ExaStencils – Advanced Multigrid Solver Generation

Christian Lengauer, Sven Apel, Matthias Bolten, Shigeru Chiba, Ulrich Rüde, Jürgen Teich, Armin Größlinger, Frank Hannig, Harald Köstler, Lisa Claus, Alexander Grebhahn, Stefan Groth, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, Christian Schmitt, and Jonas Schmitt

Christian Lengauer
University of Passau, e-mail: christian.lengauer@uni-passau.de

Sven Apel
University of Passau and Saarland University, e-mail: apel@cs.uni-saarland.de

Matthias Bolten
University of Wuppertal, e-mail: bolten@math.uni-wuppertal.de

Shigeru Chiba
The University of Tokyo, e-mail: chiba@is.titech.ac.jp

Ulrich Rüde
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: ulrich.ruede@fau.de

Jürgen Teich
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: juergen.teich@fau.de

Armin Größlinger
University of Passau, e-mail: armin.groesslinger@uni-passau.de

Frank Hannig
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: frank.hannig@fau.de

Harald Köstler
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: harald.koestler@fau.de

Lisa Claus
University of Wuppertal, e-mail: claus@math.uni-wuppertal.de

Alexander Grebhahn
University of Passau, e-mail: grebhahn@fim.uni-passau.de

Stefan Groth
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: stefan.groth@fau.de

Stefan Kronawitter
University of Passau, e-mail: kronast@fim.uni-passau.de

Sebastian Kuckuk
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: sebastian.kuckuk@fau.de

Hannah Rittich
Jülich Supercomputing Centre, Forschungszentrum Jülich, e-mail: h.rittich@fz-juelich.de

Christian Schmitt
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: christian.j.schmitt@fau.de

Jonas Schmitt
Friedrich-Alexander University Erlangen-Nürnberg, e-mail: jonas.schmitt@fau.de

Abstract Present-day stencil codes are implemented in general-purpose programming languages, such as Fortran, C, or Java, Python or derivates thereof, and harnesses for parallelism, such as OpenMP, OpenCL or MPI. Project ExaStencils pursued a domain-specific approach with a language, called ExaSlang, that is stratified into four layers of abstraction, the most abstract being the formulation in continuous mathematics and the most concrete a full, automatically generated implementation. At every layer, the corresponding language expresses not only computational directives but also domain knowledge of the problem and platform to be leveraged for optimization. We describe the approach, the software technology behind it and several case studies that demonstrate its feasibility and versatility: high-performance stencil codes can be engineered, ported and optimized more easily and effectively.

1 Overview of ExaStencils

1.1 Project Vision

ExaStencils[1] takes a revolutionary, rather than evolutionary, approach to software engineering for high performance. It seeks to provide a proof of concept that programming can be simplified considerably and that program optimization can be made much more effective by concentrating on a limited application domain. In the case of ExaStencils, it is a subdomain of geometric multigrid algorithms [85]. The idea is to write a program in a *domain-specific programming language* (DSL). In our case, it is an *external* DSL (i.e., a DSL built from scratch rather than embedded in an existing language), called ExaSlang [75]. ExaSlang is stratified into four layers of abstraction. Each layer provides a different view of the problem solution and can be enriched with information to allow for particular optimizations at that layer. ExaSlang's most abstract layer specifies the problem as a set of partial differential equations (PDEs) defined on a continuous domain. The most concrete layer allows the user to specify low-level details for an efficient implementation on the execution platform at hand. Ideally, the domain expert should only be dealing with the first layer (plus some menu-driven options). The ExaStencils code generator should be able to generate all lower code layers while applying a set of optimizations autonomously before producing efficient target code.

1.2 Project Results

This subsection summarizes the challenges that drove the development of ExaStencils and how far we got in the period of SPPEXA funding.

[1] `www.exastencils.org`

We begin with the delineation of the domain and the mathematical challenges, distinctly from the computer science challenges, that ExaStencils addressed (see Section 2). We restricted our attention to the development of smoothers for geometric multigrid methods and the required analysis tools. In this domain, local Fourier analysis (LFA) is the method of choice to analyze the developed smoothers and the entire multigrid method. To make LFA useful for our purpose, we extended the method to periodic stencils, covering block smoothers and varying coefficients [11].

The first major computer science challenge was to cover, with one single source program, a wider range of multigrid solvers than is possible with contemporary implementations based on general-purpose host languages such as Fortran or C++ (see Section 3). To this end, we decided not to build ExaSlang on an existing, general-purpose host language but to make it an external DSL. We were able to demonstrate its flexibility already early on in the project by providing a common ExaSlang program for Jacobi, Gauss-Seidel, and red-black solvers for finite differences and finite volumes with constant and linear interpolation and with restriction [53]. Later on, we introduced a way to specify data layout transformations simply by a linear expression—a feature that can aid the development process significantly [44] (see Subsections 3.1–3.2). A smaller task was to decide how to describe aspects of the execution platform in a platform-description language (TPDL) [73] (see Subsection 3.3).

The second major computer science challenge was to reach high performance with our approach to code generation on a wide range of architectures (see Section 4). One important aspect here is what information to provide at which layer (see Subsection 4.1). While the syntax of ExaSlang is partly inspired by Scala, Matlab and LaTeX, our target language is C++ with additional features that depend on the execution platform (see Subsection 4.2). We demonstrated that, with the help of our optimization techniques (see Subsections 4.3–4.5), weak scaling could be achieved on traditional cluster architectures such as the *JUQUEEN* supercomputer at the Jülich Supercomputing Center (JSC) [75, 49]. We also achieved weak scaling on the GPU cluster *Piz Daint* at the Swiss National Supercomputing Centre (CSCS). Furthermore, we demonstrated the automatic generation of solvers that can be executed on emerging architectures such as ARM [51] and FPGA platforms [77, 78].

One new concept that ExaStencils introduced into high-performance computing is that of *feature-based domain-specific optimization* [5] (see Subsection 4.6). The central idea is to view a source code, such as a stencil implementation or an application, as a member of a *program family* or *software product line* rather an as an isolated individual, and to describe the source code by its commonalities and variabilities with respect to the other family members in terms of features. A *feature* represents a concept of the domain (e.g., a type of smoother or grid) that may be selected and combined with others on demand. With this approach, a large search space of configuration choices can be reviewed automatically at the level of domain concepts and the most performant choices for the application and execution platform at hand can be identified. To this end, we devised a framework of sampling and machine learning approaches [36, 39, 81] that allow us to derive a *performance model* of a given code that is *parameterized in terms of its features*. This way, we can express performance behavior in terms of concepts of the domain and automatically determine optimal configurations that are tailored to the problem at hand, which we have demonstrated in the domain of stencil codes [30, 31, 32, 36, 39, 81, 59] and beyond (e.g., databases, video encoders, compilers, and compression tools). Our framework integrates well with the other parts of ExaStencils that use and gather domain and configuration knowledge in different phases.

Project ExaStencils came with several case studies whose breadth was to demonstrate the practicality and flexibility of the approach (see Section 5). The case studies are a central deliverable of ExaStencils. They include studies close to real-world problems: the simulation of non-Newtonian and non-isothermal fluids (see Subsection 5.3 and a molecular dynamics simulation (see Subsection 5.4).

We conducted two additional studies at the fringes of ExaStencils, exploring alternative approaches (see Section 6). In one, the option of an internal rather than external DSL was explored: ExaSlang 4 was embedded in the mutual host languages Java and Ruby to study the trade-off between the effort of the language implementation and the performance gain of the target code [17]. The outcome was that an embedding is possible but, as expected, with a loss of performance (see Subsection 6.1). In the second study, we implemented a simple multigrid solver in SPIRAL [10] (see Subsection 6.2). The success of the SPIRAL project [24, 61] a decade ago was a strong motivator for project ExaStencils. SPIRAL can handle simple algebraic multigrid solvers but would have to be extended for more complex ones.

1.3 Project Peripherals

Attempts of abstraction and automation in programming have received increased attention in the past two decades in the area of software engineering. High-performance computing has been comparatively conservative in going down this road. The reason is that the demands on performance are much higher than in general software engineering, and the architectures used to achieve it are more complex, notably with large numbers of loosely coupled processors.

The potential of an effective automation grows as the application domain shrinks. Promising domains are much smaller than those of any general-purpose programming language. The extreme is the compiler FFTW [25] that targets a single numerical problem, the fast Fourier transform. As just mentioned, SPIRAL widened the domain to linear transforms (and, lately, beyond [82]). By now, quite a number of optimizing code generators have been proposed that target stencil codes. Patus [18] has a strong focus on autotuning. The strong point of Pochoir [83] is the cache obliviousness of its target code. Pochoir addresses constant stencils and is based on the C dialect Intel Cilk with limited portability to other platforms, Devito [52] performs symbolic computation via SymPy and targets shared-memory architectures, just like Snowflake [90]. Firedrake [63] is primarily for finite-element methods and has adopted multigrid lately. One recent component of it is Coffee [54], which addresses the local assembly in finite-element methods. STELLA [34] and its spiritual successor GridTools generate code for stencil computations on distributed structured grids for climate and weather modeling.

For the domain of image processing, many approaches based on code generation exist that are conceptually similar to the idea behind project ExaStencils. Notable projects include Halide [62], HIPA^{cc} [56], PolyMage [58], and Mint [86]. The specification of image filter kernels is related to the concept of stencils, and many of the basic parallelization techniques are comparable. However, ExaSlang is fundamentally based on computational domains that are multidimensional, as opposed to the usually two-dimensional data structures used to represent images. Image processing DSLs usually target shared-memory parallel systems, i.e., single compute nodes such as multi-core CPUs or a single GPU. On the other hand, ExaStencils aims at the domain of HPC and, consequently, supports distributed-memory systems and respective parallelization techniques.

2 The Domain of Multigrid Stencil Codes

2.1 Multigrid

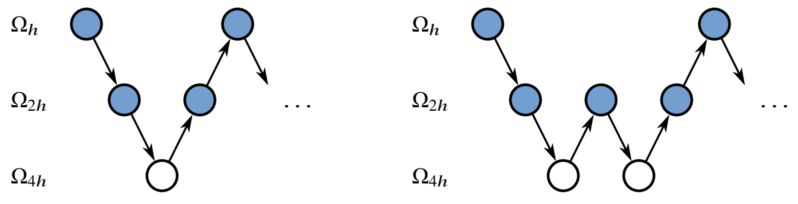
The goal of the ExaStencils project has been the automatic generation of efficient stencil codes for geometric multigrid solvers [85] on (semi-)structured grids. Multigrid methods form a class of iterative solvers for large and sparse linear systems. These methods have originally been developed in the context of the numerical solution of partial differential equations, which often involve the solution of such linear systems.

Multigrid methods combine two complementary processes: a *smoothing process* and a *coarse-grid correction*. While each of the two processes is by itself not sufficient to solve the problem efficiently, their combination yields a fast solver.

The smoothing process is usually a straightforward iterative method that converges rather slowly when used on its own. Combining such a process with a coarse-grid correction accelerates the convergence of the resulting method by augmenting the iteration with information obtained on a coarser grid. Since the grid determines the resolution at which the solution is being computed, a multigrid method considers the problem at different resolutions.

A multigrid method performs stencil computations on a hierarchy of fine to successively coarser grids. The overall cost of the method can be reduced further by applying this idea recursively, i.e., instead of two grids, we consider a hierarchy of successively coarser grids. The recursion follows a so-called *cycling strategy*, e.g., a V-cycle or a W-cycle (see Figure 1). The cycling strategy determines how much work is performed at what level of the grid hierarchy, which also has an effect on the convergence rate of the method. On the coarser grids, less processing power is required.

In summary, to construct a multigrid method, one must choose a set of components: a smoother, an interpolation, a restriction, a coarse-grid approximation, and



(a) V-cycle (b) W-cycle

Fig. 1: Cycling strategies across the grid hierarchy. The hierarchy levels are denoted by Ω_h, Ω_{2h}, and Ω_{4h}, progressing from the finest to the coarsest grid. Light circles solve the coarse-grid system directly, and dark circles solve the coarse-grid system recursively. Down arrows symbolize restrictions, up arrows interpolations.

a cycling strategy. The choice of components influences the number of iterations required to obtain an adequate solution, the computational cost per iteration, and the communication pattern of the method. Furthermore, the behavior of the method depends on the linear system to be solved.

2.2 Advancing the Mathematical Analysis of Multigrid Methods

At the start of project ExaStencils stood the search for a way to estimate the number of iterations that a multigrid method requires to produce an adequate solution. The choice of multigrid components determines the operations that must be performed per iteration. Hence, to estimate the time required per iteration, one just needs to place these operations in an appropriate order and estimate the duration time of their execution. While this is by no means a trivial endeavor, estimating the number of iterations needed is inherently different. In this section, we deal with the latter problem.

To estimate the number of iterations, we must consider the mathematical properties of a given multigrid method for a given set of components and problem. For this purpose, we must determine the contraction properties of the *iteration operator* of the multigrid method for a given configuration. To this end, *local Fourier analysis* (LFA) tells us whether the repeated application of the iteration operator lets the error converge to zero. In particular, we are interested in the resulting rate of convergence.

For project ExaStencils, we had to extend the capabilities of LFA. We wanted to analyze block smoothers, which have a higher arithmetic density than standard smoothers (see Subsection 2.3), and complex problems for which no feasible way of applying LFA was known. Furthermore, we aimed at an automation of the analysis.

At the beginning of project ExaStencils, we considered using and expanding an existing LFA software [88]. However, in the course of the project, it became quickly evident that we needed a more general approach, something beyond a collection of templates that allow to fill in some blank spots but do not facilitate the reuse or recombination of components. Since we intended to explore many possible combinations of various multigrid components, it was not feasible to program an individual LFA by hand for every combination. We developed the *LFA Lab* [65], which is based on the principle of combining a set of primitive expressions to complicated ones, enabling a much more flexible analysis.

A multigrid method, as well as its components, can be characterized by their iteration operators. An *iteration operator* describes the propagation of the approximation error during the execution of the method. If we denote the error after the k-th iteration with $e^{(k)}$ and the iteration matrix of the method with E, we have the following equation:

$$e^{(j+1)} = Ee^{(j)}$$

The spectral radius $\rho(E)$ and the operator norm $\|E\|$ of E characterize the asymptotic and worst-case error reduction factors of the method, respectively. LFA determines these quantities in a simplified setting.

Let h, with $0 < h \in \mathbb{R}$, be the step size of the infinite d-dimensional grid $G_h := h \cdot \mathbb{Z}^d$. We consider stencil operators on the space of bounded grid functions $\ell_2(G_h) := \{u : G_h \to \mathbb{C} \mid \sum_{x \in G_h} |u|^2 < \infty\}$. A *stencil operator* $A : \ell_2(G_h) \to \ell_2(G_h)$ is a linear operator given by a family $\{s_k\}_{k \in \mathbb{Z}^d}$, $s_k \in \mathbb{R}$, such that

$$Au\,(x) := \sum_{k \in \mathbb{Z}^d} s_k u(x + hk) \quad \text{for} \quad x \in G_h \text{ and } u \in \ell_2(G_h)\,.$$

Stencil operators have a particularly simple form when transformed via the discrete-time Fourier transform.

