targeted at problem domains in digital signal processing, later also at basic linear algebra. We open SPIRAL up to a new, practically relevant and challenging domain: multigrid solvers. SPIRAL is driven by algebraic transformation rules. We specify a set of such rules for a simple multigrid solver with a Richardson smoother for a discretized square 2D Poisson equation with Dirichlet boundary conditions. We present the target code that SPIRAL generates in static single-assignment form and discuss its performance. While this example required no changes of or extensions to the SPIRAL system, more complex multigrid solvers may require small adaptations.

Received . . .

KEY WORDS: high-performance computing, PDE solvers, multigrid, SPIRAL, program transformation

1. INTRODUCTION

Many applications in science and engineering require the solution of partial differential equations (PDEs). Therefore, there is a huge demand for efficient solvers for discretized partial differential equations. Multigrid methods are widely used in this application context, as they are optimal for a large subclass of problems in the sense that the amount of work necessary to solve the discretized PDE grows only linearly with the number of unknowns. Multigrid methods date back at least to the early work by Fedorenko [1] and Bakhvalov [2] and were popularised in the West by the pioneering works of Brandt [3] and Hackbusch [4]. The theoretical background of multigrid methods is well studied [5–8] and a multitude of introductory articles [9, 10] as well as comprehensive overviews [11] exist.

Multigrid methods encounter a large number of variabilities at several levels: at the problem level (type of equation, boundary conditions, discretization scheme), at the algorithmic level (smoothers, transfer operators, cycling strategy), and at the platform level (processor types, memory hierarchy, communication network). To identify, code and manage the appropriate members of the huge space

*Correspondence to: Franz Franchetti, Department of Electrical and Computer Engineering, Carnegie Mellon University, Hamerschlag Hall A312, 5000 Forbes Ave, Pittsburgh, PA-15213, U.S.A.

of possible variants efficiently, we would like to use the autotuning, program generation and code synthesis system SPIRAL [12, 13] for the automatic generation of multigrid solvers. SPIRAL has been used successfully in other limited but well delineated domains, in particular signal processing. Here, we provide a proof of concept that the SPIRAL approach can effectively be applied to the domain of multigrid methods. At this point, we can demonstrate that straightforward, performance-competitive code of a full-fledged—albeit simple—multigrid solver can be generated completely automatically. Based on this, we claim that more sophisticated solvers and more sophisticated optimizations can follow.

We limit ourselves to constant coefficient problems in the unit square, resulting in systems whose system matrix is Toeplitz. The Toeplitz structure allows for a straightforward formulation of multigrid in the language of linear algebra that is suitable for implementation using SPIRAL. Multigrid methods for Toeplitz matrices have been studied by Fiorentino and Serra [14, 15] and by Serra-Capizzano [16].

Multigrid methods consist of a smoother and a multilevel representation of the solution at the finest level, and of the error on the coarser levels. This multilevel representation poses challenges for the implementation on modern computer architectures. On the one hand, the amount of work is relatively small compared to the number of memory accesses, i.e., multigrid has a low arithmetic intensity. On the other hand, the memory access pattern—while being highly structured—is usually not optimized automatically by the compiler. Thus, the use of a program generator for multigrid methods that incorporates domain-specific optimizations has potential, which we explore.

In summary, we make the following contributions:

- We show how the mathematical formulation of the multigrid method, using structured sparse matrix operators, can be used by SPIRAL to generate efficient code and that, by exploiting this representation, we enable a rich set of further optimizations.
- Concretely, we observe that the mathematical formulation of a multigrid solver with a Richardson smoother for a discretized square 2D Poisson equation with Dirichlet boundary conditions is precisely in the form that SPIRAL needs as input. In particular, the multigrid solver can be expressed as sparse matrix vector multiplication described as parameterized recursive decomposition.
- We propose, as a proof of concept, a set of equational transformation rules for a multigrid solver that enable SPIRAL to generate optimized static single-assignment target code. The resulting scalar C code is highly optimized and mirrors code properties and style found in high-performance computational mathematical libraries.
- We present performance results of this code on an Intel SandyBridge platform that demonstrate the potential of our proposed approach.

For this particular example, no changes or extensions had to be made to the SPIRAL system – the current formal language is sufficient to capture the example. For more complicated multigrid solvers, extensions may be required.

The paper is structured as follows. In Section 2 the SPIRAL program generation and code synthesis systems as well as the signal processing language SPL are described. Multigrid methods for structured grids are introduced in Section 3 and described using matrix notation. This matrix notation is used in Section 4 to derive an algebraic description that is suitable for an implementation in the functional language SPL. The necessary breakdown rules are provided in Section 5. The resulting expansion of a small example, as well as first performance results, can be found in Section 6. Finally, we conclude and provide a discussion of possible future work.

DOI: 10.1002/cpe

2. SPIRAL

2.1. Overview

The SPIRAL[†] autotuning, program generation and code synthesis system [12, 13] has demonstrated that it is possible to synthesize automatically high-quality implementations of mathematical kernels from high-level specifications [17].

So far, the problems targeted by SPIRAL have been in the domain of digital signal processing (DSP) algorithms, e.g., linear transforms [13] like the ubiquitous discrete Fourier transform, filters, the wavelet transform, and other DSP kernels such as Viterbi decoders [18], the image formation algorithm in synthetic aperture radar (SAR) [19], space-time adaptive processing (STAP), and components of JPEG 2000. A further domain explored is basic linear algebra [20]. Target platforms include mobile devices, multicore and manycore CPUs [21], SIMD vector extensions [22], the Cell BE processor [23], FPGAs [24], and GPUs. SPIRAL has demonstrated that it can be configured in minimal time for novel architectures and produce code competitive with the best human-produced code.

2.2. SPIRAL's Signal Processing Language (SPL)

SPIRAL's Signal Processing Language (SPL) [25] extends the Kronecker Product Formalism [26] that was initially developed to capture fast algorithms for signal processing transforms like the discrete Fourier transform [21], sine/cosine transforms, and the Walsh-Hadamard transform [13].

The key idea of SPL is to represent linear transforms as matrix-vector products, where the input and output is a vector and the operation is given by the matrix. Direct application of the matrix-vector product would require $O(n^2)$ operations for a problem of size n. However, for fast algorithms, the matrix can be factorized into $O(\log n)$ sparse matrices with $O(n)$ non-zero elements, leading to $O(n \log n)$ operations. Since the matrix factorization does not depend on the input or output vector, they are dropped in SPL, which makes SPL a *declarative*, *point-free* representation of algorithms for linear transforms.

Consider the 4-point discrete Fourier transform (DFT). As shown in (1), it can be represented as a 4×4 matrix of 4^{th} roots of unity and can be factorized into 4 sparse matrices with $O(n)$ entries (8 entries for size 4),

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (1)$$

Multiplication of the DFT matrix of size n, denoted by DFT_n, with a signal vector x of length n leads to the Fourier transform of the signal vector, $y = \mathrm{DFT}_n\, x$. The fast Fourier transform (FFT) is obtained by successive multiplication of the signal vector with the four factors of the DFT matrix, i.e., by performing four sparse vector multiplications instead of a single but dense one, and by applying this factorization recursively. The small size and simplicity of SPL enables a highly concise specification of the four factors. For the above example of a 4-point DFT, the SPL expression is given by

$$\mathrm{DFT}_{mn} \rightarrow (\mathrm{DFT}_m \otimes \mathrm{I}_n)\, \mathrm{T}^{mn}_n (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}^{mn}_m, \quad (2)$$

the famous general radix *Cooley-Tukey* factorization of a discrete Fourier transform of size $m \cdot n$ into m discrete Fourier transforms of size n, followed by the application of the twiddle factors, followed by n discrete Fourier transforms of size m [26]. For our purposes, the exact definitions of the components of (2) are not important. The key to (2) is that it is written as a so-called breakdown rule using "$\rightarrow$" instead of "$=$" and that the Kronecker product "$\otimes$", see below, is used to construct the sparse matrices from smaller DFTs and the identity matrix I_n. This expresses both repetition and provides recursion needed for efficient implementations [13].

