Automating the Development of High-Performance Multigrid Solvers

Christian Schmitt, Stefan Kronawitter, Frank Hannig, Jürgen Teich, Christian Lengauer

***Abstract*—The purpose of a domain-specific language (DSL) is to enable the application programmer to specify a problem, or an abstract algorithm description, in his/her domain of expertise without being burdened by implementation details. The ideal scenario is that the implementation detail is added in an automatic process of program translation and code generation. The approach of domain-specific program generation has lately received increasing attention in the area of computational science and engineering.**

We introduce the new code generation framework Athariac. Its goal is to support the quick implementation of a language processing and program optimization platform for a given DSL based on stepwise term-rewriting. We demonstrate the framework's use on our DSL ExaSlang for the specification and optimization of multigrid solvers. On this example, we provide evidence of Athariac's potential for making domain-specific software engineering more productive.

I. Introduction and Motivation

Contemporary high-performance computing platforms are highly heterogeneous. While more than 95 % of the supercomputing systems ranked in the TOP500 list (November 2017) run on an x86-based CPU architecture, there are major exceptions such as the TaihuLight, Sequoia, and K computer, which use Sunway processors, PowerPCs, and SPARCs, respectively. In the light of energy efficiency, the ARM architecture as well as accelerators such as GPUs, many-core accelerator cards, or field-programmable gate arrays (FPGAs) are becoming increasingly popular [59]. Each of these technologies has its own trade-offs and requires different programming and optimization techniques, which usually vary between vendors and hardware generations. Beyond this issue, there are more details to consider such as the network architecture and hardware, versions of operating system or compiler, and specific communication libraries such as vendor-specific implementations of MPI. Thus, writing programs and optimizing them for a given setting is tedious work that often requires a few months even for an expert who knows all the programming techniques and trade-offs of each platform. If done right, the program will perform comparably on similar hardware. However, in case of any new hardware or platform configuration, the program has to be modified to take advantage of the new features. In many cases, tuning has to be done over and over again—usually by specialized, interdisciplinary teams.

Christian Schmitt, Frank Hannig and Jürgen Teich are with the Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg, Germany

Stefan Kronawitter and Christian Lengauer are with the Faculty of Computer Science and Mathematics, University of Passau, Germany

Here, domain-specific languages (DSLs) have an advantage. They allow the separation of the description of the algorithm from that of its implementation. Thus, when new hardware must be supported, only one program has to be modified: the DSL compiler. That done, it takes only a recompilation of the unaltered DSL programs to make use of the new features. If the DSL compiler supports generic and platform-specific code optimization, this approach leads to better performance, and we have *performance portability*. An impressive example is SPIRAL [46], which was extended to support new Intel SIMD instructions purely on their technical documentation and long before the availability of corresponding hardware, bringing performance improvements to all its applications. SPIRAL was also used to generate some of the Intel MKL library's FFT functions [41] and parts of the Intel IPP library. Another potential advantage is that of *programming productivity*. Since DSLs are programming languages specialized for particular domains, they offer a much higher level of abstraction and lower entry barriers for domain (but not programming) experts by re-using concepts and terms of the target domain [28].

In Project ExaStencils [38], it is our goal to raise the level of abstraction of the specification and the degree of automation of the generation of solvers of partial differential equations (PDEs) with the geometric multigrid method on structured grids. We expect *automation* to achieve two different improvements: (1) to provide users with the liberty of not having to worry about algorithm and implementation details by making the—in whatever sense—best choices for them during the code generation process, and (2) to provide users and developers with a wealth of variants, easily generated from the input code, in the search for the best performance.

For users, we approach this goal by providing a DSL, called *ExaSlang*, that features four layers of abstraction—each targeting a different group of users. As depicted in Fig. 2, abstractions range from the input of continuous equations—ExaSlang 1—to the imperative programming language ExaSlang 4 for the specification of concrete algorithm implementations. The language is supported by our compiler, which allows for the automatic discretization, multigrid component and parameter selection, parallelization and optimization of the equations or code provided. One of the project's main concerns is to explore how far automation can be driven in this domain. Consequently, we decided to design our own language, tailored especially towards the project goals and the feature set we would like to provide to the user via suitable abstractions. Our focus

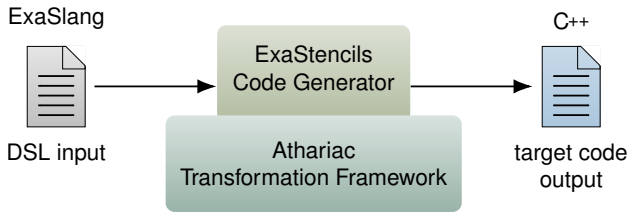


Fig. 1: Relationship between ExaSlang, ExaStencils, and Athariac.

of choice is on multigrid methods [24, 62] since they scale optimally (i.e., linearly with the number of compute nodes), which is a prerequisite for the effective use of future exascale machines. Furthermore, multigrid offers a vast variety of solver implementations since, for every component of the multigrid algorithm, different approaches exist that have mutual advantages and drawbacks.