The *discrete-time Fourier transform* (*DTFT*) $\mathcal{F}$ is a linear isometry that maps the space $\ell_2(G_h)$ onto $L_2(\Theta_h) := \{u : \Theta_h \to \mathbb{C} \mid \int_{\Theta_h} |u(x)|^2 \,\mathrm{d}x < \infty\}$, where $\Theta_h := [0, \frac{2\pi}{h})^d$. In other words, it represents a function on an infinite grid by a function on a continuous and bounded interval. A stencil operator A in Fourier space, i.e., the operator $\mathcal{F}A\mathcal{F}^{-1}$, is just the multiplication by a function $\hat{a} : L_2(\Theta_h) \to \mathbb{C}$. We call the function $\hat{a}$ the *symbol of* A. Thus, the symbol of a stencil operator is a function that encodes the entire behavior of the infinite-dimensional operator.

The symbol $\hat{a}$ of a stencil operator reveals the desired information about the operator A. We have that

$$\rho(A) = \text{ess-sup}_{\theta \in \Theta_h} |\hat{a}(\theta)| \quad \text{and} \quad \|A\| = \text{ess-sup}_{\theta \in \Theta_h} |\hat{a}(\theta)|\,,$$

where ess-sup denotes the essential supremum. These formulas mean that, to compute the spectral radius or the operator norm of a stencil operator, we must just compute the largest value of the absolute modulus of its symbol $\hat{a}$. Note that, in the definition of stencil operators, we have assumed that stencil s does not depend on position x. However, it can be useful to consider operators whose stencil is allowed to change with the position.

A stencil that depends arbitrarily on the position has no particularly simple form in the Fourier domain. However, we were able to show that periodic stencils do have a simple representation [11, 66]. A *periodic stencil* depends on the position, but has the same entries repeated periodically across the entire domain. While this does not represent stencils accurately that are variable in the entire domain, at least some variability is reflected in the analysis. We showed that a periodic stencil operator is described, after a proper reordering of the frequency domain, by a matrix of ordinary symbols—more precisely, by matrix symbols from the space $L_2^{n\times m}(\theta_{h'})$ for some appropriate positive h'. Furthermore, we were able to show that there is a one-to-one relationship between matrix symbols and periodic stencils.

Using matrix symbols, similar results for the spectral radius and operator norm hold. For an operator given by a periodic stencil, we have that

$$\rho(A) = \text{ess-sup}_{\theta\in\Theta_{h'}} \|\hat{a}(\theta)\| \quad \text{and} \quad \|A\| = \text{ess-sup}_{\theta\in\Theta_{h'}} \rho(\hat{a}(\theta)) .$$

Thus, to obtain the norm and spectral radius of the operator, we must find the largest value of the norm and spectral radius of the matrix $\hat{a}(\theta)$.

The framework of periodic stencils and matrix symbols allows for more advanced problems to be analyzed. It also lends itself to automation via software. Operators that have a matrix symbol can be combined in many ways such that the combination also has a matrix symbol. Thus, we can create a flexible LFA software by using the idea of providing first a set of primitive expressions and then means of combination and abstraction [1].

For example, the iteration operator of the weighted Jacobi method is

$$E_\mathrm{J} = I - D^{-1}A ,$$

where A is the system matrix, D the diagonal part of A, I the identity matrix and $\omega \in \mathbb{R}$ a weighting factor. If we assume that the behavior of A can be modelled sufficiently accurately by a (periodic) stencil operator on an infinite grid, we can replace each matrix by the infinite-dimensional operator given by the corresponding (periodic) stencils to simplify the analysis. Listing 1 shows the computation of the spectral radius of the iteration operator of the Jacobi method for the stencil resulting from the discretization of the Poisson equation using our software LFA Lab [65].

LFA Lab can be used as a simple Python [67] library, but it is more or less an embedded DSL for LFA. The user provides the (periodic) stencils, which yield the corresponding operators. Then, the user combines these operators with interpolation and restriction to an expression describing the desired iteration operator.

```python
from lfa_lab import *
g = Grid(2, [1.0/32, 1.0/32])
A = gallery.poisson_2d(g)
I = operator.identity(g)
omega = 0.8
E = I - omega * A.diag().inverse() * A
print((E.symbol().spectral_radius()))
```

Listing 1: Implementation of an LFA of the weighted ($\omega = 0.8$) Jacobi smoother for the solution of the Poisson equation using LFA Lab.

The code in Listing 1 is essentially a direct implementation of the formula for the iteration operator. However, keep in mind that this formula describes actually a combination of infinite dimensional operators. When calling the `symbol` method, the software determines automatically a way to represent the given iteration operator via its Fourier matrix symbol, which is a non-trivial procedure.

To compute the matrix symbol of an expression given by the user, appropriate sampling parameters must be determined. For this purpose, LFA Lab has two stages. The first stage extracts and analyzes the expression tree of the formula that the user entered. The second stage then samples the matrix symbol to obtain the spectral radius and operator norm. The power of the software lies in the fact that arbitrarily complex expressions can be analyzed.

A more complex example is the analysis of the two-grid method. It has the iteration operator

$$E_{\mathrm{TG}} = S\,(I - PA_{\mathrm{c}}^{-1}RA)\,S\,,$$

where A_{c} is the coarse grid operator, P the interpolation, R the restriction, and S the iteration operator of the smoother. Assume that we already have an analysis for the smoother. If we can express P, R, and A_{c} using Fourier matrix symbols, we can combine these with the analysis of the smoother we already have to analyze the two-grid method in its entirety.

In summary, we have constructed a powerful and flexible LFA software. The flexibility comes from a small set of primitive expressions and means of combination and abstraction. The periodic stencil operators are the primitive expressions, mathematical operations are a means of combination, and the Python programming language provides the opportunity of abstraction. The software is used to estimate the convergence rate of a multigrid method for a given set of components and a given problem. The estimate comes as a number of iterations a multigrid method needs to achieve adequate accuracy.

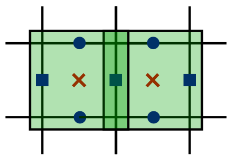
2.3 Advancing Multigrid Components

The choice of the smoothing component in a multigrid method is not always straightforward. Some problems require advanced smoothers. This can be easily appreciated when considering that, since the system contains a zero block, a pointwise relaxation is not possible [85]. The steady-state Stokes equations can be written as follows:

$$\begin{aligned} -\Delta \mathbf{u} + \nabla p &= \mathbf{f}, \quad \text{in} \quad \Omega \\ \nabla \cdot \mathbf{u} &= 0, \quad \text{in} \quad \Omega \end{aligned}$$

for a given domain Ω with boundary $\partial\Omega$. Here, $\mathbf{u}$ is the vector-valued fluid velocity, p is the pressure, and $\mathbf{f}$ describes an external force.

This linear system of PDEs can be discretized on staggered grids or by using appropriate finite elements. Here, we consider staggered grids in two dimensions. For the Stokes equations, the efficient Vanka smoother has a relatively high computational


(a) Vanka smoother

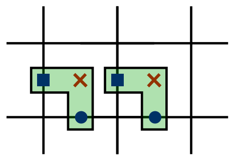
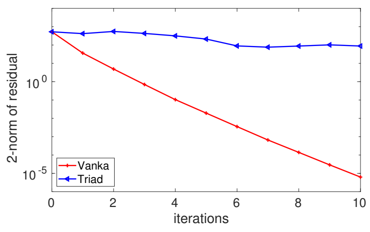
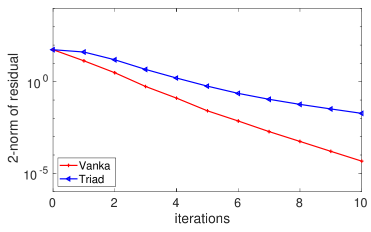

(b) Triad smoother

Fig. 2: Unknowns that are included in the blocks of the smoothing steps.



(a) Dirichlet boundary conditions



(b) Periodic boundary conditions

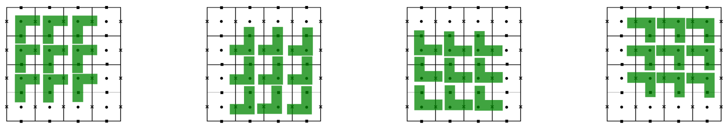
Fig. 3: Convergence behaviors of the two block smoothers applied to the Stokes equations discretized using staggered grids in the unit square with periodic and Dirichlet boundary conditions.

cost and unsatisfactory parallelization properties due to a process of overlapping block-smoothing steps (see Figure 2). This made us consider the Triad smoother as an alternative: it provides low computational cost in combination with good parallelization properties [19].

Both block smoothers are based on a collective updating process of unknowns inside one block [87]. As depicted in Figure 2, the Vanka blocks consist of five unknowns including one pressure and two velocity components in each direction while the Triad smoother comprises the simultaneous update of three unknowns including one unknown of each kind.

Numerical results for periodic boundary conditions in combination with parallelization properties and computational work show the potential of the Triad relaxation method. However, numerical results for the Stokes system with Dirichlet boundary conditions show that the Triad method in its original form has one issue: the convergence rate deteriorates tremendously. To illustrate this, we applied the method to the Stokes equation discretized on a staggered grid in the unit square, with periodic and with Dirichlet boundary conditions. As right-hand side, we chose in both cases $f_{u_x}(x, y) = 2\pi^2 \sin(\pi x)\sin(\pi y) + \pi\cos(\pi x)$, $f_{u_y}(x, y) = 2\pi^2 \cos(\pi x)\cos(\pi y) - \pi\sin(\pi y)$, and $f_p \equiv 0$; the initial guess was zero. Figure 3 shows the different convergence behaviors.

The convergence properties of the Triad smoother can be improved with the following idea: repeat the relaxation process four times while changing the unknowns contained in one block after each iteration, i.e., rotating the "L"-shaped pattern that



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 4: Order of iterations of the advanced block smoother.

describes the block to be relaxed. This algorithm, illustrated in Figure 4, improves the convergence significantly. When applying the four iterations of the proposed smoother, it is very similar to one iteration of Vanka. Thus, a smoother that has not been considered as an option before becomes a viable alternative to established smoothers for systems of PDEs. In addition, the order in which the boxes are updated can be varied. For more details and further results, see the dissertation of Lisa Claus [19].

3 Stencil-Specific Programming in ExaStencils

The central element in the ExaStencils approach is its stencil-specific programming language ExaSlang and its code generator. Then there is also a language for specifying properties of the execution platform. The two languages are discussed in this section. The code generator is the subject of the following section.

3.1 The Domain-Specific Language ExaSlang

The idea of ExaStencils is to support the domain-specific programming and optimization of stencil codes by providing different layers of abstraction, specifying the various aspects of the stencil code at the respectively suitable layer, and exploiting domain information available at that layer for an optimization of the specification. ExaSlang comes in four layers: from ExaSlang 1, the most abstract, to ExaSlang 4, the most concrete (see Figure 5).

- *Layer 1: the continuous problem*
  This is the layer for the scientist or engineer who needs the solution of the PDE. The problem is specified as a continuous equation. The present implementation supports Unicode and LaTeX symbols. Optional inputs are the specification of discretization and solver options used to autogenerate lower layers. There is also support for an automatic finite-difference discretization of operators.
- *Layer 2: the discrete problem*
  This is the most abstract layer that provides an executable description. Discretized functions are fields (data type, grid location), tied to a computational

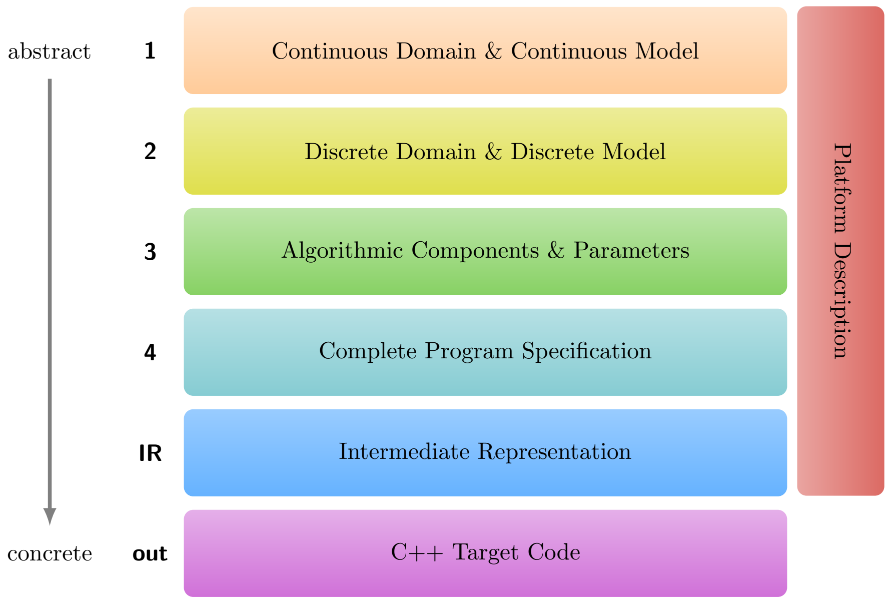

abstract
concrete
1
2
3
4
IR
out
Continuous Domain & Continuous Model
Discrete Domain & Discrete Model
Algorithmic Components & Parameters
Complete Program Specification
Intermediate Representation
C++ Target Code
Platform Description

Fig. 5: ExaSlang layers of abstraction

domain. Geometric information is provided in the form of *virtual fields* resolved to constants or field accesses. Discretized operators are provided as stencils or stencil templates.

- *Layer 3: the solver*
  At this layer, multigrid appears in the form of the specification of a solver for the discrete problem, either provided by hand or set up automatically. The implementation supports a Matlab-like syntax.
- *Layer 4: the application*
  This layer of ExaSlang can describe a full application, including communication, input/output, evaluation and visualization, but is still more abstract than C++ or Java in that it contains language constructs specific to multigrid. Optimizations at this layer include the tuning of communication patterns and data structure layouts.

Yet more concrete, but not accessible to the user, is an intermediate representation (IR) in which most code refinements take place and which forms the base for the generating of the target C++ code.

One may write one's program at the ExaSlang layer of one's choice and let the code generator refine it to a more concrete layer. One may also modify generated code to implement certain aspects that cannot be expressed at the chosen programming layer. However, the IR representation is not meant to be modified by the user.

Through its four layers, ExaSlang evolves from a declarative language at layer 1 to an imperative programming language at layer 4. Besides the standard data types that represent floating-point and integer numbers or strings, domain-specific data types represent vectors and matrices to be used for coupled systems of equations. Assembly of global vectors and matrices is not supported since we focus on local computations using stencils. In contrast, stencils are declared globally in ExaSlang programs. Furthermore, fields—corresponding to vectors that may represent the PDE's right-hand side or an approximation of the unknown function—are declared globally as well. Via declaration, certain settings important to parallelization may be specified by the user. One example is the size of ghost layers (also called halo layers), as depicted in Listing 5 in Section 3.2.