[†] www.spiral.net

DOI: 10.1002/cpe

Multigrid methods can be viewed as parameterized linear operators, and can be expressed in SPL with minimal extensions to the language. The core contribution of this paper is to develop this SPL representation of a prototypical multigrid method. SPL is a machine-readable (LISP-like) language that mirrors the usual mathematical notation for matrices and vectors, and their associated operators and operations. We will use the mathematical notation to show the full description of multigid methods in SPL.

Let us now define the mathematical notation for SPL. An $m \times n$ matrix A is denoted by $A_{m\times n}$, but the subscript is dropped whenever the size is clear from the context. We write I_n for the $n \times n$ identity matrix. 0_n is the $n \times n$ zero matrix. The *Kronecker product* of matrices $A_{m\times n}$ and $B_{p\times q}$ is defined as

$$A_{m\times n} \otimes B_{p\times q} = [a_{k,\ell}B], \quad \text{for } A_{m\times n} = [a_{k,\ell}] \ .$$

It replaces every entry $a_{k,\ell}$ of $A_{m\times n}$ by a block $a_{k,\ell}B$ resulting in a matrix of dimensions $mp \times nq$. Most important are the cases in which A or B is the identity. $\mathrm{Tridiag}_n(a, b, c)$ is a tridiagonal matrix given by

$$\mathrm{Tridiag}_n(a,b,c) = \begin{bmatrix} b & c & & & \\ a & b & c & & \\ & \ddots & \ddots & \ddots & \\ & & a & b & c \\ & & & a & b \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

In SPIRAL, a tridiagonal matrix is represented as a zero-extended filter [13], which is a signal processing construct already available in SPIRAL and which, thus, can be reused out of context. For two compatible matrices A and B, we denote *horizontal stacking* by

$$\left[\ A \ \middle|\ B\ \right]$$

and *vertical stacking* by

$$\left[\frac{A}{B}\right].$$

We denote the n-dimensional canonical basis vector with a 1 at the ith location by e_i^n. A *gather matrix* $\mathrm{G}_{b,s}^{n\times N}$ is an $n \times N$ matrix and gathers data from the input vector $x = (x_0, \ldots, x_{N-1})^T \in \mathbb{R}^N$ starting at base b with stride s, i.e., resulting in the vector

$$\left(x_b, x_{b+s}, \ldots, x_{b+(n-1)s}\right)^T \in \mathbb{R}^n.$$

A *scatter matrix* $\mathrm{S}_{b,s}^{N\times n}$ performs the inverse operation of $\mathrm{G}_{b,s}^{n\times N}$ and scatters the entries x_i of a vector $x = (x_i)_{0\le i<n} \in \mathbb{R}^n$ into a vector $y \in \mathbb{R}^N$ at locations $b + is,\ 0 \le i < n$ while setting all other elements of y to 0. Formally,

$$\mathrm{S}_{b,s}^{N\times n} = \left[\ e_b^N \ \middle|\ e_{b+s}^N \ \middle|\ \cdots \ \middle|\ e_{b+(n-1)s}^N\ \right] \quad \text{and} \quad \mathrm{G}_{b,s}^{n\times N} = \left(\mathrm{S}_{b,s}^{N\times n}\right)^T.$$

Using the usual arithmetic operations for matrices, the matrices defined above can be combined to form *matrix formulas*. In particular, we use the operations

$$A + B, \quad A \cdot B, \quad A \otimes B, \quad \text{and} \quad \prod_{i=0}^{k-1} A_i$$

for compatible matrices A, B, and A_i. Such matrix formulas represent a data flow graph that SPIRAL translates into highly efficient code.

In SPIRAL, algorithms are defined via breakdown rules. A *breakdown rule* breaks a *non-terminal* (a symbol that is not yet fully, i.e., *terminally*, expanded) into smaller non-terminals of equal or different type and provides a matrix formula that expresses the initial non-terminal as an SPL expression parameterized by the smaller non-terminals. *Termination rules* explain a non-terminal in

DOI: 10.1002/cpe

terms of an SPL formula. A *rewriting system* expands a *specification* given as a non-terminal using breakdown rules and termination rules until all non-terminals are *terminated*, i.e., nothing is left to be expanded. By recursively substituting the SPL expressions for the respective non-terminals, SPIRAL derives an SPL formula for the initial non-terminal (the specification). This process may require a backtracking search, as more than one breakdown rule may match a given non-terminal, but not all non-terminals can be terminated. An example of such a breakdown rule is the Kronecker product identity

$$A_n \otimes B_n = \big(A_n \otimes I_n\big) \cdot \big(\mathrm{I}_n \otimes B_n\big).$$

SPL defines a non-terminal $\mathrm{Tensor}(.,.)$ and a breakdown rule

$$\mathrm{Tensor}(A_n, B_n) \to \big(A_n \otimes I_n\big) \cdot \big(\mathrm{I}_n \otimes B_n\big). \tag{3}$$

In this case, SPIRAL first expands the child non-terminals A_n and B_n completely via their respective rules and eventually substitutes the resulting expressions for A_n and B_n into (3) to build the final SPL expression for $\mathrm{Tensor}(A_n, B_n) = A_n \otimes B_n$.

Next, we explain how SPL (matrix) formulas are translated to code by SPIRAL's *SPL compiler* [25]. Table I shows how to translate matrix formula fragments to basic sequential loop code. These translation rules are applied recursively until the complete formula is translated to code. This is only the first step in code translation and optimization. SPIRAL then applies loop optimizations [27] and basic block optimizations [25] to obtain the final code. However, these additional optimizations are beyond the scope of this paper.

3. MULTIGRID METHODS

Multigrid methods are used to solve linear systems of the form

$$Ax = b, \quad A \in \mathbb{R}^{n \times n},\ x, b \in \mathbb{R}^n, \tag{4}$$

usually arising from the discretization of a partial differential equation (PDE). Unlike direct solvers like Gaussian elimination, multigrid methods are iterative: starting from an initial guess $x^{(0)}$, they yield a sequence of approximations $x^{(k)}$, $k = 0, 1, \ldots$. The efficacy of multigrid methods relies on the following observation: if a few iterations of a simple iterative method, like Gauss-Seidel, are applied to a linear system arising from the discretization of an elliptic PDE, the error is not reduced very efficiently, but it is smooth. As a consequence, this error can be represented very well using fewer degrees of freedom on a coarser discretization mesh, and so can be the residual that is given by $r = b - Ax$. The error is given as the solution of the system

$$Ae = r$$

and, as the error is smooth and well represented on a coarser discretization mesh, this equation can be solved on the coarser grid. This reduces the number of unknowns substantially: in the case of geometric multigrid methods, like the ones considered in this work, a factor of 4 in 2D and 8 in 3D is common. After the coarse-grid solution has been obtained, the approximation of the error is transferred back to the fine grid and the current approximation of the solution is updated by adding the interpolated coarse-grid error. To get rid of high-frequency error components that are introduced by this update, a second round of smoothing steps is performed. This procedure can be applied recursively until the number of unknowns is small enough to solve the system either directly or accurately enough using a simple iterative method. In contrast to many other iterative methods, multigrid methods are optimal in the sense that the convergence rate, i.e., the decay of the norm of the residual, is linear and independent of the system size for a large class of problems. This class includes, e.g., finite difference or finite element discretizations of elliptic PDEs defined on sufficiently smooth domains. As these discretizations result in sparse system matrices, the cost of each iteration is $\mathcal{O}(n)$. As a result, the overall cost of reducing the residual by a given factor is linear in the data size and, thus, optimal.