For the implementation of such an ambitious project, a well-working foundation is needed for the developers mentioned above, e.g., us as developers of ExaStencils. Here, we present our code transformation framework Athariac, which is simple to use, yet a powerful tool for the implementation of compilers of DSLs. We show how Athariac contributes to the implementation of ExaSlang and its compiler by providing data structures to represent user programs and specify transformations—such as optimizations—of the programs. As an example, we present our implementation of two optimizations that are part of our ExaStencils compiler—built on top of Athariac—to demonstrate that implementing DSLs need not be a cumbersome task. Finally, we propose a number of metrics of programming productivity and performance to attest that code generation is a viable approach for multigrid-based numerical solvers in particular.

To summarize the introductions above, Athariac is the code generation framework behind our ExaStencils code generator (we will use the term ExaStencils compiler synonymously). As input, our compiler takes the user's program in our DSL ExaSlang, and its output is in C++ plus MPI, OpenMP, or CUDA. Athariac itself, however, is general enough to work on any input and output language(s). A graphical representation of the relationships is depicted in Fig. 1.

The rest of the paper is structured as follows. Section II commences with some necessary information on the different technologies on which ExaStencils relies. Section III presents related work on programming and abstraction models for the domain of computational science and engineering as well as foundations for the implementation of DSLs, followed by the introduction of our code transformation framework Athariac in Section IV, which was used to implement ExaSlang. Sections V and VI provide implementation details of ExaSlang that make heavy use of our framework's features. Finally, in Section VII, productivity advantages of DSLs are illustrated on the example of ExaSlang, before we conclude and discuss possible future efforts.

II. Background and Basics

Let us introduce some basics of DSLs, followed by an overview of our own DSL ExaSlang. We conclude this section with a brief review of multigrid methods, our application domain.

A. Domain-Specific Languages

Modeling complex real-world scenarios in a machine-readable form is more accessible for domain experts when they can use concepts, objects, and terms of their domain. These abstractions can be provided in a number of different ways. One of the most popular approaches is by implementing a domain-specific language (DSL) [43]. Such a language can be created by extending and/or restricting an existing programming language. Quite often, this *host* language is a general-purpose language. The derived language is called an *embedded*, or *internal*, DSL. Alternatively, it is also possible to create a new language, a so-called *external* DSL, from first principles [18].

The implementation of the DSL compiler or interpreter depends on the type of DSL. Usually, for an internal DSL, the host language compiler is modified whereas, for an external DSL, a new compiler or interpreter must be implemented. Consequently, an external DSL requires a higher implementation effort, but also provides greater freedom in the choice of syntax and semantics.

There are a number of popular domain-specific languages. The following is (by far) non-exhaustive list of popular domain-specific languages: R has been proposed for statistic computations, LaTeX for typesetting of scientific articles, and OpenGL for graphics. More generally, SQL is ubiquitous in the realm of databases. Another example is Extensible Stylesheet Language Transformations (XSLT) for the transformation of XML documents to other XML files, but also other formats, such as PDF, PNG, or HTML.

B. ExaSlang

ExaSlang aims at providing suitable abstractions for three different groups of users: (1) engineers and application scientists, (2) mathematicians, and (3) computer scientists. Since each group has different needs and requirements, ExaSlang is not a monolithic language, but a hierarchy of four different language layers, as depicted in Fig. 2.

- ExaSlang 1 is the most abstract layer. It aims at providing a suitable workbench for engineers and scientists who need to solve a PDE as part of a bigger project and, thus, are interested in getting the result as quickly and exactly as possible, but do not care much about the realization of the underlying solver. At this layer, a continuous equation with a corresponding computational domain and boundary conditions can be specified.
- ExaSlang 2 is one refinement step more concrete. It enables users to specify (or, if generated automatically, modify) discretized versions of the equations at layer 1. We consider this to be the main workplace for mathematicians experimenting with different approaches

to discretization, as well as interested engineers and scientists.
- ExaSlang 3 exposes the underlying multigrid algorithm by allowing its components to be specified. This layer is intended for mathematicians and computer scientists who implement and evaluate new smoothers or inter-grid operators with respect to convergence or performance.
- ExaSlang 4 is the most concrete layer of the ExaSlang hierarchy. It enables the full specification of a multigrid algorithm and exposes aspects of the parallelization and communication for modification by the user. We consider this layer appropriate for computer scientists interested in parallel efficiency and using the full feature set of language, domain, and code generator.