3.2 An ExaSlang Example

To illustrate how our language stack can be used to implement different aspects of partial differential equations (PDEs) solvers, let us take the Poisson equation

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega, \\ u &= g \quad \text{on} \quad \partial\Omega \end{aligned}$$

for a given domain Ω—here the unit square—Dirichlet boundary conditions $g = cos(\pi x) - sin(2\pi y)$ and the right-hand side $f = \pi^2 cos(\pi x) - 4\pi^2 sin(2\pi y)$. Listing 2 shows an exemplary layer 1 code for this specification. From it, our generator is able to derive representations at subsequent layers. The refinement methods employed in this process are described in Subsection 4.1 further below. Listings 3 and 4 illustrate variants similar to the auto-generated ones expressing the discretized version of the given equation (Listing 3) and the multigrid algorithm used to solve for it (Listing 4). Based on them, a complete layer 4 program can be assembled, comparable to the one illustrated in Listing 2. In the example code at the lower layers, parts of the source code has been omitted for the sake of compactness. A complete specification and examples of other PDEs are part of Sebastian Kuckuk's dissertation [47].

```
Ω = ( 0, 1 ) × ( 0, 1 )

u ∈   Ω = 0.0
u ∈ ∂ Ω = cos ( π x ) - sin ( 2 π y )

f ∈   Ω = π^2 cos ( π x ) - 4 π^2 sin ( 2 π y )

op = - Δ

uEq: op * u == f

/* discretization and solver hints (see Subsection 4.1) */
```

Listing 2: ExaSlang 1 code for the complete specification of the 2D Poisson problem.

```
global from [ 0, 0 ] to [ 1, 1 ]

Solution with Real on Node of global = 0.0
Solution@finest on boundary =
  cos ( PI * x ) - sin ( 2.0 * PI * y )
Solution@(all but finest) on boundary = 0.0

RHS with Real on Node of global =
  PI**2 * cos ( PI * vf_nodePos_x ) -
  4.0 * PI**2 * sin ( 2.0 * PI * vf_nodePos_y )

Laplace from Stencil {
  [ 0,  0] =>  2.0 / ( vf_gridWidth_x**2 ) +
               2.0 / ( vf_gridWidth_y**2 )
  [-1,  0] => -1.0 / ( vf_gridWidth_x**2 )
  [ 1,  0] => -1.0 / ( vf_gridWidth_x**2 )
  [ 0, -1] => -1.0 / ( vf_gridWidth_y**2 )
  [ 0,  1] => -1.0 / ( vf_gridWidth_y**2 )
}

SolEq {
  Laplace * Solution == RHS
}
```

Listing 3: ExaSlang 2 code for a complete specification of the 2D Poisson problem.

```
Field Residual from Solution
override bc for Residual with 0.0

Operator Restriction  from default restriction
  on Node with 'linear'
Operator Prolongation from default prolongation
  on Node with 'linear'

Function Smoother@all {
  repeat 3 times {
    Solution += diag_inv ( Laplace ) * ( RHS -
      Laplace * Solution ) where (i0 + i1) % 2 == 0
    Solution += diag_inv ( Laplace ) * ( RHS -
      Laplace * Solution ) where (i0 + i1) % 2 == 1
  }
}

Function VCycle@coarsest {
  /* implementation of a coarse-grid solver */
}

Function VCycle@(coarsest + 1 to finest) {
  Smoother ( )

  Residual = RHS - Laplace * Solution
  RHS@coarser = Restriction * Residual

  Solution@coarser = 0.0
  VCycle@coarser ( )

  Solution += Prolongation@coarser * Solution@coarser

  Smoother ( )
}
```

Listing 4: ExaSlang 3 implementation of a V(3,3)-cycle using an RBGS smoother.

```
Layout DefLayout<Real, Node >@all {
  duplicateLayers = [1, 1] with communication
  ghostLayers     = [1, 1] with communication
}

Field Solution< global, DefLayout, /* bc's */ >@finest
Field Solution< global, DefLayout, 0.0 >@(all but finest)
Field RHS     < global, DefLayout, None >
Field Residual< global, DefLayout, 0.0 >

/* operators as on layers 2 and 3 */

Function Smoother@all {
  color with (i0 + i1) % 2 {
    loop over Solution {
      Solution += omega * diag_inv ( Laplace ) *
                  ( RHS - Laplace * Solution )
    }
    communicate Solution
  }
}

/* VCycle functions */

Function Application {
  /* initialization */

  repeat 10 times {
    VCycle@finest ( )
  }

  /* de-initialization */
}
```

Listing 5: ExaSlang 4 code of a full application with a fixed number of V-cycles to solve for Poisson's equation discretized with finite differences.

```
<gpu name="Tesla_V100" role="worker">
  <param name="compute_capability" value="7.0" />
  <param name="api" value="cuda" />
    <memory size="16" unit="GigaByte" Type="HBM2">
      <param name="bandwidth" value="900" unit="GigaBps"/>
    </memory>
  <core quantity="5120" frequency="1246" frequency_unit="MegaHz" />
</gpu>
```

Listing 6: ExaSlang description of an accelerator card.

3.3 The Target-Platform Description Language

To be able to optimize a code adequately, one must know details of the execution platform. Our code generation process is governed by more than one hundred parameters that allow to select specific code refinements or to set device-specific properties. Examples include the use of vector units on CPUs and the corresponding instruction set to use, e.g., SSE or AVX on x86 CPUs, NEON on ARM-based CPUs or even QPX for IBM's POWER architecture. This yields a design space that is too large for users to be able to specify a (near-)optimal configuration of code generation settings. However, we may be able to derive sensible parameters from a structured description of the target platform. To this end, one element of ExaSlang is the so-called target platform description language (TPDL) [73]. One design goal was to increase the modularity and reusability of hardware component descriptions, such as CPUs or accelerators, to let users compose systems based on a repository of ready-made parametric snippets. By treating these in a fashion similar to the class concept in object-oriented programming languages, users can infer instantiations with parameters set appropriately. A short example describing an accelerator card is provided in Listing 6. It enumerates a number of technical hardware details, but also contains the important software information on the compute capability, i.e., which features of the target technology CUDA may be used.

A small, yet flexible library supports information retrieval from a TPDL specification, This enables DSL developers to check for certain information and aggregate or evaluate characteristics of the target platform. Developers need not to worry about the instantiations and their parameters but may just use the discrete specification, since all the processing required has already been done. For many recurring tasks, predefined queries are available. Listing 7 shows a predefined query to return all working units in a system. Its result is filtered to retain only working units that consume more than 50W of electrical power.

```
val workers = predefinedQuery(tpdlTree, GetWorkingUnits)
val workers50 = workers.filter(w => w.power > Watt(50))
workers50.foreach(System.out.println(_))
```

Listing 7: Query to enumerate working units consuming more than 50W of power.

4 The ExaSlang Code Generator

4.1 Refinement of ExaSlang Programs

The overall goal of ExaStencils has been to enable users to choose the layer most appropriate for them and code exclusively at this layer, e.g., by providing a continuous formulation of the problem to be solved at layer 1 and nothing more. Ideally, our framework would then automatically derive suitable discretizations (layer 2), solver components (layer 3) and parallelism particulars (layer 4). However, in practice, this requires *domain knowledge* whose automatic inference is beyond the capabilities of present software technology. We address this issue by introducing *hint* specifications which allow us to progress automatically to subsequent layers of ExaSlang. For example, at layer 1, *discretization hints* may be supplied. As Listing 8 shows, continuous functions are discretized at certain points of a computational grid, such as node positions. We also support the specification of operator discretization using finite differences. An optional renaming is also possible at this stage.

```
DiscretizationHints {
  u  => Solution                                on Node
  op => Laplace  with "FiniteDifferences" on Ω order 2
  uEq
}
```

Listing 8: A discretization hint block for a scalar equation in ExaSlang 1.

Then, we combine the equation provided at layer 1 and the discretization hints to synthesize a discretized form of the equation at layer 2. While layer 2 expresses the problem to be solved, its solution is specified at layer 3. This can be achieved either by implementing a suitable iterative solver by hand or by issuing a directive for our *generate solver* interface. Listing 9 illustrates such a directive.

```
generate solver for Solution in SolEq with {
  solver_smoother_numPre        = 3
  solver_smoother_numPost       = 3
  solver_smoother_coloring      = "red-black"
  solver_cgs                    = "ConjugateGradient"
}
```

Listing 9: A generate solver statement in ExaSlang 3.

For cases in which no further modification is required, matching *solver hints* may be provided at the upper layers to set up the code in Listing 9 automatically, allowing users to work exclusively at one layer. The generated solver is by default a geometric multigrid variant. In the concrete case of Listing 9, our framework would generate a standard V-cycle using three pre- and post-smoothing steps of a red-black Gauss-Seidel (RBGS) and a conjugate gradient (CG) coarse-grid solver. Frequently, minor adaptations of the generated solver are necessary. They can either be implemented by taking the implementation generated at layer 3, adapting it and replacing the original generate solver directive with the result. A more generic approach is to add *modifiers* to the generate solver directive. They target usually a certain *stage* of the multigrid solver, e.g., the restriction or the correction, at one or more levels of the hierarchy. These stages can either be replaced completely with custom layer 3 code, or arbitrary layer 3 statements may be added to be executed before or after the stage. A more complex option exists for *smoother stages*, as illustrated in Listing 10 for the case of a block smoother to be used when solving for the Stokes equations on a staggered grid (see Section 5).

```
smootherStage {
  loopBase p solveFor {
    u@[0, 0] u@[1, 0]
    v@[0, 0] v@[0, 1]
    p
  }
}
```

Listing 10: ExaSlang 3 code for a smoother stage for the 2D Stokes problem.

After the algorithmic specification at layer 3 is completed, it must be transformed to a program at layer 4. This requires, most importantly, the addition of data layout information for fields, loops to compose kernels, and suitable communication statements. Listings 11 and 12 illustrate the first half of an RBGS smoother.

```
Solution += ( diag_inv ( Laplace )
              * ( RHS - Laplace * Solution )
            ) where (i0 + i1) % 2 == 0
```

Listing 11: ExaSlang 3 code for the first half of an RBGS smoother.

```
communicate Solution
loop over Solution where ( i0 + i1 ) % 2 == 0 {
  Solution += diag_inv ( Laplace ) * ( RHS - Laplace * Solution )
}
```

Listing 12: ExaSlang 4 code for the first half of an RBGS smoother derived from its ExaSlang 3 counterpart in Listing 11.

This description of features is by no means complete. More detail is available in Sebastian Kuckuk's dissertation [47].

We designed the language stack of ExaSlang to enable maximum flexibility for users. They can either work at one level exclusively and make use of the hint system to generate more concrete specifications automatically, or implement different parts of their application at multiple levels, or mix both approaches.

```
var s = new DefaultStrategy("simple strategy")
s += new Transformation("resolve constants", {
  case Identifier("PI") => RealConstant(3.1415)
  case Identifier("E")  => RealConstant(2.7183)
})
s += new Transformation("constant folding", {
  case Addition(a : RealConstant, b : RealConstant)
    => RealConstant(a + b)
})
s.apply
```

Listing 13: A strategy with two simple transformations.

4.2 Generation of Target Code

After the domain-specific program has been refined to a complete program specification (ExaSlang 4), the IR code is subjected to many non-algorithmic transformations (recall Figure 5 on page 12). Among others, parallelization techniques are applied to the code (see Subsection 4.4), memory layouts are selected and applied to variables (see Subsection 4.3), required code fragments are inserted, and finally the program is written to disk as C++ files.

To express all these steps in a short syntax, we developed the code-transformation framework *Athariac* [74]. It allows to modify the tree-based representation of a program—called the *abstract syntax tree* (AST)—by using simple, yet powerful rewrite rules. Essentially, we specify a pattern in the AST, such as the node that represents a certain ExaSlang statement, and then define a structure with which to replace it. Of course, we can also remove nodes by specifying an empty replacement. A single rewrite rule is called a *transformation*, and a group of transformations is called a *strategy*. In Listing 13, a simple strategy for code-generation time evaluation of mathematical expressions is presented. First, the two identifiers *E* and *PI* are replaced by their constant numerical values. Then, we look for constant numerical values to be summed, e.g., 2.1 + 3.1415, and replace them with the actual result of the addition. Note that the constant-folding transformation must be applied multiple times for expressions involving multiple additions.

By chaining and conditional execution of strategies in a fixed order, ExaSlang programs can be refined for specific program configurations. Depending on the selected code-generation parameters and input-program size, between 200 and 300 transformations are required to generate C++ code. In Figure 6, the trajectory of a program from the intermediate representation to C++ output is depicted to illustrate the basic approach that applies to every program. Most of the optimizations are applied in this trajectory. Naturally, a concrete code-generation path may differ, as users can skip certain optimizations, or transformations may not be applied because the preconditions required are not satisfied. For example, it does not make sense to generate code for heterogeneous target devices if none exist.

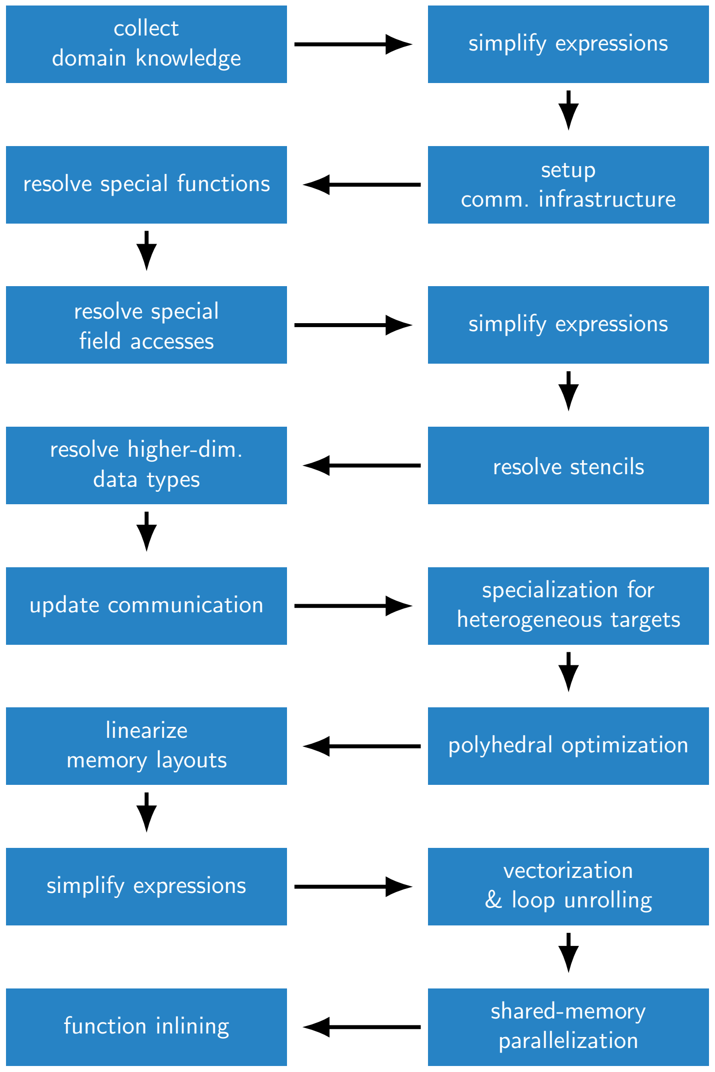

collect domain knowledge
simplify expressions
setup comm. infrastructure
resolve special functions
resolve special field accesses
simplify expressions
resolve stencils
resolve higher-dim. data types
update communication
specialization for heterogeneous targets
polyhedral optimization
linearize memory layouts
simplify expressions
vectorization & loop unrolling
shared-memory parallelization
function inlining

Fig. 6: The transformation trajectory of an ExaSlang program from its IR to C++.