DOI: 10.1002/cpe

Table I. From matrix formulas to code. The subscript of matrices A and B specifies the (square) matrix size. `x[b:s:e]` denotes the subvector of x starting at `b`, ending at `e`, extracted at stride `s`.

Matrix formula	Matlab-style pseudo code
$y = \mathrm{I}_n\, x$	`for (i=0; i<n; i++)` `  y[i] = x[i];`
$y = 0_n\, x$	`for (i=0; i<n; i++)` `  y[i] = 0.0;`
$y = \mathrm{G}_{b,s}^{n\times N}\, x$	`for (i=0; i<n; i++)` `  y[i] = x[b+i*s];`

$y = \mathrm{S}_{b,s}^{N\times n}\, x$	`y = 0.0;`
	`for (i=0; i<n; i++)`
	`  y[b+i*s] = x[i];`

$y = \mathrm{Tridiag}_n(a,b,c)x$	`y[0] = b*x[0]+c*x[1];`
	`for (i=1; i<n-1; i++)`
	`  y[i] = a*x[i-1]+b*x[i]+c*x[i+1];`
	`y[n-1] = a*x[n-2]+b*x[n-1];`

$y = (A_n + B_n)x$	`y[0:1:n-1] = A(x[0:1:n-1]) + B(x[0:1:n-1]);`
$y = (A_n \cdot B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1]);`
$y = [A_n \vert B_n]\, x$	`y[0:1:n-1] = A(x[0:1:n-1]) + B(x[n:1:2*n-1]);`
$y = \begin{bmatrix} A_n \\ B_n \end{bmatrix} x$	`y[0:1:n-1] = A(x[0:1:n-1]);` `y[n:1:2*n-1] = B(x[0:1:n-1]);`

$y = \left(\prod_{i=0}^{k-1} A_i\right) x$	`y = x;`
	`for (i=0; i<k; i++)`
	`  {x = y; y = A(i, x);}`

$y = (\mathrm{I}_m \otimes A_n)x$	`for (i=0; i<m; i++)` `  y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1]);`
$y = (A_m \otimes \mathrm{I}_n)x$	`for (i=0; i<n; i++)` `  y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n]);`

The efficient implementation of multigrid methods is tedious as it requires knowledge of the target platform like cache sizes, number of registers, etc. Insofar, the implementation of multigrid methods, in a way, optimized for the target platform, shares the same problems and challenges as algorithms in signal processing, e.g., the fast Fourier transform, to which code generation techniques and the automated optimization of the implementations have been applied before.

In order to reuse previously known techniques in the signal processing language SPL [25–27] as part of SPIRAL [12,13,17], a description of multigrid methods in the form of linear transformations is needed. We provide this description and a prototypical implementation.

While multigrid methods can be applied to a large variety of problems, here, we limit ourselves to the simplest possible example, the Poisson equation with Dirichlet boundary conditions in the unit cube of arbitrary dimension, i.e., the partial differential equation

$$\begin{aligned} -\Delta u(x) &= f(x), \quad x \in \Omega := [0,1]^d, \\ u(x) &= 0, \quad x \in \partial\Omega. \end{aligned} \tag{5}$$

DOI: 10.1002/cpe

In two dimensions, discretization on an $m \times n$ grid with 2nd-order finite differences yields the system matrix

$$A = I_m \otimes \frac{1}{(n+1)^2} \operatorname{Tridiag}_n(-1, 2, -1) + \frac{1}{(m+1)^2} \operatorname{Tridiag}_m(-1, 2, -1) \otimes I_n.$$

Note that we have here eliminated the boundary values from the system, as is standard procedure in the Dirichlet case. Thus, our $m \times n$ grid encompasses only interior nodes. For a given right-hand side $b \in \mathbb{R}^{m \cdot n}$, which is obtained by sampling the function f at equidistant points on the two-dimensional grid, the task is to solve (4) and obtain an approximate solution x of (5). The generalization to d-dimensional cubes is straightforward.

3.1. Multigrid methods

As introduced earlier, multigrid methods make use of the following basic observation. If an iterative method like the Gauss-Seidel method is applied to a linear system arising from the discretization of a partial differential equation and the development of the error is observed, one notices that, while the error is not reduced much, it is smoothed out after a few iterations. A smooth error is well represented on a coarser grid. Therefore, it is reasonable to calculate the residual of the current approximation x

$$r = b - Ax$$

and solve the residual equation

$$Ae = r$$

approximately on the coarser grid. The approximate error is then used to update the current guess on the fine grid. In multigrid terminology, this step is denoted as *coarse-grid correction*. Subsequently, an iterative method can be used to smooth the current approximation again. To apply this idea, in addition to an iterative method as a smoother, we also need grid transfer operators and a coarse representation of our problem. Two grid transfer operators are required: a restriction to transfer the residual from the fine to the coarse grid and a prolongation that transfers the approximate error from the coarse to the fine grid. For the latter, usually, interpolation routines are used. As restriction, either (weighted) injection or the adjoint of the prolongation are common. The coarse representation of the system matrix that represents the problem is often chosen as a coarse rediscretization of the underlying problem, or by using variational principles. In most cases, the coarse problem is still too large to be solved directly, so the procedure is applied recursively until the problem is small enough to be solved directly. This results in a multigrid method. In summary, the multigrid method $\mathcal{MG}_i$, whose index i represents the level with $\mathcal{MG}_0$ denoting the original system, is given by Algorithm 1.

Algorithm 1 Multigrid cycle $x_{n_i} = \mathcal{MG}_i(x_{n_i}, b_{n_i})$

$x_{n_i} \leftarrow \mathcal{S}_i^{\nu_1}(x_{n_i}, b_{n_i})$ {pre-smoothing}
$r_{n_i} \leftarrow b_{n_i} - A_i x_{n_i}$ {calculate residual}
$r_{n_{i+1}} \leftarrow R_i r_{n_i}$ {restrict residual}
$e_{n_{i+1}} \leftarrow 0$ {initialize coarse-level error approximation}
if $i + 1 = l_{\max}$ **then**
  $e_{n_{l_{\max}}} \leftarrow A_{l_{\max}}^{-1} r_{n_{l_{\max}}}$ {direct solve}
else
  for $j = 1, \ldots, \gamma$ **do**
    $e_{n_{i+1}} \leftarrow \mathcal{MG}_{i+1}(e_{n_{i+1}}, r_{n_{i+1}})$ {recursive call}
  end for
end if
$e_{n_i} \leftarrow P_i e_{n_{i+1}}$ {prolongate coarse-level error approximation}
$x_{n_i} \leftarrow x_{n_i} + e_{n_i}$ {update current approximate solution}
$x_{n_i} \leftarrow \tilde{\mathcal{S}}_i^{\nu_2}(x_{n_i}, b_{n_i})$ {post-smoothing (possibly with a different smoother)}