The ordinal sequence of the ExaSlang layers traces the process of solving a PDE: specification of the equation, discretization, algorithm selection and implementation, and finally parallelization and optimization. ExaStencils' vision is to make all selection processes eventually automatic, such that the user must only specify the problem at layer 1 and make some simple, menu-driven choices concerning the algorithm and execution platform. Currently, the algorithmic properties of many elliptic PDEs that involve the Laplace operator can be processed fully automatically, i.e., we are able to discretize the operator using finite differences and transform the computational domain, as well as user-specified variables and boundary conditions, to corresponding fields and functions. Subsequently, corresponding smoothers and a complete solver implementation are generated automatically. As our implementations of these automatic transformations improve, we will also work on the automatic selection of good or optimal feature and parameter sets [20], e.g., to determine the problem-specific and target-platform-specific optimal coarse-grid solver and smoothers, the multigrid hierarchy size, or the number of smoothing steps. Often, more experienced users might want to experiment with concepts not implemented or considered by our code generator's automatic feature selection process by working at the corresponding ExaSlang layer(s). An example might be the use of finite elements discretization, which currently requires the specification of discretized operators and equations at layer 2. From there, the remaining code can again be generated automatically.

In Project ExaStencils, we chose, for a number of reasons, to design our own external DSL rather than reusing an existing language. First, the majority of the existing DSLs for the solution of PDEs and focused on the application of stencils do not give full regard to multigrid, but make it cumbersome to work with its hierarchy of grid sizes. Frequently, it is hard to specify the grid hierarchy itself or to declare operators that map values between different grid sizes. A further reason is that, while ExaStencils focuses on the solution of PDEs, another main concern of the project is to explore how far automation can be driven in this domain. As such, we need a flexible vehicle to communicate the users' specifications to our code generator. With regard to the four abstractions layers of ExaSlang

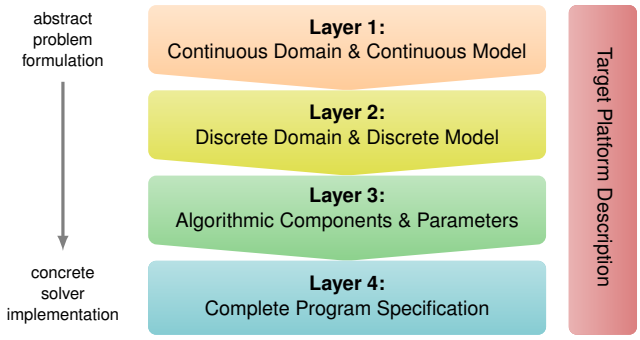


Fig. 2: Multi-layered approach of ExaSlang [54].

presented, among each other, they need to be consistent in their syntax and terms, and, at the same time, need to provide the corresponding abstractions and concepts that provide enable automation. Effectively, this makes it virtually impossible to reuse existing languages.

We implemented ExaSlang as an external DSL, since only this approach allowed us to tailor each layer optimally towards the designated user groups. Opting for an external DSL obviously has the drawback of higher efforts not only on the language implementation's side, but also on the users' side, who have to learn a completely new language. The former burden can be eased by a suitable toolkit, as will be seen shortly. The latter can be overcome by making the upper layers of the language increasingly comfortable for domain experts via the inclusion of abstractions from their domain. One might argue that re-using an external language results in an even lower entry barrier. Yet, in our experience, (novice) users often are confused by a mix of different layers of abstraction in a language or make assumptions—based on their knowledge of the host language—that do not hold for the embedded DSL. A typical example in many C-based stencil languages is the absence of pointer arithmetic or the modification of the memory layout.

At each ExaSlang layer, many automatic decisions may require a profound knowledge of the target system. One example is the automatic selection of the best smoother: Jacobi smoothers tend to make better use of GPUs, since every calculation is done independently, in contrast to the Gauss-Seidel method. Another example is the choice of SIMD extensions for vectorization, or cache sizes for blocking. The system is specified in a separate language, the *target platform description language (TPDL)*, which can be considered orthogonal to the four algorithmic language layers. It features a template mechanism with which available device descriptions can be combined to descriptions of large systems such as supercomputers [53].