In Figure 7, program sizes during the code-refinement process of the three-dimensional optical flow application (see Subsection 5.3) are depicted for a selection of different parallelization techniques. For all variants, we see different stages of the process. First, information on the program is gathered and preparations are done, i.e., setting array sizes and memory layouts. Next, minor code refinements are applied, such as the replacement of stencil convolutions with their corresponding computational rules. Then, depending on the parallelization technique, memory

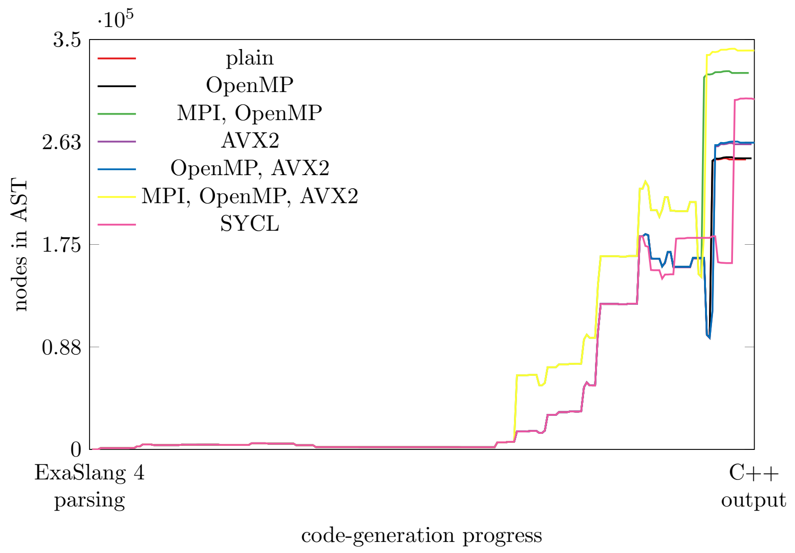


Fig. 7: AST size (number of nodes) during code generation for different variants of the 3D optical flow application, starting at ExaSlang 4.

layouts are imposed and loops are generated, respectively modified. The execution times of Athariac's code generator are usually on the same order of magnitude as an invocation of the target C++ compiler, i.e., within a few seconds to minutes. More detail on Athariac can be found in a recent issue of the Proceedings of the IEEE [74] and in Christian Schmitt's dissertation [72].

4.3 Target-Specific Optimization

Stencil codes have a very simple structure but they are difficult to optimize since the memory bandwidth usually acts as a performance brake. In project ExaStencils, a diverse set of optimizations has been developed to overcome this. We describe a small selection briefly here. More detail on all implemented techniques, including those not presented here, can be found in Stefan Kronawitter's dissertation [43].

Data Layout Transformations

Much effort of adapting stencil codes from one application or execution platform to another goes into making the data layout fit for best efficiency. One major limiting factor is usually the available memory bandwidth. Examples of such adaptations are color splitting for multi-color kernels or switching between an array of structs (AoS) and a struct of arrays (SoA).

ExaSlang admits flexible layout transformation directives for the specification of arbitrary affine transformations that are then applied automatically by the code generator [44]. Even though different variants of a color splitting or an SoA-to-AoS transformation may be most commonly used, the ExaStencils code generator is by no means restricted to them. Moreover, the only modification required to adapt the memory layout of any field is the insertion of an affine function that specifies the transformation. No other part of the application, including data initialization and communication, must be modified to make the new layout generally available. Explicit albeit very simple modifications to other parts are only necessary if they are meant to use different data arrangements. In this case, additional fields must be inserted for each layout and copy kernels are necessary.

This approach avoids unnecessary changes in the source code and constitutes a big advance in the ease of testing and evaluating different memory layout schemes in order to identify the best memory layout. There are other systems that offer similar transformation devices but not in this generality.

Polyhedral Code Exploration

Besides a layout modification, an affine transformation can lead to the most efficient implementation of a frequent structure in stencil codes: the loop nest. A popular approach to selecting such transformations automatically is the polyhedron model for loop optimization [22]. As for the data layout, the search is also here for best data locality. However, established automatic transformation techniques based on the PLuTo algorithm [13, 6, 12] fail to yield optimal results.

Our first attempt to select better transformations was a specialized variant of the PLuTo algorithm available in the ExaStencils code generator. While it is capable of detecting very good schedules for some stencil codes, it can also encounter problematic ones. For example, it fails completely for RBGS kernels. Thus, we developed a new, optimized, multi-dimensional polyhedral search space exploration for the ExaStencils code generator [46] that obtains in several cases better results than existing approaches, such as different PLuTo variants or PolyMage [58]. It also has the capability of specializing the search for the domain of stencil codes, which reduces the exploration effort dramatically without significantly impairing performance. An extreme but still beneficial approach is to choose the first schedule selected by our specialized search without any further evaluation. This may not lead to the best performance but it avoids the overhead of a complete exploration—and the performance improvement is still satisfactory: in most experiments it was only a few percentage points below the best variant explored.

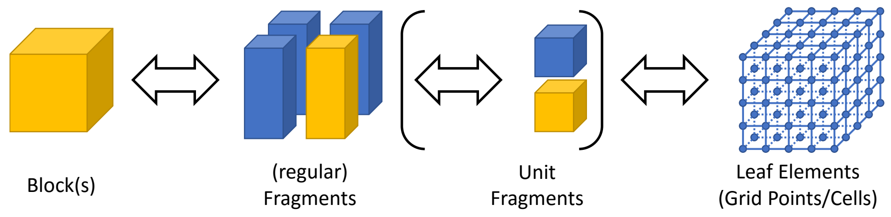


Fig. 8: Partitioning of the computational domain in ExaStencils [49].

Vectorization

A third optimization focuses on the vector units available in most processor architectures that provide single-instruction-multiple-data (SIMD) parallelism. Their use is typically mandatory for highest performance. However, each architecture comes with its own vector instruction set. Intel x86 features Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX) in several different versions, IBM's BlueGene/Q provides Quad Processing eXtension (QPX), and some ARM processors implement the Neon instruction set. Even though all of these sets target the same problem, their implementations differ not only in detail but also in key aspects, such as the way in which data can be fetched from main memory.

As a remedy, contemporary compilers are equipped with rudimentary automatic vectorization capabilities most of which are, unfortunately, not very effective. On top, the more advanced compilers can exclude some popular architectures and be costly or not widely available. Since the ExaStencils code generator comes with its own vectorization phase [43, 74], it avoids any dependence on a special target compiler. Currently, it supports Intel SSE3, AVX, and AVX2, as well as IBM's QPX and ARM Neon.

4.4 Parallelization

To parallelize ExaSlang applications automatically, mainly two concepts must be implemented [49]. First, data must be partitioned and distributed across the available compute resources and, second, data between the partitions must be synchronized periodically. We realize the former by splitting our computational domain into *blocks* which are further subdivided into *fragments*. Each fragment holds a part of the computational grid and all data associated with it. This hierarchical approach is depicted in Figure 8 and permits an efficient mapping to different execution platforms. For instance, blocks can be mapped to MPI ranks while each fragment inside is handled by a distinct OpenMP thread. Mapping single fragments to accelerators is also possible, as explained later on. The synchronization of data can be controlled by users at layer 4 via *communicate* directives. They specify the field to be communicated and can additionally be parameterized to communicate only certain parts, e.g., specific ghost layers. Each communicate directive triggers the generation of a function that implements the corresponding behavior. This permits the reuse of functions, which can become quite lengthy if the same communication is reissued throughout an application. The function contains code specifying an MPI exchange for fragments residing in different blocks and simple copy kernels for fragments inside the same block. For the interblock case, this includes copying the required data to buffers, calling the appropriate MPI functions, waiting for their completion and copying back data received. The communication buffers required are set up automatically by our framework and, additionally, are shared between multiple communication routines where possible. This minimizes the memory footprint as well as the allocation and deallocation overhead.

Kernels updating data in one fragment can additionally be parallelized. This can be done either with OpenMP or, if available, via an accelerator, e.g., a GPU or reconfigurable hardware such as an FPGA.

Accelerators

To accelerate computations with Nvidia GPUs, we require multiple steps during code generation and developed a back-end to emit corresponding CUDA[2] code. First, fields must be extended to manage a CPU and a GPU version. We also introduce flags that keep track of field data being changed on the host or device side at this point. Next, compute kernels are transformed to their CUDA counterparts and wrapped by an interface function passing variables and fields, used inside the kernel, as parameters. Then we replace the original kernel with a user-controlled condition branching to either the original CPU kernel or the new GPU kernel. Both variants are extended to set the flags previously described after execution. The same flags can also be used to control the use of added copy operations between host and device. This ensures that data is always synchronized correctly while the overhead is minimized.

As an alternative to CUDA, SYCL is a new technology for the inclusion of the OpenCL ecosystem into the world of C++. It strives to combine the strengths of both worlds, e.g., by allowing to use custom data structures and templates inside computational kernels, or by providing an implementation of the parallel C++ Standard Template Library that can be executed not only on multicore CPUs, but also on accelerators. Furthermore, it aims at a reduction of OpenCL boiler-plate code by enabling the direct issuance of `parallel_for` statements instead of forcing the user to declare kernels, map their arguments, etc. Additionally, it detects automatically and initiates memory transfers between host and device.

We had to specialize our code generation workflow slightly to address the characteristics of SYCL. While, in ExaSlang, the dimensionality of a kernel can be arbitrarily large, SYCL only supports up to three-dimensional kernels, i.e., 3D iteration domains. Because of this, we implemented a custom mapping to linearize

[2] CUDA is a proprietary application programming interface and de-facto standard for Nvidia GPUs.

a kernel with more than three dimensions to a single dimension while maintaining correct indexing. Furthermore, SYCL has currently no built-in support for reduction operators, such as the sum of all data items. Since multigrid methods depend on reduction operators for the computation of norms used to approximate the reduction of the residual, we implemented our own reduction operator and used a template function that is called in the corresponding places. Our approach uses standard parallelization techniques that should perform well on CPUs and GPUs.

SYCL separates work items into work groups in the same sense in which these concepts are used by CUDA and OpenCL. While this concept is not paramount, our code generator can optionally specify the granularity of work groups and modify data accesses inside the computational kernels correspondingly.

SYCL is merely a specification, as of yet without a complete and optimized implementation. We verified our SYCL back-end with the commercial product ComputeCpp by Codeplay and also with the experimental open-source implementation triSYCL[3] [33]. Although SYCL still needs to mature, it has great potential as a unified middleware for a multitude of devices: in addition to targeting CPUs and GPUs, the two major vendors of reconfigurable hardware, Xilinx and Intel/Altera, are actively working to accept SYCL code. This will allow us to extend the pool of ExaSlang-supported hardware targets without having to develop a separate code-generator back-end, such as the one presented in the next paragraph.

Reconfigurable Hardware

A particularly interesting target platform for scientific computing is reconfigurable hardware, such as in field-programmable gate arrays (FPGAs). They are increasingly used in high-performance computing because of their high throughput and energy efficiency. We started with extending an existing code generator for imaging processing to produce a description of a simple V-cycle for further processing using C/C++-based high-level synthesis via Vivado HLS for Xilinx FPGAs [68, 69]. This laid the base for a custom Vivado HLS back-end in our code generator, resulting in the ability to emit algorithm descriptions for Vivado HLS stemming from user input in ExaSlang [77]. To test the capabilities of the HLS back-end, we considered a starting grid of size 4096×4096 and used a V(2,2)-cycle. For the coarse-grid solution at a size of 32×32, a corresponding number of Jacobi smoother iterations was applied. The biggest difference to the execution on a CPU or GPU is that all multigrid operators were laid out spatially on the FPGA chip. Another essential consideration in mapping algorithms to reconfigurable hardware is the handling of memory and data transfers. FPGAs provide only very scarce amounts of on-chip memory, called block RAM, that can be accessed with full speed. This implies that data needs to be stored off-chip, e.g., in RAM that can be found on the FPGA board or even in the host computer's memory. As a consequence, these transfers need to be supported by the hardware with corresponding buffers, which are mapped to first-in-first-out (FIFO)

[3] `https://github.com/triSYCL/triSYCL`

hardware buffers. In an FPGA, FIFO buffers can be either composed of registers or implemented by on-chip block RAM.

Another important concern of designing reconfigurable hardware descriptions is the usage of computational resources on the chip. For our initial experiments, we used single-precision floating-point data types and were able to fit the complete design on a mid-range Kintex 7 FPGA. On this chip, our design was able to outperform an Intel i7-3770 in terms of throughput by a factor of approximately 3. Switching from single to double precision does not change any performance aspects but merely requires more chip resources. As a result, we had to switch to the larger Virtex 7 FPGA to house the design but were able to retain the performance advantage.

Based on these results, we worked on incorporating a more relevant coarse-grid solution algorithm into our hardware designs. By a clever arrangement of the required memory buffers and transfers, we were able to implement a conjugate-gradient solver and map it to the Virtex 7 using double-precision floating-point numbers [78]. This algorithm is usually not well-suited for FPGAs because of its random memory-access patterns that break the preferable pipelining model. As a consequence, data needs to be stored on-chip in block RAM to retain performance. However, this takes away resources required by other multigrid components, e.g., smoothers and inter-grid operators. We conducted several experiments and were able to outperform an Intel Xeon E5-1620 v3 in many cases [78]. As an optimization, we overlapped successive V-cycles, i.e., we were able to reuse multigrid operators on the FPGA when they were no longer required by the previous V-cycle iteration provided the data dependences were respected. Theoretically, this should nearly double the throughput, which was confirmed in our numerical experiments.