DOI: 10.1002/cpe

In the simplest case, we restrict ourselves to grid sizes m and n, each some power of 2 minus 1, and to a coarsening by a factor of 2. If $m = n$ this allows the system to be coarsened to one single unknown, allowing for an easy direct solution of the coarsest 1×1 "system". For our problem (5), this means that the system matrix $A = A_0$ on the finest grid is given by

$$A_0 = \mathrm{I}_{(2^k-1)^2} \otimes \frac{1}{(2^\ell)^2} \operatorname{Tridiag}_{2^\ell-1}(-1,2,-1) + \frac{1}{(2^k)^2} \operatorname{Tridiag}_{2^k-1}(-1,2,-1) \otimes \mathrm{I}_{2^\ell-1}\,.$$

In the following, we assume that $m = n = 2^{\ell_{\max}} - 1$, i.e.,

$$A_0 = \mathrm{I}_{2^{\ell_{\max}}-1} \otimes \frac{1}{(2^{\ell_{\max}})^2} \operatorname{Tridiag}_{2^{\ell_{\max}}-1}(-1,2,-1) + \frac{1}{(2^{\ell_{\max}})^2} \operatorname{Tridiag}_{2^{\ell_{\max}}-1}(-1,2,-1) \otimes \mathrm{I}_{2^{\ell_{\max}}-1}\,.$$

3.2. Smoother

Often splitting methods are used as smoothers. This includes Richardson, damped Jacobi or successive over-relaxation (SOR). One of the simplest methods, the Richardson method, is given by

$$x^{(k+1)} = \mathcal{S}_{\mathrm{Richardson},\omega}(x^{(k)}, b) = x^{(k)} + \omega \left(b - Ax^{(k)}\right),$$

where ω is chosen suitably. Due to its simplicity, this is our method of choice for our proof of concept.

3.3. Restriction

The simplest restriction operator is the injection. Injection assigns to every coarse grid point the value at the corresponding fine grid point. In one dimension, this means taking every other grid point, starting with the second one, and can be expressed as

$$R_{\mathrm{inj}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ & & \vdots & & & \ddots & & \vdots & \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{bmatrix} \in \mathbb{R}^{\frac{n-1}{2} \times n}.$$

Higher-dimensional versions are obtained by taking Kronecker products of the one-dimensional restriction. Note that the injection matrix is a gather matrix,

$$R_{\mathrm{inj}} = \mathrm{G}_{1,2}^{(n-1)/2 \times n}\,.$$

The injection matrix also plays a vital role in defining the other grid transfer operators. For instance, the full-weighting operator that is the transpose of the linear interpolation times $\frac{1}{2}$ in one dimension is given by

$$R_{\mathrm{fw}} = R_{\mathrm{inj}} \cdot \operatorname{Tridiag}_n \left(\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\right).$$

Again, higher-dimensional variants are obtained by taking the d-fold Kronecker product of the one-dimensional variant.

3.4. Interpolation

An example interpolation operator is the linear interpolation, given in one dimension by

$$P_{\mathrm{lin}} = \operatorname{Tridiag}_n \left(\frac{1}{2}, 1, \frac{1}{2}\right) \cdot R_{\mathrm{inj}}^T.$$

Also, in the interpolation case, the corresponding higher-dimensional versions can be obtained by forming Kronecker products as in the restriction case.

DOI: 10.1002/cpe

3.5. Coarse-grid operator

As mentioned before, in the simplest case, the coarse-grid operator is just a rediscretization of the original continuous problem. In our case, this means that the operators are given by

$$A_i = \mathrm{I}_{2^{k-i}-1} \otimes \frac{1}{2^{k-i}} \operatorname{Tridiag}_{2^{k-i}-1}(-1,2,-1) + \frac{1}{2^{k-i}} \operatorname{Tridiag}_{2^{k-i}-1}(-1,2,-1) \otimes \mathrm{I}_{2^{k-i}-1}, \quad i = 0,1,\ldots,k-1.$$

4. ALGEBRAIC DESCRIPTION

The basic operations used in Algorithm 1 are, in principle, not linear but affine, involving not only the current approximate solution but also the right-hand side. To overcome this limitation, the approximation is stacked on top of the right-hand side and the transformations are expressed such that the right-hand side is not changed.

The grid transfer operators do not act on the current approximation and the right-hand side but rather on the residual or the approximation of the error on the coarse grid.

4.1. Multigrid Method

Following the description in Section 3 and Algorithm 1 we can represent a multigrid method as an operator

$$MG_\ell = S_\ell^{\nu_2} \cdot CGC_\ell \cdot S_\ell^{\nu_1}, \tag{6}$$

composed of three components. The application of the smoothing operator S_ℓ in ν_1 pre-smoothing and ν_2 post-smoothing steps and a coarse-grid correction CGC_ℓ in between. Following our block approach, a single application of the operator MG_ℓ, i.e., a single multigrid cycle, will map a pair consisting of old approximate solution and right-hand side onto a pair consisting of a new, improved approximate solution and the right-hand side

$$\begin{pmatrix} x^{(k+1)} \\ b \end{pmatrix} = MG_\ell \begin{pmatrix} x^{(k)} \\ b \end{pmatrix}.$$

The subscript ℓ is to remind us that one of the central multigrid ideas is to employ the same solution approach recursively for the problems on the coarser grids. Below, we derive an explicit algebraic representation of MG_ℓ by considering its individual components.

4.2. Smoother

The Richardson smoother is described by the following block matrix

$$\begin{pmatrix} x^{(k+1)} \\ b \end{pmatrix} = \underbrace{\begin{bmatrix} I - \omega A & \omega I \\ 0 & I \end{bmatrix}}_{=:\, S_{\mathrm{Richardson},\omega}} \cdot \begin{pmatrix} x^{(k)} \\ b \end{pmatrix}.$$

4.3. Coarse-Grid Correction

Let us derive a representation of the coarse-grid correction step of the multigrid method as a linear operator. This is a little more sophisticated than with the smoother. Thus, we first perform it for the case of a hierarchy consisting only of two grid levels.

4.3.1. Two-level case In the two-level case, the coarse-grid correction calculates the residual by

$$r = b - Ax.$$

Then, the residual is restricted to obtain the coarse residual, i.e.,

$$r_c = Rr$$

DOI: 10.1002/cpe

and the coarse error is obtained as the solution of the system

$$A_c e_c = r_c. \tag{7}$$

The coarse error is then prolongated by

$$e = P e_c$$

and the current approximation is updated by adding this approximation

$$x = x + e.$$

Combining everything yields

$$x = x + PA_c^{-1}R(b - Ax) = (I - PA_c^{-1}RA)x + PA_c^{-1}Rb. \tag{8}$$

Following the above convention, we obtain

$$\begin{pmatrix} x^{(k+1)} \\ b \end{pmatrix} = \underbrace{\begin{bmatrix} I - PA_c^{-1}RA & PA_c^{-1}R \\ 0 & I \end{bmatrix}}_{=:\,CGC} \cdot \begin{pmatrix} x^{(k)} \\ b \end{pmatrix}. \tag{9}$$

4.3.2. Multilevel case In a real multigrid method, with a hierarchy consisting of more than two levels, the solution of the linear system (7) is replaced by a recursive call to the multigrid method applied to the coarse system of smaller size.

Let CGC_ℓ be the coarse-grid correction at level ℓ within operator (6). The application of the inverse problem matrix A_c^{-1} on the coarser level to solve (7) will now be replaced by a recursive invocation of multigrid. Effectively the inverse is approximated by

$$A_c^{-1} \approx \begin{bmatrix} \mathrm{I} & 0 \end{bmatrix} \cdot MG_{k+1}^{\gamma} \cdot \begin{bmatrix} 0 \\ \mathrm{I} \end{bmatrix}.$$

Here, γ is the number of iterations used to solve the coarse system, cf. Algorithm 1. A choice of $\gamma = 1$ corresponds to a V-cycle, while $\gamma = 2$ corresponds to a W-cycle. Note that, in the coarse-grid correction scheme, it is essential that one always starts with a zero intial guess for the approximate error on the coarser grids. Hence the top zero block in the vertically stacked operator on the right.

With n_ℓ for the system size at level ℓ, we obtain

$$CGC_i = \begin{bmatrix} I_{n_i} - P_i \begin{bmatrix} \mathrm{I}_{n_{i+1}} & 0 \end{bmatrix} MG_{i+1}^{\gamma} \begin{bmatrix} 0 \\ \mathrm{I}_{n_{i+1}} \end{bmatrix} R_i A_i & P_i \begin{bmatrix} \mathrm{I}_{n_{i+1}} & 0 \end{bmatrix} MG_{i+1}^{\gamma} \begin{bmatrix} 0 \\ \mathrm{I}_{n_{i+1}} \end{bmatrix} R_i \\ 0 & \mathrm{I}_{n_i} \end{bmatrix},$$

for $i = 0, 1, \ldots, k-3$, where k is the number of levels, and

$$CGC_{k-2} = \begin{bmatrix} \mathrm{I}_{n_{k-2}} - P_{k-2}A_{k-1}^{-1}R_{k-2}A_{k-2} & P_{k-2}A_{k-1}^{-1}R_{k-2} \\ 0 & \mathrm{I}_{n_{k-2}} \end{bmatrix}.$$

Although these definitions contain two applications of the specified number of iterations of the multigrid method at the next level, the applications can be combined as in (8).

5. GENERATING MULTIGRID METHODS USING SPIRAL

In this section, we describe the SPIRAL formalization as SPL breakdown rules of a multigrid solver for an $n \times n$ discretized 2D 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. Table II summarizes the breakdown rules in (10)–(20).

DOI: 10.1002/cpe

Table II. SPIRAL SPL breakdown rules for a multigrid solver $\mathrm{MGSolvePDE}_{n,\omega,r,m}$ for a $n \times n$ discretized 2D 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.

$$\mathrm{MGSolvePDE}_{n,\omega,r,m} \to [\mathrm{I}_{n^2} \mid 0_{n^2}] \cdot \left(\prod_{i=0}^{m-1} \mathrm{MGCycle}_{n,\omega,r} \right) \cdot \begin{bmatrix} 0_{n^2} \\ \mathrm{I}_{n^2} \end{bmatrix} \tag{10}$$

$$\mathrm{MGCycle}_{n,\omega,r} \to \mathrm{CGC}_{n,\omega,r} \cdot \mathrm{Richardson}_{n,\omega,r} \tag{11}$$

$$\mathrm{CGC}_{n,\omega,r} \to \begin{bmatrix} \mathrm{CoarseError}_{n,\omega,r} \\ 0_{n^2} \mid \mathrm{I}_{n^2} \end{bmatrix} \tag{12}$$

$$\mathrm{CoarseError}_{n,\omega,r} \to \mathrm{Interpolate}_n \cdot \mathrm{Scatter}_n \cdot \mathrm{Solve}_{n,\omega,r} \cdot \mathrm{Gather}_n \cdot \mathrm{Residual}_n \tag{13}$$

$$\mathrm{Interpolate}_n \to \mathrm{Tridiag}_n\left(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2\right) \otimes \mathrm{Tridiag}_n\left(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2\right) \tag{14}$$

$$\mathrm{Scatter}_n \to \mathrm{S}_{1,2}^{n \times (n-1)/2} \otimes \mathrm{S}_{1,2}^{n \times (n-1)/2} \tag{15}$$

$$\mathrm{Solve}_{n,\omega,r} \to \begin{cases} \frac{1}{4}\,\mathrm{I}_1, & n = 1 \\ \left[\mathrm{I}_{((n-1)/2)^2} \mid 0_{((n-1)/2)^2}\right] \cdot \mathrm{MGCycle}_{(n-1)/2,\omega,r} \cdot \begin{bmatrix} 0_{((n-1)/2)^2} \\ \mathrm{I}_{((n-1)/2)^2} \end{bmatrix}, & n > 1 \end{cases} \tag{16}$$

$$\mathrm{Gather}_n \to \mathrm{G}_{1,2}^{(n-1)/2 \times n} \otimes \mathrm{G}_{1,2}^{(n-1)/2 \times n} \tag{17}$$

$$\mathrm{Residual}_n \to [\mathrm{Tridiag}_n(1, -2, 1) \otimes \mathrm{I}_n + \mathrm{I}_n \otimes \mathrm{Tridiag}_n(1, -2, 1) \mid \mathrm{I}_{n^2}] \tag{18}$$

$$\mathrm{Richardson}_{n,\omega,r} \to \prod_{i=0}^{r-1} \begin{bmatrix} \mathrm{ResidueLaplace}_{n,\omega} & \omega\,\mathrm{I}_{n^2} \\ 0_{n^2} & \mathrm{I}_{n^2} \end{bmatrix} \tag{19}$$

$$\mathrm{ResidueLaplace}_{n,\omega} \to \mathrm{Tridiag}_n(\omega, 0.5 - 2\omega, \omega) \otimes \mathrm{I}_n + \mathrm{I}_n \otimes \mathrm{Tridiag}_n(\omega, 0.5 - 2\omega, \omega) \tag{20}$$

The entry point of the rule system is the SPL non-terminal $\mathrm{MGSolvePDE}_{n,\omega,r,m}$ that represents a specification of the multigrid solver detailed above. In fact, it corresponds to m successive applications of MG_0, i.e., the multigrid operator of (6) for the finest level. (10) translates $\mathrm{MGSolvePDE}_{n,\omega,r,m}$ to m multigrid *cycles* $\mathrm{MGCycle}_{n,\omega,r}$. Next, (11) translates a multigrid cycle to a Richardson smoother $\mathrm{Richardson}_{n,\omega,r}$ followed by a coarse-grid correction $\mathrm{CGC}_{n,\omega,r}$. Note that the order in the formula is reversed, as vectors are applied from right to left. We restrict ourselves here to pre-smoothing. However, this is only for simplicity, not a conceptual limitation.

(12) expresses the coarse-grid correction $\mathrm{CGC}_{n,\omega,r}$ in terms of the coarse error operator $\mathrm{CoarseError}_{n,\omega,r}$. (13) explains $\mathrm{CoarseError}_{n,\omega,r}$ in terms of $\mathrm{Interpolate}_n$, $\mathrm{Scatter}_n$, $\mathrm{Solve}_{n,\omega,r}$, Gather_n, and $\mathrm{Residual}_n$. Thus, (13) encodes the choice of the restriction operator to be *injection*. The choice of *linear interpolation* is encoded in (14), which terminates the non-terminal $\mathrm{Interpolate}_n$. (17) and (15) encode 2D gathering and scattering of every other lattice point in a 2D grid.

The non-terminal $\mathrm{Solve}_{n,\omega,r}$ captures the multigrid recursion. It encodes the base case of $n = 1$ in which a direct solve is applied, and recurses back to the multigrid cycle $\mathrm{MGCycle}_{n,\omega,r}$ for $n > 1$. The non-terminal $\mathrm{Residual}_n$ computes the residual. The non-terminal $\mathrm{Richardson}_{n,\omega,r}$ captures the Richardson smoother with parameter ω and r iterations. It is reduced to the Laplace residue by (19), which is finally reduced to tridiagonal matrices in (20).