4.5 Compositional Optimization

The ExaStencils framework enables the automatic generation of geometric multigrid solvers from a high-level representation either by using the generate solver interface or by means of a direct specification of its algorithmic components at layer 3. However, in many cases, the construction of an efficient and scalable multigrid method is a difficult task requiring extensive knowledge in numerical mathematics. Therefore, it is typically performed manually by a domain expert. Part of ExaStencils' vision is to automate all steps from the specification of a problem in the continuous domain (layer 1) to the generation of an efficient implementation on the target platform. To achieve this goal, we construct geometric multigrid methods for solving systems of linear equations as program optimization tasks. We have developed a context-free grammar for the automatic construction of solver instances from a given set of algorithmic options, whose production rules are shown in Figure 9 for the case of pointwise smoothers. Each rule defines the list of expressions by which a certain production symbol, denoted $\langle \bullet \rangle$, can be replaced. To generate a multigrid expression, starting with the symbol $\langle S \rangle$, this process is repeated recursively until the produced expression contains only terminal symbols or the empty string λ. A more

$$
\begin{aligned}
\langle S \rangle &\models \textsc{iterate}(\langle c^h \rangle,\ \omega,\ \langle \mathcal{P} \rangle) \\
\langle c^h \rangle &\models \textsc{apply}(\langle B_h \rangle,\ \langle c^h \rangle) \ \mid\ \textsc{residual}(A_h,\ \langle s^h \rangle) \\
\langle s^h \rangle &\models \textsc{iterate}(\langle c^h \rangle,\ \omega,\ \langle \mathcal{P} \rangle) \\
\langle s^h \rangle &\models \textsc{iterate}(\textsc{coarse-grid-correction}(P_{2h},\ \langle c^{2h} \rangle,\ \omega),\ \omega,\ \lambda) \ \mid\ (u_0^h,\ f_0^h,\ \lambda,\ \lambda) \\
\langle B_h \rangle &\models \textsc{inverse}(\textsc{diagonal}(A_h)) \\
\langle c^{2h} \rangle &\models \textsc{apply}(\langle B_{2h} \rangle,\ \langle c^{2h} \rangle) \ \mid\ \textsc{residual}(A_{2h},\ \langle s^{2h} \rangle) \\
\langle c^{2h} \rangle &\models \textsc{coarse-cycle}(A_{2h},\ u_0^{2h},\ \textsc{apply}(R_h,\ \langle c^h \rangle)) \\
\langle s^{2h} \rangle &\models \textsc{iterate}(\langle c^{2h} \rangle,\ \omega,\ \langle \mathcal{P} \rangle) \ \mid\ \textsc{iterate}(\textsc{apply}(P_{4h},\ \langle c^{4h} \rangle),\ \omega,\ \lambda) \\
\langle B_{2h} \rangle &\models \textsc{inverse}(\textsc{diagonal}(A_{2h})) \\
\langle c^{4h} \rangle &\models \textsc{apply}(A_{4h}^{-1},\ \textsc{apply}(R_{2h},\ \langle c^{2h} \rangle)) \\
\langle \mathcal{P} \rangle &\models \textsc{red-black partitioning} \ \mid\ \lambda
\end{aligned}
$$

Fig. 9: A formal grammar for the generation of multigrid solvers.

detailed description, including the semantics of the expressions generated following this grammar, can be found elsewhere [79].

The effectiveness of an iterative method is determined by two objectives: its rate of convergence and its compute performance on the target platform. To estimate both objectives for a given multigrid solver expression in reasonable time, we employ automated local Fourier analysis (LFA), as described in Subsection 2.2, for the first and a roofline performance model [89] for the second objective. Our goal is to obtain the set of Pareto-optimal solvers with respect to both objectives. A direct enumeration and evaluation of all possible configurations leads to an exponential operation increase with the number of multigrid levels. Thus, a brute-force search is infeasible in most cases. As a remedy, we employ evolutionary computation, a family of stochastic optimization methods stemming from the field of artificial intelligence and inspired by the principle of biological evolution. These methods evolve a population of candidate solutions, which are iteratively improved through the use of so-called genetic operators. The operators are specifically designed for the given problem representation, in our case the set of expressions that can be generated according to the grammar described in Figure 9. After the optimization is completed, we transform the expression of each Pareto-optimal solver to an algorithmic representation which is emitted in the form of ExaSlang 3 code that can then serve to evaluate the solver's rate of convergence and compute performance on the target hardware. As a first step, we have implemented our optimization approach in Python[4], using the libraries LFA Lab [65] and DEAP [23]. We point the interested reader to preliminary results of the optimization of multigrid solvers for the steady-state heat equation on a multi-core CPU [79].

[4] `https://github.com/jonas-schmitt/evostencils`

In the future, we intend to integrate this implementation into the generate solver interface for the automatic construction of efficient and scalable geometric multigrid solvers based on a given specification.

4.6 Feature-Based Domain-Specific Optimization

One new concept that ExaStencils introduced into high-performance computing is that of *feature-based domain-specific optimization* [5]. The central idea is to look at a source program (e.g., a stencil code or application) not as an isolated individual but as a member of a *program family* or *software product line* and to specify it by its commonalities and variabilities with respect to the other family members in terms of features. A *feature* represents a domain concept (e.g., a type of smoother or grid) that may be selected and combined with others on demand. With this approach, a wide search space of configuration choices can be reviewed automatically at the level of domain concepts and the most performant choices for the application and execution platform at hand can be identified.

Features and configuration options can have a significant influence on the performance of the generated code. The influences of some might already be known to the developer of the system, but other influences may be opaque and likewise may be influences that arise from interactions among features, called *feature interactions*. Also, the influences that one may expect based on theory and domain knowledge may not match the actual influences in the program, which often depend on implementation details.

Ideally, a developer or user knows the optimal choices for all features. The goal must then be to exploit *domain knowledge* to identify the most performant configuration. But since, as just explained, domain knowledge is unevenly developed and remains often incomplete, ExaStencils resorts to machine learning to derive a *feature-specific performance model* [81] that provides a comprehensible description of the effects of features and feature interactions on performance. The machine-learning techniques employ a set of configurations, the *learning set* or *sample set*, as input to learn from.

To this end, we have developed a *framework of configuration sampling and machine learning approaches* [36, 39, 81, 59] that allow us to derive a performance model of a given code that is *parameterized in terms of its features*. This way, we can express performance behavior in terms of concepts of the domain and determine automatically optimal configurations that are tailored to the problem at hand, which we have demonstrated in the domain of stencil codes [30, 32, 31] and beyond (e.g., databases, video encoders, compilers, and compression tools) [36, 39, 81]. Our framework integrates well with the other parts of ExaStencils that use and gather domain and configuration knowledge in different phases. It is similar in spirit to the performance modeling tool Extra-P, developed in the SPPEXA projects Catwalk und ExtraPeak [14], though we concentrate on flexibility in choosing and combining different sampling and learning techniques.

To demonstrate the usefulness of the machine-learning approach, we performed experiments on a large number of configurable software systems from different domains, among them different multigrid solvers including an implementation in ExaSlang [30, 31, 32]. To settle on a suitable learning set, we analyze the influence of different sampling strategies on the accuracy of the performance-influence models [81].

Performance-influence models are not only meant to determine optimal configuration but also to *validate and refine domain knowledge*. We must be sure that the observed influences on the performance of the system match with our postulated knowledge, otherwise the system or the knowledge base must be revised. In a case study of a multigrid system working on triangular meshes, albeit not written in ExaSlang, we were able to validate the domain knowledge of the developers of the system [31].

For complex systems, learning a performance-influence model that captures accurately the performance behavior of all individual configurations is often time-consuming and the resulting models can become very large (i.e., many terms describing only a small influence on the performance). However, it is not always necessary to learn such a complex model; it might be beneficial to learn faster a simpler model that is less accurate but accurate enough for a given use-case [39]. In a set of experiments, we have demonstrated the tradeoff between accuracy, complexity, and time to learn a performance-influence model and shown that even simple models can predict all configurations with high accuracy. Depending on the use case of the performance-influence model, it can be of greater value to get quickly a rough impression of the most relevant influences of configuration options rather than waiting for a long time for a more detailed but also more complex model.

5 Case Studies

An essential ingredient of ExaStencils was a collection of case studies to test our approach for performance, target performance, and versatility. The more illustrative ones are sketched in this section. Section 5.1 deals with scalar elliptic PDEs, Section 5.2 with image processing applications and Section 5.3 with aspects of computational fluid dynamics. Section 5.4 details how generated solvers can be coupled with existing code, in our case molecular dynamics simulations. Finally, Section 5.5 goes beyond code generation and discusses a design decision study of porous media applications.

5.1 Scalar Elliptic Partial Differential Equations

A prominent and well-researched PDEs example that describes the steady state of the distribution of a physical quantity $u : \mathbb{R}^d \to \mathbb{R}$, such as heat in a solid medium, is

$$\begin{aligned} -\nabla \cdot (a\nabla u) &= f \quad \text{in} \quad \Omega\,, \\ u &= g \quad \text{on} \quad \partial\Omega \end{aligned}$$

for a given domain Ω of dimensionality d, suitable boundary conditions, right-hand side $f : \mathbb{R}^d \to \mathbb{R}$ and a thermal conductivity $a : \mathbb{R}^d \to \mathbb{R}$ of the material to be simulated. Assuming a finite-difference discretization on a uniform grid, it can be observed that a constant a leads to a stencil with constant coefficients, whereas a variable a leads to one with variable coefficients. In the latter case, we always store all stencil coefficients, rather than values of a, unless specified otherwise. Lastly, if a is 1, the equation simplifies to the Poisson equation introduced in Section 3.2, which is given by $-\nabla^2 u = -\Delta u = f$.

In previous publications [75, 49, 44, 43], we demonstrated our ability to generate highly optimized and massively parallel geometric multigrid solvers for this model problem. For example, we achieve weak scalability up to the full *JUQUEEN* supercomputer, i.e., using up to 458 752 cores across 28 867 nodes, for moderately sized problems of 1 to 16 million unknowns per core. This is a good result considering that the ratio of data to be communicated, e.g., via MPI, is quite high compared to the required amount of computation. Figure 10 summarizes the obtained time to solution for up to $7.3 \cdot 10^{12}$ unknowns using a hybrid MPI/OpenMP parallelization for which one V(3, 3)-cycle takes about 3.6s. A similar scaling behavior can also be observed in a scaling experiment on *Piz Daint* (see Figure 11). With a largest problem of $7.3 \cdot 10^{12}$ unknowns solved, the ExaStencils software reaches for the Poisson equation a size comparable to the finite-element approach [27] developed in the TerraNeo project [8], where the solution of a stabilized tetrahedral linear finite-element discretization of a Stokes system is demonstrated with $1.7 \cdot 10^{13}$ unknowns. This is based on a Uzawa-type smoother [20] for the Stokes system. A first direct comparison has been made recently [41].

In a further single-node case study, we compared ExaStencils against the High-Performance Geometric Multigrid (HPGMG) benchmark [2]. Here, on one core of an Intel Xeon E5-2630 v2, the vectorized code generated by Athariac (see Subsections 4.2 and 4.3) was able to solve 14.6 times as many unknowns per second as HPGMG. A closer inspection, using likwid [84], revealed that this performance gain is mainly due to a higher MFLOPS rate and a much better memory bandwidth exploitation of the solvers generated by ExaStencils. Further details, including also software productivity metrics (e.g., lines of code and Halstead complexity measures), are available elsewhere [74]. As an example, see also our case study on molecular dynamics (Subsection 5.4).

5.2 Image Processing

Next, we consider applications in variational image processing. State-of-the-art denoising algorithms based on total generalized variation have been implemented in ExaSlang [26] using a preconditioned Douglas-Rachford iteration for the under-

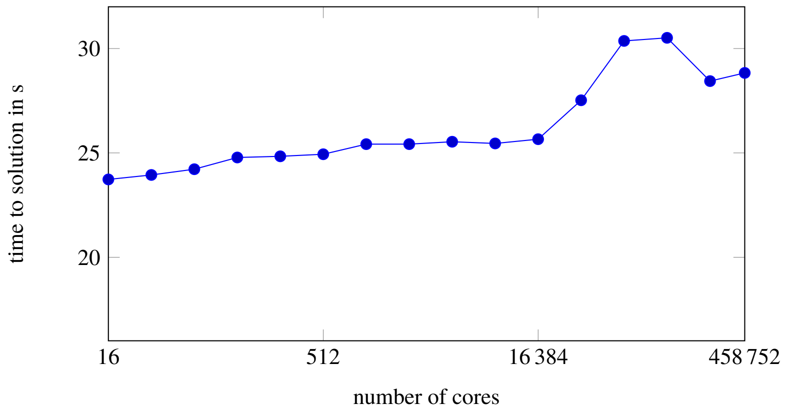


Fig. 10: Weak scaling for generated multigrid solvers using V(3,3)-cycles to solve for Poisson's equation in 3D on *JUQUEEN* [49, 47]. The largest problem solved consists of $7.3 \cdot 10^{12}$ unknowns.

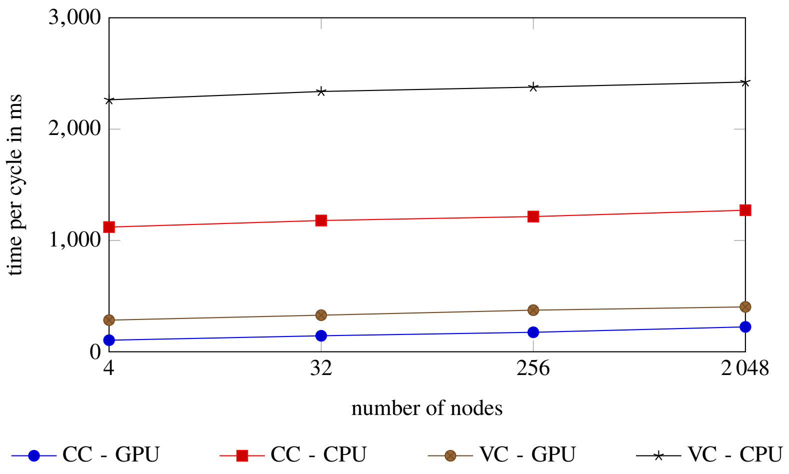


Fig. 11: Weak scaling on *Piz Daint* for constant coefficients (CC) and variable coefficients (VC). In all cases, the number of iterations required is nearly constant. Performance differences between CPU and GPU reflect the hardware characteristics.

lying saddle-point problem. Results show a speedup of more than 4 compared to a reference Matlab implementation on CPU. As an indication of the capability of solving systems of PDEs as well, our most prominent imaging case study is the implementation of optical flow solvers.

Computation of the optical flow refers to the approximation of the apparent motion between two or more images which are part of an image sequence I. We used this application to illustrate ExaSlang's higher-dimensional data types [76].

Let us assume that $I(x, y, t)$ refers to a specific pixel (x, y) in an image sequence at time t, where t could refer to a certain frame in a video stream. Furthermore, we assume that a moving object's intensity does not change over time. Then, for small movements (corresponding to small time differences between two images), we may describe the movement of an intensity value at a pixel (x, y, t) as follows:

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

After performing a Taylor expansion and reordering factors, we can define the partial image derivatives $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$ and $I_t = \frac{\partial I}{\partial t}$. This leads to the temporal gradient $\nabla_\theta I = (I_x, I_y, I_t)^T$ and the optical flow vector $(u, v) = \left(\frac{dx}{dt}, \frac{dy}{dt}\right)$, which can be transformed to the following system of PDEs to be solved:

$$\begin{aligned} -\alpha\Delta u + I_x(I_x u + I_y v) &= -I_x I_t \\ -\alpha\Delta v + I_y(I_x u + I_y v) &= -I_y I_t \end{aligned}$$

Here, α denotes the diffusion coefficient, which we set to 1 in this example. Furthermore, we set the time gradient I_t also to 1 for simplification purposes. After discretization, we obtain the following stencil:

$$\begin{pmatrix} & \begin{pmatrix} -\alpha & \\ & -\alpha \end{pmatrix} & \\ \begin{pmatrix} -\alpha & \\ & -\alpha \end{pmatrix} & \begin{pmatrix} 4\alpha + I_x^2 & I_x I_y \\ I_x I_y & 4\alpha + I_y^2 \end{pmatrix} & \begin{pmatrix} -\alpha & \\ & -\alpha \end{pmatrix} \\ & \begin{pmatrix} -\alpha & \\ & -\alpha \end{pmatrix} & \end{pmatrix}$$

This stencil can be directly mapped to Exaslang and used in a smoother kernel, as illustrated in Listing 14.