The non-terminals $\mathrm{Residual}_n$, $\mathrm{Interpolate}_n$, and $\mathrm{ResidueLaplace}_{n,\omega}$ encode the problem type to be a Poisson equation and the Dirichlet boundary condition via the entries of the tridiagonal matrix.

The rule set (10)–(20) in Table II completely specifies non-terminal $\mathrm{MGSolvePDE}_{n,\omega,r,m}$ in terms of basic SPIRAL SPL objects and operators. This is all we need to generate code for $\mathrm{MGSolvePDE}_{n,\omega,r,m}$, as the next section shows. However, the rule set is highly specialized to the particular problem (Poisson equation with Dirichlet boundary condition) and multigrid method choices (Richardson smoother and injection as restriction).

The description of a multigrid solver for an $n \times n$ discretized 2D Poisson equation, given by (10)–(20) in Table II, is independent of the problem size and can be applied to arbitrarily large problems. However, SPIRAL's code generation provides three distinct use cases. In order

DOI: 10.1002/cpe

of growing complexity, SPIRAL can (1) generate small fixed-size kernels, often called *codelets*, (2) large but fixed-size functions that implement a particular problem size very well using SIMD vector instructions and parallelism, and (3) a general-size library which allows the choice of some configuration parameters and, in particular, the problem size at run time. Step (2) requires loop-level optimizations and step (3) requires delaying of problem decompositions from compile time to runtime and support for online autotuning, which makes these cases much harder than step (1). The inclusion of any new domain in SPIRAL must proceed from case 1 via case 2 to case 3.

Clearly, for practical multigrid solvers, case 3 is the most desirable but also hardest case, and both case 2 and case 3 are subject of future work. However, even case 1, as presented here, has immediate practical use, as evidenced by previous successes in the domains of numerical linear algebra [28], FFTs [29,30], and stencil computations [31]: by generating a large set of appropriately chosen small, fixed-size kernels that implement particular algorithmic choices and that are very well optimized, the following technique from the domain of the *automatic performance tuning* community can be leveraged: a recursive algorithmic skeleton breaks larger problems recursively down into smaller problems, in a *divide and conquer* approach. At some point, the recursion is unrolled and special base cases for both the divide and conquer phase, that implement a small number of recursion levels, are used. These kernels are generated automatically and make up most of the run time. By searching for and selecting the suitable breakdown strategy, *performance portability* (i.e., good performance across a wide range of platforms) is achieved, and the human tuning effort is minimized.

6. EXAMPLES AND RESULTS

6.1. Multigrid Cycle

First, we show SPIRAL-generated code for a 3×3 point multigrid cycle. The code is in standard C with data type `double`. It is fully unrolled and uses pointer dereferencing / pointer arithmetic for all array accesses. All standard SPIRAL back-end optimizations [25] have been applied: array scalarization, copy propagation, common subexpression elimination, algebraic simplification, and constant folding. The code is in single static-assignment (SSA) form, but assignments can be large expressions, i.e., we do not convert to three-address code.

We start with the specification of the code,

$$\mathrm{MGCycle}_{3,0.125,1}, \tag{21}$$

which tells SPIRAL to generate code for a multigrid cycle on a grid of 3×3 points for a 2D Poisson equation with Dirichlet boundary conditions. The parameters $\omega = 0.125$ and $r = 1$ for the Richardson smoother are fixed (but could be run-time parameters). Further, the restriction is injection. SPIRAL first expands specification (21) using (11) and (12), leading to

$$\begin{aligned} &\to \mathrm{CGC}_{3,0.125,1} \cdot \mathrm{Richardson}_{3,0.125,1} \\ &\to \begin{bmatrix} \mathrm{CoarseError}_{3,0.125,1} \\ [0_9 \,|\, \mathrm{I}_9] \end{bmatrix} \cdot \begin{bmatrix} \mathrm{ResidueLaplace}_{3,0.125} & 0.125\,\mathrm{I}_9 \\ 0_9 & \mathrm{I}_9 \end{bmatrix}. \end{aligned} \tag{22}$$

Next, (22) is expanded using (13) and (20), yielding

$$\begin{aligned} \to &\begin{bmatrix} \mathrm{Interpolate}_3 \cdot \mathrm{Scatter}_3 \cdot \mathrm{Solve}_{3,0.125,1} \cdot \mathrm{Gather}_3 \cdot \mathrm{Residual}_3 \\ [0_9 \,|\, \mathrm{I}_9] \end{bmatrix} \\ &\cdot \begin{bmatrix} \mathrm{Tridiag}_3(0.125, 0.25, 0.125) \otimes \mathrm{I}_3 + \mathrm{I}_3 \otimes \mathrm{Tridiag}_3(0.125, 0.25, 0.125) & 0.125\,\mathrm{I}_9 \\ 0_9 & \mathrm{I}_9 \end{bmatrix}. \end{aligned} \tag{23}$$

In a final step, (23) is expanded using (14)–(18). We define the tridiagonal matrices

$$\begin{aligned} T_1 &= \mathrm{Tridiag}_3\left(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2\right) \\ T_2 &= \mathrm{Tridiag}_3(1, -2, 1) \\ T_3 &= \mathrm{Tridiag}_3(0.125, 0.25, 0.125) \end{aligned}$$

DOI: 10.1002/cpe

and obtain the final expansion

$$\mathrm{MGCycle}_{3,0.125,1} \rightarrow \begin{bmatrix} T_1 \otimes T_1 \cdot \mathrm{S}_{1,2}^{3\times1} \otimes \mathrm{S}_{1,2}^{3\times1} \cdot \frac{1}{4}\,\mathrm{I}_1, \cdot\, \mathrm{G}_{1,2}^{1\times3} \otimes \mathrm{G}_{1,2}^{1\times3} \cdot [T_2 \otimes \mathrm{I}_3 + \mathrm{I}_3 \otimes T_2 \mid \mathrm{I}_9] \\ 0_9 \mid \mathrm{I}_9 \end{bmatrix} \cdot \begin{bmatrix} T_3 \otimes \mathrm{I}_3 + \mathrm{I}_3 \otimes T_3 & 0.125\,\mathrm{I}_9 \\ 0_9 & \mathrm{I}_9 \end{bmatrix}. \quad (24)$$

SPL formula (24) is an 18×18 matrix and represents the specification (non-terminal) $\mathrm{MGCycle}_{3,0.125,1}$ as a fully expanded (terminated) SPL formula that contains only basic SPL objects and operators. SPIRAL then uses (3) and the code generation rules from Table I together with the optimizations described in [25, 27] to generate the final code, which is given in Listing 1. The function `mgcycle3()` requires 60 floating-point operations (additions and multiplications) and runs in 71 clock cycles on a single core of a 3.6 GHz Intel Xeon E3-1290. It loads 18 double precision numbers (144 bytes) of input data, as well as 5 unique double precision constants (40 bytes), and produces 18 double precision outputs (144 bytes). This results in an *arithmetic intensity* of 5.45 bytes/flop. This represents good efficiency relative to the resources used, as the code is scalar single-threaded C that does not leverage SIMD vector instructions and multicore parallelism. The code is in style very similar to the codelets used in FFTW [29].

6.2. Performance Results

Let us now present preliminary performance results. The multigrid solver problem sizes that we can currently produce with SPIRAL are limited to a few recursion steps ($n = 1, 3, 7, 15, 31$), since we do not support loop optimization and recursive code generation for multigrid solvers, yet. Supporting larger problem sizes is an ongoing research effort that requires internal extensions of the SPIRAL code generation and optimization engines to support the code patterns seen in multigrid solvers. Currently, SPIRAL generates unrolled code and inefficient loop nests and exhausts the instruction cache for small sizes.

We generated small multigrid kernels: a single Richardson step, coarse-grid correction, a multigrid cycle, and a complete multigrid solver for small problem sizes ($n = 1, 3, 7, 15, 31$). We used $r = 4$ Richardson steps in the multigrid cycle, $m = 10$ multigrid cycles for a complete solve, and $\omega = 0.2$ in the Richardson step. We measured the performance of the SPIRAL-generated kernels on a single core of a 3.6 GHz Intel Xeon E3-1290 (SandyBridge) using the Intel C++ Compiler 12.1.5 with option `-O3` and SIMDization as well as parallelization turned off. The results are summarized in Table III.

For small sizes, the performance is as expected for SPIRAL-generated L1 cache-resident code. The code performs well for problem sizes where the code fits into the L1 instruction cache. For larger sizes, the generated code becomes large, and the instruction cache is exhausted. At this point, the performance deteriorates. This is an artefact of the missing loop-level and recursion optimizations. As reference, a complete multigrid solver for the discretisation of (5) on a grid with $n = 7$ in 2D generated by SPIRAL is about 3,500 lines of code when fully unrolled.

As noted in Section 5, small unrolled highly optimized kernels are often used in high-performance libraries as the base of recursion, and usually contain the bulk of the floating-point operations [29]. SPIRAL-based code generation for multigrid solvers would be able to generate highly efficient multigrid codelets to be used in a larger solver despite the fact that code generation is limited to unrolled code due to the missing loop-level optimizations.

7. CONCLUSION AND OUTLOOK

We have presented a first example of generating a multigrid solver with the SPIRAL code generation and autotuning system. This shows that the SPIRAL code generator—while up to now having been used mainly for applications in signal processing and alike—can be used to generate multigrid methods for structured grids.

DOI: 10.1002/cpe

Listing 1: Code generated by SPIRAL for the SPL specification $\mathrm{MGCycle}_{3,0.125,1}$. The function `mgcycle3()` requires 60 floating-point operations (additions and multiplications) and runs in 71 clock cycles on a single core of a 3.6 GHz Intel Xeon E3-1290.

```
// Multigrid cycle
// input: double X[18], output: double Y[18]
// n = 3, \omega = 0.125, r = 1
//
void mgcycle3(double  *Y, double  *X) {
    double a462, a463, a464, a465, a466, a467, a468, a469,
        a470, a471, a472, a473, a474, a475, a476, a477,
        a478, a479, a480, s507, s508, s509, s510, s511, s514;
    a462 = (0.25**(X));
    a463 = (0.125**((X + 3)));
    a464 = (0.125**(X));
    a465 = (0.25**((X + 3)));
    a466 = (0.125**((X + 6)));
    a467 = (0.25**((X + 6)));
    a468 = (0.25**((X + 1)));
    a469 = (0.125**((X + 4)));
    a470 = (0.125**((X + 1)));
    a471 = (0.25**((X + 4)));
    a472 = (0.125**((X + 7)));
    a473 = (0.25**((X + 7)));
    a474 = (0.25**((X + 2)));
    a475 = (0.125**((X + 5)));
    a476 = (0.125**((X + 2)));
    a477 = (0.25**((X + 5)));
    a478 = (0.125**((X + 8)));
    a479 = (0.25**((X + 8)));
    s507 = (a468 + a469 + a464 + a468 + a476 + (0.125**((X + 10))));
    s508 = (a464 + a465 + a466 + a465 + a469 + (0.125**((X + 12))));
    s509 = (a476 + a477 + a478 + a469 + a477 + (0.125**((X + 14))));
    s510 = (a469 + a473 + a466 + a473 + a478 + (0.125**((X + 16))));
    a480 = (2.0*(a470 + a471 + a472 + a463 + a471 + a475 + (0.125**((X + 13)))));
    s511 = (0.25*((s507 - a480) + s510 + (s508 - a480) + s509));
    s514 = (0.5*s511);
    *(Y) = s514;
    *((Y + 3)) = s511;
    *((Y + 6)) = s514
    *((Y + 1)) = s511;
    *((Y + 4)) = (2.0*s511);
    *((Y + 7)) = s511;
    *((Y + 2)) = s514;
    *((Y + 5)) = s511;
    *((Y + 8)) = s514;
}
```

Table III. Performance Results of small multigrid kernels on a single core of a 3.6 GHz Intel Xeon E3-1290 (SandyBridge) using scalar code. All results are given in clock cycles.

Kernel	$n = 3$	$n = 7$	$n = 15$	$n = 31$
Single Richardson Step	67	369	1,757	27,167
Coarse-Grid Correction	32	255	1,630	68,012
Multigrid Cycle	190	1,623	19,045	–
Complete MG Solve	1,517	24,311	–	–

For this purpose, no extension of the SPIRAL generator and formal language was necessary because we were able to formulate multigrid as a composition of linear operators. To do so, the current approximation is stacked on top of the affine part of multigrid, which is the right-hand side at the current level, as described at the end of Section 4. The SPIRAL implementation is straightforward, but substantial work remains to be done to scale code generation from kernels

DOI: 10.1002/cpe

to loop code and a full autotuning library. Overall, we conclude that the approach is feasible and opens the SPIRAL code generation technique for a broader class of applications.

This work is a first contribution to allow multigrid methods to benefit from SPIRAL's domain-specific code generation and performance optimization capabilities. In addition to exploitation of vector instructions and multicore parallelism, SPIRAL provides powerful mechanisms for automatic selection of storage layout for spatial locality, together with coordinated blocked execution for temporal locality. Further, it applies automatic performance tuning to select good implementation parameters like block sizes. SPIRAL currently performs all its basic block level optimizations like common subexpression elimination, algebraic simplifications, array scalarization, and copy propagation for multigrid code. To scale the approach to larger sizes, three key optimizations need to be adapted to support multigrid code: code pattern specific loop merging, generation of recursive parameterizable implementations from recursive specifications, and parallelism support.

Currently, the parametrization of through the choice of restriction, smoother, and PDE are hard-coded into the rules. A challenge for future work on multigrid will be to separate these choices from the rule system and develop a parameterized meta rule system that allows for search over algorithmic choices. This can be achieved by a slight generalization of the current rules and by adding termination and breakdown rules that capture other smoothers and restrictions. Parameterizing the PDE type and boundary conditions is a harder problem, since a number of rules depend on the particular discretization of the PDE and its boundary conditions and, thus, more general right-hand sides of the rules for $\mathrm{Residual}_n$, $\mathrm{Interpolate}_n$, and $\mathrm{ResidueLaplace}_{n,\omega}$ will be required.

In the future, we will go on to handle more complex multigrid methods, e.g., for different problems and involving other algorithmic components. We are currently working on identifying possible extensions of SPIRAL that allow for a more elegant formulation of the methods as well as for generation of more efficient code.