All corresponding variables, such as the unknowns u and v, are represented in ExaSlang as vectors of corresponding dimensionality. Thanks to our code generation approach, we can easily conduct parameter studies. For example, we varied the number of pre- and post-smoothing steps between 0 and 6 to study their influence on the convergence rate (see Christian Schmitt's dissertation [72] for details). Furthermore, we switched vectorization for Intel's AVX2 instruction set on and off. For our measurements, we used a single compute node equipped with an Intel i7-6700 CPU. We varied the number of threads between one (1T) and eight (8T) and used likwid [84] for our measurements, which is a supported back-end for ExaSlang's profiling capabilities.

A comparison of the energy consumption in relation to the application's execution time is depicted in Figure 12. Clearly, the application is limited by the available

```
loop over fragments {
loop over Flow@current {
  Flow[next]@current = Flow[active]@current + (
    ( inverse ( diag ( SmootherStencil@current ) ) ) *
    ( RHS@current -
        SmootherStencil@current * Flow[active]@current )
    )
}}
advance Flow@current
```

Listing 14: Jacobi smoother definition using slots for the flow field.

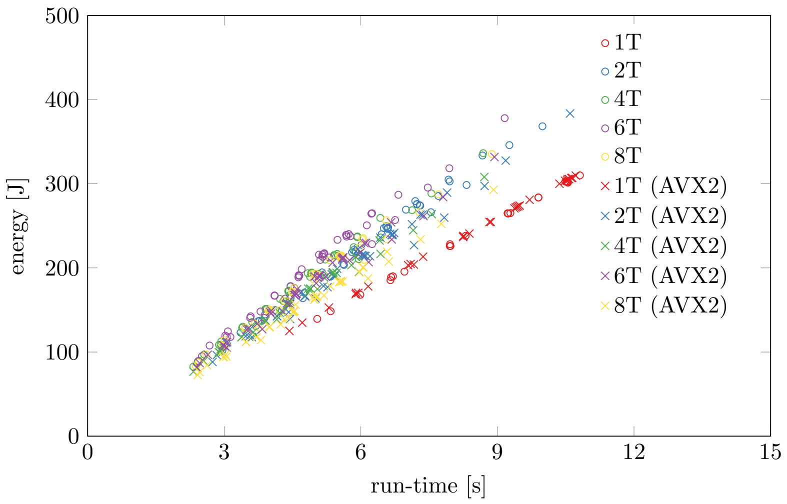


Fig. 12: Parameter study of different parallelization approaches and V-cycle configurations for the two-dimensional optical flow application.

memory bandwidth, as a speedup can be observed when going from one to two threads, but additional usage of CPU cores does not improve the speedup further. The use of AVX2 yields a small performance improvement compared to the scalar variants and, consequently, results in lower energy consumption. The V-cycle configurations that performed best in this case study are the V(4,2)-cycle, the V(3,3)-cycle, and the V(2,4)-cycle.

5.3 Computational Fluid Dynamics

Our next area of application is computational fluid dynamics (CFD), where we considered the solution of Stokes and Navier-Stokes equations [47, 48].

Let us turn first to the Stokes equations as introduced in Subsection 2.3. We discretized the given linear system of PDEs using finite differences and volumes on staggered grids. Additionally, more advanced multigrid components are required. This includes overlapping block smoothers that may additionally be colored. In ExaSlang, both can be expressed concisely and intuitively, as demonstrated in Subsection 4.1. Choosing a suitable coarse-grid solver is also a highly efficient process in our framework as it can be configured via our generate solver directive. Currently, we support conjugate gradient (CG), conjugate residual (CR), biconjugate gradient stabilized (BiCGSTAB) and minimal residual (MINRES), all with optional restart mechanisms, as well as simply applying the smoother. Of course, implementing one's own solver at layer 3 or layer 4 is still possible. We found that, in the collection of variants above, a BiCGSTAB with an added restart mechanism works reasonably well for the present test case [47].

Extending this work towards the full Navier-Stokes equations is the next step. For incompressible fluids, they can be given as

$$\begin{aligned} \frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\, \boldsymbol{u} - \nu \nabla^2 \boldsymbol{u} + \frac{1}{\rho} \nabla p &= \boldsymbol{f}_{\mathbf{u}} \\ \nabla \cdot \boldsymbol{u} &= 0, \end{aligned}$$

on a given domain Ω, which is square in 2D and cubic in 3D, with suitable boundary conditions and with $\mathbf{u}$, p and $\mathbf{f_u}$ as before, ρ as the density and ν as the kinematic viscosity given by the ratio of the dynamic viscosity μ and ρ. The switch from Stokes to Navier-Stokes raises two challenges. First, since the equations become time-dependent, a suitable time integration is required. We choose a traditional implicit Euler, which can be implemented easily at layer 3 or layer 4. For each time step, one can then call either a generated solver variant or a custom implementation. Second, the employed solver has to be able to deal with non-linear equations. To this end, we allow expressing suitable linearizations, such as those based on Newton's method and Picard iterations [47, 21], at layer 2. The linearization chosen can be applied either locally or globally, and both variants can be expressed in tandem with the discretization approach at layer 2. Then, non-linear multigrid solvers based on the full approximation scheme can be tailored at layer 3 automatically. Scaling tests on the *JUWELS* supercomputer exhibit good strong scalability for $2.7 \cdot 10^8$ unknowns and good weak scalability on up to 24 576 cores, the largest amount available to us at the time of the experiment [47]. The results are summarized in Figures 13 and 14.

This approach can also be extended to the scope of our most sophisticated application, namely the simulation of non-Newtonian and non-isothermal fluids. Originally, we ported a legacy FORTRAN code to ExaSlang 4 [48] resulting not only in a considerable reduction in code complexity and size of about one order of magnitude, but also in a reasonable gain in performance of also about one order of magnitude

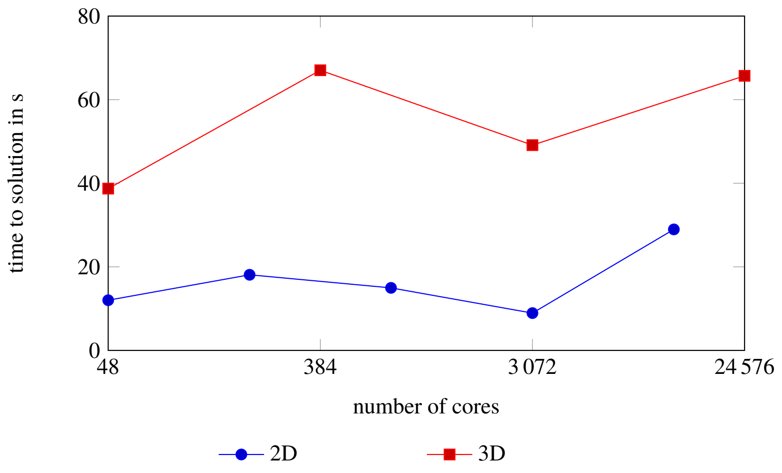


Fig. 13: Weak scaling of generated multigrid solvers for the Navier-Stokes equations in 2D and 3D on *JUWELS* [47].

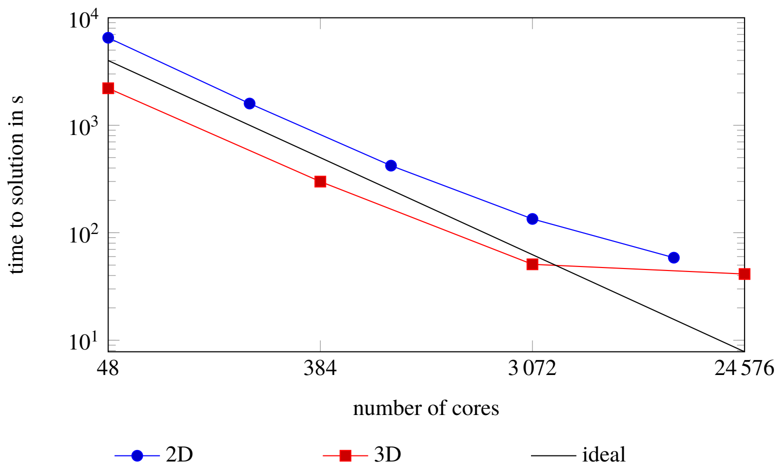


Fig. 14: Strong scaling of generated multigrid solvers for the Navier-Stokes equations in 2D and 3D on *JUWELS* [47].

on average [48]. One prerequisite is the support of staggered non-uniform grids by our code generation framework and DSL, which we extended accordingly. After implementing fully parallel grid setup routines, we are now also able to generate applications executable on GPUs and cluster architectures from the same ExaSlang code. This application also serves as a motivation to extend our generator to apply domain-specific optimizations, such as our sophisticated loop-carried common subexpression elimination [45], which are able to improve performance even further. Finally, using the previously described layer 2 and layer 3 capabilities allows for even more expressive and concise specifications, which have moreover a greater ability to reflect concepts familiar to domain scientists.

Lastly, we should mention that we invested some time to explore alternative approaches, such as the Lattice-Boltzmann method, which can also be implemented in our DSL [64]. An early evaluation showed that applications generated with the ExaStencils framework are about a factor of 2 slower compared to the state-of-the-art multiphysics framework waLBerla [5].

5.4 Molecular Dynamics Simulation

To illustrate the versatility of the ExaStencils approach, we demonstrated that the ExaStencils compiler could be used to generate a multigrid solver that integrates into a large, existing simulation code. For this purpose, we considered a molecular dynamics simulation code used in practice [71, 70], a simulation that computes the motion of the electrons and nuclei of molecules or groups of molecules.

The simulation is a so-called *real-space density functional theory* (RSDFT) simulation. A simulation in real space computes the wave functions on a discretized Cartesian grid, which avoids costly computations of the fast Fourier transform (FFT) that methods which work in Fourier space have to perform. The simulation computes the motion of the particles using the Kohn-Sham density functional theory, which is implemented using a Car-Parrinello molecular dynamics scheme.

The RSDFT simulation requires the computation of the electron-electron Coulomb interaction potential of the charge density. This potential is computed efficiently by solving the corresponding Poisson equation using a multigrid method. The RSDFT code contains a manually derived multigrid solver, which we wanted to replace by a solver generated by the ExaStencils compiler.

However, integrating a solver into an existing code base requires some additional steps above the ones necessary to generate a stand-alone solver. The RSDFT code has its own set of grid data structures for storage of the wave functions. To obtain optimal performance, the ExaStencils compiler generates a set of internal data structures that are the most suitable for the computation (see Subsection 4.3). These data structures are usually not compatible with the ones an application code uses. While it would be possible to force the compiler to use the RSDFT data structures, it would deprive the compiler of some of its optimization opportunities. As a workaround, we used the compiler to generate a set of data transfer routines that copy the grid functions from the RSDFT data structures to the internal ones and vice versa.

In Table 1, we compare the performance of the automatically generated multigrid solver and the legacy solver that was manually written in C. Note that both solvers

[5] www.walberla.net

Table 1: Timings of the solution of the Poisson equation discretized on a grid of n points on *JURECA* [42] using the legacy solver and an automatically generated multigrid solver.

	Legacy			ExaStencils		
Cores	Iterations	Time	Time/Iteration	Iterations	Time	Time/Iteration
$n = 127^3$						
1	6	0.87 s	1.4×10^{-1} s	6	0.76 s	1.3×10^{-1} s
2	6	0.51 s	8.5×10^{-2} s	6	0.39 s	6.4×10^{-2} s
4	6	0.31 s	5.1×10^{-2} s	6	0.27 s	4.5×10^{-2} s
8	6	0.27 s	4.4×10^{-2} s	6	0.13 s	2.1×10^{-2} s
$n = 255^3$						
1	28	32.3 s	1.15 s	6	6.00 s	1.00 s
2	28	17.2 s	0.61 s	6	3.01 s	0.50 s
4	28	9.70 s	0.35 s	6	1.67 s	0.28 s
8	28	5.72 s	0.20 s	6	1.01 s	0.17 s
16	28	3.86 s	0.14 s	6	0.84 s	0.14 s
24	28	2.94 s	0.10 s	6	0.44 s	7.3×10^{-2} s

implement different coarse-grid solution strategies. Thus, the overall timings are not directly comparable, because the number of iterations change between the implementations. However, the time per iteration is indicative for the quality of the code of the generated solver, and we see that it is comparable to the time per iteration of the legacy code. The generated code is often slightly faster.

We can conclude that the ExaStencils approach is also feasible in situations that involve an already existing codebase.

5.5 Porous Media

A further study, in cooperation with SPPEXA project EXA-DUNE [7], concerned the design decision made for stencil codes in the domain of porous media [29]. The goal of this study was to help application engineers understand the complexity of stencil computations and the relation between the mathematical model of an application and the stencil used to solve the partial differential equation underlying the application. To model these design decisions, we use feature orientation as introduced in Subsection 4.6. An example of a design decision is the choice of the type of boundary conditions that have to be satisfied by a given application. We differentiate between decisions at the mathematical and the stencil level, and we devise one feature model for each of these levels. Since decisions at the mathematical level may affect decisions that can be made at the stencil level, we also include constraints between these two levels.

To demonstrate the usefulness of feature modeling, we considered a set of applications that deviate from the standard 5-point stencil used in many benchmarks. We demonstrated that feature models can be used beyond simple mathematical problems by considering problems of different complexity. Specifically, we started with the simple heat equation and went on to more complex equations, such as the advection-diffusion equation with operator splitting or Richard's equation. For all of these equations, we were able to express the design decisions at the mathematical and the stencil level as well as their dependences. The result is a comprehensive set of feature models that can be uses to formally reason about the variabilities in stencil codes (e.g., to identify dependencies among configuration decisions).

6 Variants of the ExaStencils Approach

6.1 ExaSlang 4 Embedded, not External

In the second funding period, ExaStencils was part of an international collaboration with Shigeru Chiba, co-funded by the Japan Science and Technology Agency (JST). This project has already been reported to and reviewed by JST as part of the Japanese strategic basic research programme CREST [17]. The summary is also available at the author's Web site[6].

The project aimed at a software architecture for embedded domain-specific languages. The cooperation of the DFG and JST in SPPEXA gave Chiba's team the opportunity to explore the applicability of its software architecture in the context of ExaStencils. ExaSlang 4 is a standalone language that needs its own compiler and development environments, such as an editor and a debugger, and that has a relatively large development cost. The principle of embedding is an approach to reduce the size and cost of the infrastructure needed. An embedded DSL is implemented on top of a general-purpose host language as a library with a programming interface that looks like a language. Since an embedded DSL is a library, its development cost is usually smaller and programmers can reuse the environments of the host language.

The result of the project was an embedded-DSL architecture based on run-time metaprogramming. In this architecture, a part of the running program is *reified* at run time; that is, its abstract syntax tree is dynamically extracted. Then it is translated to efficient binary code to be run. For example, the host language could be Java, while the program is translated to C/C++ and compiled to machine code. The idea is to view the extracted code as a program not in the host language, but in the DSL with a slightly different semantics. Hence, we can apply domain-specific optimization, such as partial evaluation, during translation.