ACKNOWLEDGEMENTS

Bolten and Lengauer gratefully received funding via project *ExaStencils* [32] in the DFG priority programme 1648 *SPPEXA* (grant numbers BO 3405/2 and LE 912/15). This has been cooperative work that started at the Dagstuhl Seminar *Advanced Stencil-Code Engineering* [33] in April 2015. Kelly was also supported by the EPSRC project *PRISM* (grant number EP/L000407/1). *SPPEXA* also partially supported the attendance of Franchetti and Kelly at this seminar. Mohr is an unfunded member of *SPPEXA* project *TerraNeo* [34]. Franchetti was supported by DARPA under agreement number FA8750-12-2-0291 and HR0011-13-2-0007. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

1. Fedorenko RP. The speed of convergence of one iterative process. *USSR Comp. Math. Math. Phys.* 1964; **4**(3):227–235.
2. Bakhvalov NS. On the convergence of a relaxation method with natural constraints on the elliptic operator. *USSR Comp. Math. Math. Phys.* 1966; **6**:101–135.
3. Brandt A. Multi-level adaptive technique (MLAT) for fast numerical solution to boundary value problems. *Proc. Third Int. Conf. on Numerical Methods in Fluid Mechanics*, *Lecture Notes in Physics*, vol. 18, Cabammes H, Temam R (eds.), Springer-Verlag: Berlin, Heidelberg, New York, 1973; 82–89.
4. Hackbusch W. Ein iteratives Verfahren zur schnellen Auflösung elliptischer Randwertprobleme. *Rep. 76-12*, Institute for Applied Mathematics, University of Cologne 1976.
5. Hackbusch W. On the multi-grid method applied to difference equations. *Computing* 1978; **20**:291–306.
6. McCormick SF, Ruge JW. Multigrid methods for variational problems. *SIAM J. Numer. Anal.* October 1982; **19**(5):924–929.
7. Braess D, Hackbusch W. A new convergence proof for the multigrid method including the V-cycle. *SIAM J. Numer. Anal.* October 1983; **20**(5):967–975.
8. Mandel J, McCormick SF, Ruge JW. An algebraic theory for multigrid methods for variational problems. *SIAM J. Numer. Anal.* February 1988; **25**(1):91–110.
9. Briggs WL, Henson VE, McCormick SF. *A Multigrid Tutorial*. 2nd edn., Society for Industrial and Applied Mathematics, 2000.
10. Yavneh I. Why Multigrid methods are so efficient. *Computing in Science and Engg.* 2006; **8**(6):12–22.
11. Trottenberg U, Oosterlee C, Schüller A. *Multigrid*. Academic Press, 2001.
12. Püschel M, Franchetti F, Voronenko Y. Spiral. *Encyclopedia of Parallel Computing*, Padua D, *et al.* (eds.). Springer, 2011; 1920–1933.

DOI: 10.1002/cpe

13. Püschel M, Moura JMF, Johnson J, Padua D, Veloso M, Singer B, Xiong J, Franchetti F, Gacic A, Voronenko Y, *et al.*. SPIRAL: Code generation for DSP transforms. *Proc, IEEE* 2005; **93**(2):232– 275. Special issue on Program Generation, Optimization, and Adaptation.
14. Fiorentino G, Serra S. Multigrid methods for Toeplitz matrices. *Calcolo* 1991; **28**:238–305.
15. Fiorentino G, Serra S. Multigrid methods for indefinite Toeplitz matrices. *Calcolo* 1996; **33**:223–236.
16. Serra-Capizzano S. Convergence analysis of two-grid methods for elliptic Toeplitz and PDEs matrix sequences. *Numer. Math.* 2002; **92**:433–465.
17. Franchetti F, de Mesmay F, McFarlin D, Püschel M. Operator language: A program generation framework for fast kernels. *Domain Specific Languages (DSL)*, *Lecture Notes in Computer Science*, vol. 5658, Taha WM (ed.), Springer, 2009; 385–410.
18. de Mesmay F, Chellappa S, Franchetti F, Püschel M. Computer generation of efficient software Viterbi decoders. *High Performance Embedded Architectures and Compilers (HiPEAC)*, *Lecture Notes in Computer Science*, vol. 5952, Patt YN, Foglia P, Duesterwald E, Faraboschi P, Martorell X (eds.), Springer, 2010; 353–368.
19. McFarlin D, Franchetti F, Moura JMF, Püschel M. High performance synthetic aperture radar image formation on commodity architectures. *Proc. SPIE* Apr 2009; **7337**:08.
20. de Mesmay F, Franchetti F, Voronenko Y, Püschel M. Automatic generation of adaptive libraries for matrix-multiplication. Proc. 5th Int. Workshop on Parallel Matrix Algorithms and Applications (PMAA) Jun 2008. Presentation #46 (abstract reviewed), `www.cfe-csda.org/pmaa08/pmaaBOA.pdf`.
21. Franchetti F, Püschel M, Voronenko Y, Chellappa S, Moura JMF. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine* 2009; **26**(6):90–102. Special issue on Signal Processing on Platforms with Multiple Cores.
22. Franchetti F, Püschel M. Short vector code generation for the discrete Fourier transform. *Proc. 17th Int. Parallel and Distributed Processing Symposium (IPDPS)*, IEEE Computer Society, 2003; 10 pp.
23. Chellappa S, Franchetti F, Püschel M. Computer generation of fast Fourier transforms for the Cell Broadband Engine. *Proc. Int. Conf. on Supercomputing (ICS)*, ACM, 2009; 26–35.
24. Milder PA, Franchetti F, Hoe JC, Püschel M. Hardware implementation of the discrete Fourier transform with non-power-of-two problem size. *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, IEEE, 2010; 1546–1549.
25. Xiong J, Johnson J, Johnson R, Padua D. SPL: A language and compiler for DSP algorithms. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2001; 298–308.
26. Van Loan C. *Computational Framework of the Fast Fourier Transform*. SIAM, 1992.
27. Franchetti F, Voronenko Y, Püschel M. Loop merging for signal transforms. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2005; 315–326.
28. Whaley RC, Dongarra J. Automatically tuned linear algebra software (ATLAS). *Proc. Supercomputing*, 1998; 33 pp. Article 38, `math-atlas.sourceforge.net`.
29. Frigo M, Johnson SG. The design and implementation of FFTW3. *Proc. IEEE* Feb 2005; **93**(2):216–231. Special issue on Program Generation, Optimization, and Platform Adaptation.
30. Voronenko Y, de Mesmay F, Püschel M. Computer generation of general size linear transform libraries. *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, ACM, 2009; 102–113.
31. Kong M, Veras R, Stock K, Franchetti F, Pouchet LN, Sadayappan P. When polyhedral transformations meet SIMD code generation. *Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, ACM, 2013; 127–138.
32. Lengauer C, Apel S, Bolten M, Größlinger A, Hannig F, Köstler H, Rüde U, Teich J, Grebhahn A, Kronawitter S, *et al.*. ExaStencils: Advanced stencil-code engineering. *Euro-Par 2014: Parallel Processing Workshops, Part II*, *Lecture Notes in Computer Science*, vol. 8806, Lopes L, *et al.* (eds.), Springer, 2014; 553–564.
33. Lengauer C, Bolten M, Falgout RD, Schenk O. Advanced stencil-code engineering (Dagstuhl Seminar 15161). *Dagstuhl Reports* Dec 2015; **5**(4):56–75.
34. Rüde U, Gmeiner B, Bartuschat D, Wohlmuth B, Waluga C, Huber M, John L, Bunge HP, Weismüller J, Baumann S, *et al.*. TerraNeo – Integrated co-design of an exascale earth mantle modeling framework. *inSiDE* Spring 2016; **14**(1):78–81.

DOI: 10.1002/cpe