To evaluate this architecture, Chiba's team developed two software systems. One is Bytespresso [15], which they used to implement an embedded DSL that mimics Exaslang 4. They were successful in developing the DSL with a similar programming

[6] `https://post-peta-crest.github.io/chiba/`

```
Fragments.loop_over do
  Flow_u.current.loop_over do
    Flow_u[:next].current = Flow_u[:active].current +
     ((1.0 / diag(SmootherStencil_u.current)) *
      (RHS_u.current
        - SmootherStencil_u.current * Flow_u[:active].current))
  end
end
```

Listing 15: Specification of an optical flow solver in Ruby-embedded ExaSlang 4.

interface and then measured the execution performance. Although the computational part alone experienced a loss by a factor of 3, the total execution including compilation was twice as fast in the largest case measured: 12 grid levels[7], which is still small-scale. This demonstrates that, for small-scale problems, the embedded version of Exaslang 4 is more suitable for interactive execution than the external version.

Chiba's team also explored the expressive power of the architecture it proposed in Yadriggy [16], a framework for embedding DSLs in Ruby. Compared to the ExaSlang 4 implementation in Java using Bytespresso, exhibiting a source code syntax far from the original, the Ruby-embedded version on Yadriggy features a syntax much closer to the original. This is due to the flexible syntax of Ruby and the development support by Yadriggy. For example, the code fragment depicted in Listing 15 is ExaSlang 4 code taken from the optical flow case study (Subsection 5.2, Listing 14) embedded in Ruby. This internal DSL code is not interpreted as Ruby code and, hence, the language does not have to adhere to Ruby's semantics.

6.2 A Multigrid Solver in SPIRAL

Forerunner and motivator for the vision of ExaStencils was the US project SPIRAL [61, 24]. SPIRAL's initial and foremost domain has been linear transformations. Recently it went on to small-scale linear algebra [82]. ExaStencils' domain is a subdomain of multigrid computations. At project halftime, we put a very simple case of multigrid on SPIRAL: a solver with a Richardson smoother for a discretized square 2D Poisson equation with Dirichlet boundary conditions [10]. The central step is to bring the smoother into an algebraic form of about a dozen rewrite equations. It was an effort of a few days, but the present implementation of SPIRAL does not support the broader domain of ExaStencils. For the generation of efficient code, special adaptations are required as illustrated in the Bachelor thesis of Sebastian Schweikl [80]. This is a consequence of the fact that SPIRAL's target domain has

[7] Figures of the measurements can be found in the CREST report [17].

been signal processing and not the solution of PDEs. The applicability of SPIRAL in its present form to our domain is very limited.

7 The Legacy of ExaStencils

7.1 Effort

ExaStencils was not the only project in SPPEXA to address the solution of PDEs via multigrid methods. However, it was the only one that took a revolutionary rather than evolutionary software approach, i.e., that did not build on existing software tools or application software but started afresh with the automatic, domain-specific synthesis and optimization of application software via a set of dedicated domain-aware software tools. The challenge was to identify the domain knowledge that is useful in helping an optimization, to build the domain-aware tools and to generate some target codes that demonstrate that this approach is realistic and promising.

A central ingredient of the approach is the stratification of the DSL ExaSlang into four layers of abstraction, each addressing a different set of aspects of the stencil code. The domain knowledge is twofold. Knowledge of the application problem is provided by hints that enable the ExaSlang compiler to optimize at the next-more concrete layer. These hints can be supplied as code annotations or conceptually by other means, e.g., configuration files or GUIs, as is the case in the ExaSlang Level 1 editor documented in Tim Ammenhäuser's Bachelor's thesis [4]. Knowledge of the execution platform is provided by a dedicated platform description language.

The main effort in the project concerned the design and implementation of the ExaSlang compiler and code generator which had to be built from scratch, since ExaSlang is an external DSL. This effort led to three dissertations of the six that emerged from the project [43, 47, 72] and two habilitation theses [40, 35]. In a sideline, a comparative study was made with an internal DSL mimicking the most concrete layer of the external ExaSlang, which proved the approach doable but not the first choice for exascale software.

Another major effort was to drive the development of performance prediction further. On the software side, this will soon be realized by one cumulative dissertation introducing the, also revolutionary, generation of performance prediction models via machine learning in the setting of software product line engineering [28]. On the mathematics side, a dissertation drove local Fourier analysis for the convergence prediction of multigrid codes further [66]. Another dissertation in the realm of multigrid math drove the technology of multigrid smoothers forward [19].

The final major element was the case studies conducted to illustrate the impact of the ExaStencils approach.

7.2 Outreach

Let us sketch how ExaStencils profited from and nurtured other research projects.

There was a bilateral exchange of ideas and best practices with SPPEXA project TerraNeo [8] that deals with multigrid solvers on block-structured grids for applications in geoscience. Both ExaStencils and TerraNeo have dealt in loose collaboration with high-performance multigrid codes. One result was an evaluation from the perspective of code generation technology [41]. A further systematic comparison of the performance regarding the algorithms, data structures and their implementation is left to the future. However, the generative programming technologies of ExaStencils are already being leveraged for the redesign of the TerraNeo code basis [38], where they help to simplify the software structure and to reduce the coding effort.

In cooperation with SPPEXA project EXA-DUNE [7], we developed a semi-automatic variability extraction approach that generates a family of applications based on a given application [29]. An ExaStencils-generated application solves the same problem as the EXA-DUNE application but uses sets of methods different from the DUNE framework [9] with the goal of optimizing performance. For example, applications might use different grid implementations or different finite-element maps provided by the DUNE framework. The goal of this approach is to ease the burden of having to understand the DUNE framework when adapting a given application to new use cases and optimizing performance. In a first evaluation, the approach was able to identify over 90% of the alternative but compatible methods a developer of the framework has identified. Using this automated approach, we were even able to identify a bug and an inconsistency in the DUNE framework.

A molecular dynamics simulation on the application platform RSDFT [71] profited from the ExaStencils approach. The port of a development in ExaSlang to RSDFT was successful.

Applied researchers in triangular meshes profited from the ExaStencils approach of validating and refining domain knowledge via a performance-influence model [31]. In simpler words, an automatic learning procedure confirmed their design choices and their effects on performance and made them aware of others.

In a technically related BMBF-funded project, called HighPerMeshes, which also aims at creating a DSL but for the scientific-computing domain of unstructured meshes [37], the parallelization technology SYCL was evaluated [3]. This fueled our interest in creating a SYCL back-end for our code generator which, in turn, provided valuable insights into possible code generation strategies and optimizations for project HighPerMeshes.

SPIRAL [24, 61] was ExaStencils' main motivator and an orientation point for positioning its contribution [10]. At present, SPIRAL is not as serious a platform for multigrid solver development as ExaStencils and it would support the algebraic rather than the geometric form of multigrid.

Similarly, HIPA^{cc} [55, 56], a DSL embedded in C++ and a source-to-source translator for image processing applications that can target a wide variety of parallel accelerator architectures, inspired us on how to deal with both domain and architecture knowledge. In turn, at the start of ExaStencils, we introduced language constructs to HIPA^{cc} to process and represent data at different resolutions, which enables the specification of applications that work on image pyramids as well as 2D stencil-based multigrid methods. By decoupling the algorithm from its schedule, HIPA^{cc} allows the generation of efficient stencil-code implementations for single accelerators [57].

A recent DFG-funded project on ocean modeling (grant. no 320126236) challenges the extensibility of ExaStencils' application domain by considering time-dependent hyperbolic partial differential equations, discretized using finite-volume or higher-order discontinuous Galerkin methods on block-structured triangular grids. Since, so far, an explicit time-stepping scheme is being applied, no solvers like multigrid are required. However, it has been shown already that the ExaStencils framework is capable of generating efficient and scalable code for such applications on GPU clusters [50].

7.3 Potential

The ExaSlang compiler and code generator remain at the stage of a prototype, but we believe that they have demonstrated that a stratified optimizing DSL approach can provide dramatically increased convenience for the application programmer and high flexibility concerning the execution platform. The effort of adapting a complex code to a different application or platform can be reduced significantly. The price to be paid is good knowledge of the theory on which the application rests and a significant development cost in implementing the DSL. The latter can be reduced by going the internal rather than the external way, at the price of reduced expressiveness and flexibility in terms of target code optimization.

Our choice not to go with an embedded DSL tends to be not popular in present-day DSL development but is essential. We did not want to be bound by any language limitations in exploring the potential of the ExaStencils approach to domain-specific programming and optimization. To achieve high portability and execution performance of generated programs, we selected C/C++ as the target language.

Our case studies covered the range from textbook examples to realistic applications. The latter are represented by a simulation of non-Newtonian flow, fully developed in the ExaSlang world, and a molecular dynamics simulation which used an ExaStencils solver through an automatically generated compatibility interface. We have demonstrated that we can reach the expected performance, achieve portability to most contemporary HPC platforms, and profit from a substantial decrease of development time of new PDE models that can be expressed in ExaSlang.

Since the start of our project, code generation has become a standard technique in many HPC codes and, thus, the concepts developed in ExaStencils are used or can be adapted to other domains. Currently, a popular approach is the generation of single compute kernels from an embedded DSL for existing software frameworks. In addition, C++ template-based embedded DSLs are common. Frequently, both the concrete DSL and the resulting implementations are very application-specific, but the intermediate representations and code transforms can be defined quite generally and then optimized for the specific case. In the future, it should be possible to extend the ExaStencils approach to other HPC applications to enable more holistic optimizations than currently established approaches.

Acknowledgements

Project ExaStencils received generous funding from the Deutsche Forschungsgemeinschaft (DFG) in its priority programme SPP 1648 “Software for Exascale Computing” under grant numbers AP 206/7, BO 3405/2, LE 912/15, RU 422/15 and TE 163/17. We thank the Jülich Supercomputing Center (JSC) for providing access to the supercomputers *JUQUEEN* and *JURECA* and the Swiss National Supercomputing Centre (CSCS) for providing computational resources and access to the supercomputer *Piz Daint*. We are grateful to Rochus Schmid for letting us have the RSDFT code for the molecular dynamics simulation.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. The MIT Press, 2nd edn. (1996)
2. Adams, M.F., Brown, J., Shalf, J., Straalen, B.V., Strohmaier, E., Williams, S.: HPGMG 1.0: A benchmark for ranking high performance computing systems. Tech. Rep. LBNL-6630E, Lawrence Berkeley National Laboratory (2014)
3. Afzal, A., Schmitt, C., Alhaddad, S., Grynko, Y., Teich, J., Förstner, J., Hannig, F.: Solving Maxwell’s equations with modern C++ and SYCL: A case study. In: Proc. Ann. IEEE Int’l Conf. on Application-Specific Systems, Architectures and Processors (ASAP), pp. 49–56. IEEE (2018)
4. Ammenhäuser, T.: Online-Editor und Visualisierung für ExaStencils. Bachelor’s thesis, University of Wuppertal (Apr 2019)
5. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer-Verlag (Oct 2013)
6. Bandishti, V., Pananilath, I., Bondhugula, U.: Tiling stencil computations to maximize parallelism. In: Proc. Conf. on High Performance Computing Networking, Storage and Analysis (SC). pp. 40:1–40:11. IEEE Computer Society (2012)
7. Bastian, P., Altenbernd, M., Dreier, N.A., Engwer, C., Fahlke, J., Fritze, R., Geveler, M., Göddeke, D., Iliev, O., Ippisch, O., Mohring, J., Müthing, S., Ohlberger, M., Ribbrock, D., Turek, S.: EXA-DUNE: Flexible PDE solvers, numerical methods and applications. In this volume
8. Bauer, S., Bunge, H.P., Drzisga, D., Ghelichkhan, S., Huber, M., Kohl, N., Mohr, M., Rüde, U., Thönnes, D., Wohlmuth, B.: TerraNeo – Mantle convection beyond a trillion degrees of freedom. In this volume
9. Blatt, M., Burchardt, A., Dedner, A., Engwer, C., Fahlke, J., Flemisch, B., Gersbacher, C., Gräser, C., Gruber, F., Grüninger, C., Kempf, D., Klöfkorn, R., Malkmus, T., Müthing, S., Nolte, M., Piatkowski, M., Sander, O.: The distributed and unified numerics environment, version 2.4. Archive of Numerical Software (ANS) **4**(100), 13–29 (2016)
10. Bolten, M., Franchetti, F., Kelly, P.H.J., Lengauer, C., Mohr, M.: Algebraic description and automatic generation of multigrid methods in SPIRAL. Comput. Pract. Exp. (CCPE) **29**(17), 4105:1–4105:11 (Sep 2017), special issue: Advanced Stencil-Code Engineering

11. Bolten, M., Rittich, H.: Fourier analysis of periodic stencils in multigrid methods. SIAM J. Sci. Comput. (SISC) **40**(3), A1642–A1668 (2018)
12. Bondhugula, U., Bandishti, V., Pananilath, I.: Diamond Tiling: Tiling techniques to maximize parallelism for stencil computations. IEEE Trans. Parall. Distr. (TPDS) **28**(5), 1285–1298 (May 2017)
13. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). pp. 101–113. ACM (2008)
14. Calotoiu, A., Copik, M., Hoefler, T., Ritter, M., Wolf, F.: ExtraPeak – Advanced automatic performance modeling for HPC applications. In this volume
15. Chiba, S., Zhuang, Y., Scherr, M.: Deeply reifying running code for constructing a domain-specific language. In: Proc. 13th Int'l Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ), pp. 1:1–1:12. ACM (2016)
16. Chiba, S.: Foreign language interfaces by code migration. In: 18th ACM SIGPLAN Int'l Conf on Generative Programming: Concepts & Experiences (GPCE). pp. 1–13. ACM (Oct 2019)
17. Chiba, S., Zhuang, Y., Dao, T.C.: A development platform for embedded domain-specific languages. In: Sato, M. (ed.) Advanced Software Technologies for Post-Peta Scale Computing, chap. 8, pp. 139–161. Springer Singapore (2019)
18. Christen, M., Schenk, O., Burkhart, H.: PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Proc. 25th IEEE Int'l Symp. on Parallel and Distributed Processing (IPDPS). pp. 676–687 (May 2011)
19. Claus, L.: Multigrid Smoothers for Saddle Point Systems. Ph.D. thesis, University of Wuppertal (2019), 133 pages
20. Drzisga, D., John, L., Rude, U., Wohlmuth, B., Zulehner, W.: On the analysis of block smoothers for saddle point problems. SIAM J. Matrix Anal. Appl. **39**(2), 932–960 (2018)
21. Elman, H.C., Silvester, D.J., Wathen, A.J.: Finite Elements and Fast Iterative Solvers. Numerical Mathematics and Scientific Computation, Oxford University Press, 2nd edn. (2014)
22. Feautrier, P., Lengauer, C.: Polyhedron model. In: Padua et al. [60], pp. 1581–1592
23. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. J. Mach. Learn. Res. (JMLR) **13**, 2171–2175 (Jul 2012)
24. Franchetti, F., Low, T.M., Popovici, D.T., Veras, R.M., Spampinato, D.G., Johnson, J.R., Püschel, M., Hoe, J.C., Moura, J.M.F.: SPIRAL: Extreme performance portability. Proc. IEEE **106**(11), 1935–1968 (Nov 2018), special issue: From High-Level Specification to High-Performance Code
25. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005), special issue: Program Generation, Optimization, and Platform Adaptation
26. Gerecke, M.: Implementierung des TGV Algorithmus mithilfe von ExaSlang. Bachelor's thesis, Friedrich-Alexander University Erlangen-Nürnberg (2017)
27. Gmeiner, B., Huber, M., John, L., Rüde, U., Wohlmuth, B.: A quantitative performance study for Stokes solvers at the extreme scale. J. Computational Science **17**, 509–521 (2016)
28. Grebhahn, A.: Performance Prediction of Highly Configurable Software Systems: Multigrid Systems and Beyond. Ph.D. thesis, Faculty of Computer Science and Mathematics, University of Passau (2020), to be submitted
29. Grebhahn, A., Engwer, C., Bolten, M., Apel, S.: Variability of stencil computations for porous media. Concurrency and Computation: Practice and Experience **29**(17), 4119:1–4119:14 (Sep 2017), special issue: Advanced Stencil-Code Engineering
30. Grebhahn, A., Kuckuk, S., Schmitt, C., Köstler, H., Siegmund, N., Apel, S., Hannig, F., Teich, J.: Experiments on optimizing the performance of stencil codes with SPL Conqueror. Par. Proc. Lett. (PPL) **24**(3), Article 1441001, 19 pages (Sep 2014)
31. Grebhahn, A., Rodrigo, C., Siegmund, N., Gaspar, F.J., Apel, S.: Performance-influence models of multigrid methods: A case study on triangular meshes. Comput. Pract. Exp. (CCPE) **29**(17), 4057:1–4057:13 (Sep 2017), special issue: Advanced Stencil-Code Engineering
32. Grebhahn, A., Siegmund, N., Köstler, H., Apel, S.: Performance prediction of multigrid-solver configurations. In: Software for Exascale Computing – SPPEXA 2013–2015. pp. 69–88. LNCSE 113, Springer (Sep 2016)

33. Groth, S., Schmitt, C., Teich, J., Hannig, F.: SYCL code generation for multigrid methods. In: Proc. 22nd Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES). pp. 41–44. ACM (May 2019)
34. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., Schulthess, T.C.: Stella: A domain-specific tool for structured grid methods in weather and climate models. In: Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC). pp. 41:1–41:12. ACM (Nov 2015)
35. Hannig, F.: Domain-specific and resource-aware computing (Dec 2017), Habilitation thesis, Friedrich-Alexander University Erlangen-Nürnberg, 444 pages
36. Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., Apel, S.: Distance-based sampling of software configuration spaces. In: Proc. IEEE/ACM Int'l Conf. on Software Engineering (ICSE). pp. 1084–1094. IEEE Computer Society (May 2019)
37. Kenter, T., Mahale, G., Alhaddad, S., Grynko, Y., Schmitt, C., Afzal, A., Hannig, F., Förstner, J., Plessl, C.: OpenCL-based FPGA design to accelerate the nodal discontinuous Galerkin method for unstructured meshes. In: Proc. 26th IEEE Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM). pp. 189–196. IEEE (2018)
38. Kohl, N., Thönnes, D., Drzisga, D., Bartuschat, D., Rüde, U.: The HyTeG finite-element software framework for scalable multigrid solvers. Int. J. Parallel, Emergent and Distributed Systems (IJPEDS) pp. 1–20 (2018)
39. Kolesnikov, S., Siegmund, N., Kästner, C., Grebhahn, A., Apel, S.: Tradeoffs in modeling performance of highly-configurable software systems. Software and Systems Modeling (SoSyM) **18**(3), 2265–2283 (Jun 2019)
40. Köstler, H.: Effiziente numerische Algorithmen und Softwareentwicklung für hochparallele Rechensysteme (Aug 2014), Habilitation thesis, Friedrich-Alexander University Erlangen-Nürnberg, 94 pages
41. Köstler, H., Heisig, M., Kohl, N., Kuckuk, S., Bauer, M., Rüde, U.: Code generation approaches for parallel geometric multigrid solvers. Analele Stiintifice ale Universitatii Ovidius, Seria Matematica (2019), accepted and to appear
42. Krause, D., Thörnig, P.: JURECA: Modular supercomputer at Jülich Supercomputing Centre. J. Large-Scale Research Facilities (JLSRF) **4**, A132, 9 pages (2018)
43. Kronawitter, S.: Automatic Optimization of Stencil Codes. Ph.D. thesis, University of Passau (2019), defended and to appear
44. Kronawitter, S., Kuckuk, S., Köstler, H., Lengauer, C.: Automatic data layout transformations in the ExaStencils code generator. Par. Proc. Lett. (PPL) **28**(3), Article 1850009, 18 pages (Sep 2018)
45. Kronawitter, S., Kuckuk, S., Lengauer, C.: Redundancy elimination in the ExaStencils code generator. In: Carretero, J., et al. (eds.) Algorithms and Architectures for Parallel Processing (ICA3PP), Collocated Workshops. pp. 159–173. LNCS 10049, Springer (2016), First Int'l Workshop on Data Locality in Modern Computing Systems (DLMCS)
46. Kronawitter, S., Lengauer, C.: Polyhedral search space exploration in the ExaStencils code generator. ACM Trans. Archit. Code Op. (TACO) **15**(4), 40:1–40:25 (Jan 2019)
47. Kuckuk, S.: Automatic Code Generation for Massively Parallel Applications in Computational Fluid Dynamics. Ph.D. thesis, Friedrich–Alexander-Universität Erlangen-Nürnberg (2019), defended and to appear
48. Kuckuk, S., Haase, G., Vasco, D., Köstler, H.: Towards generating efficient flow solvers with the ExaStencils approach. Comput. Pract. Exp. (CCPE) **29**(17), 4062:1–4062:17 (Sep 2017), special issue: Advanced Stencil-Code Engineering
49. Kuckuk, S., Köstler, H.: Automatic generation of massively parallel codes from ExaSlang. Computation **4**(3), Article 27, 20 pages (Sep 2016), special issue: High Performance Computing (HPC) Software Design
50. Kuckuk, S., Köstler, H.: Whole program generation of massively parallel shallow water equation solvers. In: 2018 IEEE Int'l Conf. on Cluster Computing (CLUSTER). pp. 78–87. IEEE (Sep 2018)

51. Kuckuk, S., Leitenmaier, L., Schmitt, C., Schönwetter, D., Köstler, H., Fey, D.: Towards virtual hardware prototyping for generated geometric multigrid solvers. Tech. Rep. CS 2017-01, Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (Mar 2017), 10 pages
52. Lange, M., Kukreja, N., Louboutin, M., Luporini, F., Vieira, F., Pandolfo, V., Velesko, P., Kazakas, P., Gorman, G.: Devito: Towards a generic finite difference DSL using symbolic Python. In: Prof. Workshop on Python for High-Performance and Scientific Computing (PyHPC). pp. 67–75. IEEE (2016)
53. Lengauer, C., Apel, S., Bolten, M., Größlinger, A., Hannig, F., Köstler, H., Rüde, U., Teich, J., Grebhahn, A., Kronawitter, S., Kuckuk, S., Rittich, H., Schmitt, C.: ExaStencils: Advanced stencil-code engineering. In: Lopes, L., et al. (eds.) Euro-Par 2014: Parallel Processing Workshops, Part II. pp. 553–564. LNCS 8806, Springer (2014)
54. Luporini, F., Varbanescu, A.L., Rathgeber, F., Bercea, G.T., Ramanujam, J., Ham, D.A., Kelly, P.H.J.: Cross-loop optimization of arithmetic intensity for finite element local assembly. Trans. Archit. Code Op. (TACO) **11**(4), 57:1–57:25 (Jan 2015)
55. Membarth, R., Hannig, F., Teich, J., Körner, M., Eckert, W.: Generating device-specific GPU code for local operators in medical imaging. In: Proc. 26th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). pp. 569–581. IEEE (May 2012)
56. Membarth, R., Reiche, O., Hannig, F., Teich, J., Körner, M., Eckert, W.: HIPA^{cc}: A domain-specific language and compiler for image processing. IEEE Trans. Parall. Distr. (TPDS) **27**(1), 210–224 (Jan 2016)
57. Membarth, R., Reiche, O., Schmitt, C., Hannig, F., Teich, J., Stürmer, M., Köstler, H.: Towards a performance-portable description of geometric multigrid algorithms using a domain-specific language. J. Parallel Distr. Com. (JPDC) **74**(12), 3191–3201 (Dec 2014)
58. Mullapudi, R.T., Vasista, V., Bondhugula, U.: PolyMage: Automatic optimization for image processing pipelines. SIGARCH Computer Architecture News **43**(1), 429–443 (Mar 2015)
59. Nair, V., Yu, Z., Menzies, T., Siegmund, N., Apel, S.: Finding faster configurations using FLASH. IEEE Trans. Softw. Eng. (TSE) (2018), online first
60. Padua, D.A., et al. (eds.): Encyclopedia of Parallel Computing. Springer (2011)
61. Püschel, M., Franchetti, F., Voronenko, Y.: Spiral. In: Padua et al. [60], pp. 1920–1933
62. Ragan-Kelley, Jonathan, e.a.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Notices **48**(6), 519–530 (Jun 2013), Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)
63. Rathgeber, F., Ham, D.A., Mitchell, L., Lange, M., Luporini, F., Mcrae, A.T.T., Bercea, G.T., Markall, G.R., Kelly, P.H.J.: Firedrake: Automating the finite element method by composing abstractions. ACM Trans. Math. Softw. (TOMS) **43**(3), 24:1—24:27 (Jan 2017)
64. Ribica, D.: Code Generation vs. HPC Framework. Bachelor's thesis, Friedrich-Alexander University Erlangen-Nürnberg (2018)
65. Rittich, H.: LFA Lab, `https://hrittich.github.io/lfa-lab/`
66. Rittich, H.: Extending and Automating Fourier Analysis for Multigrid Methods. Ph.D. thesis, Faculty of Mathematics and Natural Sciences, University of Wuppertal (Jun 2017), 202 pages
67. van Rossum, G.: Python Reference Manual. Tech. Rep. CS-R9525, Centrum voor Wiskunde en Informatica (CWI) (Apr 1995)
68. Schmid, M., Reiche, O., Schmitt, C., Hannig, F., Teich, J.: Code generation for high-level synthesis of multiresolution applications on FPGAs. In: Proc. First Int'l Workshop on FPGAs for Software Programmers (FSP). pp. 21–26. arXiv:1408.4721 (Sep 2014)
69. Schmid, M., Schmitt, C., Hannig, F., Malazgirt, G.A., Sönmez, N., Yurdakul, A., Cristal, A.: Big Data and HPC acceleration with Vivado HLS. In: Koch, D., Hannig, F., Ziener, D. (eds.) FPGAs for Software Programmers, chap. 7, pp. 115–136. Springer (Jun 2016)
70. Schmid, R., Tafipolsky, M., König, P.H., Köstler, H.: Car-Parrinello molecular dynamics using real space wavefunctions. Phys. Status Solidi (b) **243**(5), 1001–1015 (2006)
71. Schmid, R.: Car-Parrinello simulations with a real space method. J. Comput. Chem. **25**(6), 799–812 (2004)

72. Schmitt, C.: A Domain-Specific Language and Source-to-Source Compilation Framework for Geometric Multigrid Methods. Ph.D. thesis, Friedrich-Alexander University Erlangen-Nürnberg (May 2019), Verlag Dr. Hut, 203 pages
73. Schmitt, C., Hannig, F., Teich, J.: A target platform description language for code generation in HPC. In: Workshop Proc. 31st GI/ITG Int'l Conf. on Architecture of Computing Systems (ARCS). pp. 59–66. VDE (Apr 2018)
74. Schmitt, C., Kronawitter, S., Hannig, F., Teich, J., Lengauer, C.: Automating the development of high-performance multigrid solvers. Proc. IEEE **106**(11), 1969–1984 (Nov 2018), special issue: From High-Level Specification to High-Performance Code
75. Schmitt, C., Kuckuk, S., Hannig, F., Köstler, H., Teich, J.: ExaSlang: A domain-specific language for highly scalable multigrid solvers. In: Proc. 4th Int'l Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC). pp. 42–51. IEEE Computer Society (Nov 2014)
76. Schmitt, C., Kuckuk, S., Hannig, F., Teich, J., Köstler, H., Rüde, U., Lengauer, C.: Systems of partial differential equations in ExaSlang. In: Bungartz, H.J., Neumann, P., Nagel, W.E. (eds.) Software for Exascale Computing – SPPEXA 2013-2015. pp. 47–67. LNCSE 113, Springer (Sep 2016)
77. Schmitt, C., Schmid, M., Hannig, F., Teich, J., Kuckuk, S., Köstler, H.: Generation of multigrid-based numerical solvers for FPGA accelerators. In: Größlinger, A., Köstler, H. (eds.) Proc. 2nd Int'l Workshop on High-Performance Stencil Computations (HiStencils). pp. 9–15. www.viaprinto.de (Jan 2015)
78. Schmitt, C., Schmid, M., Kuckuk, S., Köstler, H., Teich, J., Hannig, F.: Reconfigurable hardware generation of multigrid solvers with conjugate gradient coarse-grid solution. Par. Proc. Lett. (PPL) **28**(4), Article 1850013, 20 pages (2018)
79. Schmitt, J., Kuckuk, S., Köstler, H.: Optimizing Geometric Multigrid Methods with Evolutionary Computation (Oct 2019), arXiv:1910.02749
80. Schweikl, S.: Multigrid for the SPIRAL Prototype in Scala. Bachelor's thesis, University of Passau (Sep 2017)
81. Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-influence models for highly configurable systems. In: Proc. European Software Engineering Conf. and ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering (ESEC/FSE). pp. 284–294. ACM Press (Aug 2015)
82. Spampinato, D., Fabregat-Traver, D., Bientinesi, P., Püschel, M.: Program generation for small-scale linear algebra applications. In: Proc. Int'l Symp. on Code Generation and Optimization (CGO). pp. 327–339. ACM (2018)
83. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proc. 23rd Ann. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). pp. 117–128. ACM (2011)
84. Treibig, J., Hager, G., Wellein, G.: likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: Proc. 39th Int'l Conf. on Parallel Processing Workshops (ICPPW). vol. 1, pp. 207–216 (2010), Int'l Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)
85. Trottenberg, U., Oosterlee, C.W., Schuller, A.: Multigrid. Academic Press (2001)
86. Unat, D., Cai, X., Baden, S.B.: Mint: Realizing CUDA performance in 3D stencil methods with annotated C. In: Proc. Int'l Conf. on Supercomputing (ICS). pp. 214–224. ACM (Nov 2011)
87. Vanka, S.: Block-implicit multigrid solution of Navier-Stokes equations in primitive variables. J. Comput. Phys. **65**(1), 138–158 (Jul 1986)
88. Wienands, R., Joppich, W.: Practical Fourier Analysis For Multigrid Methods. Chapman Hall / CRC Press (2005)
89. Williams, S., Watermann, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Comm. ACM **52**(4), 65–76 (Apr 2009)
90. Zhang, N., Driscoll, M., Markley, C., Williams, S., Basu, P., Fox, A.: Snowflake: A lightweight portable stencil DSL. In: Proc. IEEE Int'l Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 795–804. IEEE (May 2017)