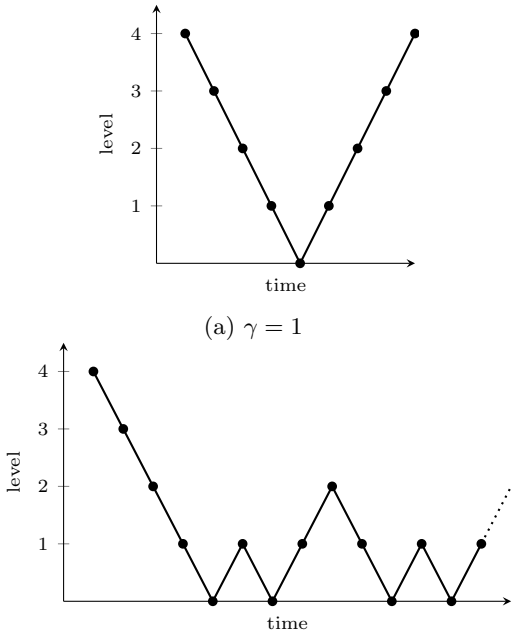
In what follows, we use and explain occasionally ExaSlang 4 code for illustration of examples. There is also more detailed literature on ExaSlang [54, 55].

C. Multigrid Algorithm

The multigrid method [24, 62] is based on two principles: (1) the *smoothing property*, that is, the fact that classical

if not coarsest level **then**
  $\bar{u}_h^{(k)} \leftarrow \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A_h, f_h\right)$ ▷ pre-smoothing
  $r_h \leftarrow f_h - A_h \bar{u}_h^{(k)}$ ▷ compute residual
  $r_H \leftarrow \mathcal{R} r_h$ ▷ restrict residual
  $e_H^{(0)} = 0$
  for $j = 1$ to γ **do** ▷ recursion
    $e_H^{(j)} \leftarrow \mathrm{MG}_H\left(e_H^{(j-1)}, A_H, r_H, \gamma, \nu_1, \nu_2\right)$
  end for
  $e_h \leftarrow \mathcal{P} e_H^{(\gamma)}$ ▷ interpolate error
  $\tilde{u}_h^{(k)} \leftarrow \bar{u}_h^{(k)} + e_h$ ▷ coarse grid correction
  $u_h^{(k+1)} \leftarrow \mathcal{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A_h, f_h\right)$ ▷ post-smoothing
else
  solve $A_h u_h = f_h$ exactly ▷ coarse-grid solution
end if

Fig. 3: Recursive V-cycle to compute an iterative solution $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A_h, f_h, \gamma, \nu_1, \nu_2\right)$.



(a) $\gamma = 1$

(b) $\gamma = 2$

Fig. 4: Schemes of the V- and W-cycle.

iterative methods such as Jacobi or Gauss-Seidel (GS) can smooth the error after very few steps, and (2) the *coarse-grid principle*, that is, the fact that a smooth function on a fine grid can be approximated satisfactorily on a grid with fewer discretization points. The multigrid method combines these two principles into a single iterative solver. The recursive algorithm alternates between fine and coarse grids in a hierarchy of grids.

The basic multigrid algorithm is depicted in Fig. 3. In the *pre-smoothing* step, high-frequency error components are damped first. Subsequently, a new error approximation, the *residual*, is calculated. Its low-frequency error components are then approximated on coarser grids (the *restriction* of the residual). Parameter γ governs the number of recursive calls: $\gamma = 1$ defines a *V-cycle*, whereas $\gamma = 2$ defines a *W-cycle*; see Fig. 4. Returning from the recursive call, the residual is projected (*prolongated*) back to the finer grids and eliminated there (*coarse-grid correction*). At the end, remaining high-frequency error components are smoothed again (*post-smoothing*).

Note our nomenclature: with *level*, we mean the level in the hierarchy of grids, i.e., a grid of a certain size. With *layer*, we refer to an instance in our hierarchy of programming languages, as explained in Section II-B.

III. Related Work

Numerous research efforts exist that focus on easing the modeling or implementation of science and engineering problems. *Programming abstractions* can be classified roughly into DSL-based or library-based. Of course, combinations of both also exist, e.g., embedded DSLs implemented via template metaprogramming. For DSLs and highly related to compiler research, a lively research topic is the efficient implementation of languages and supporting frameworks to aid language developers. Important to language users is not only the modeling of algorithms but also the tuning of implementations, either hand-written or generated. As computing platforms are highly complex and provide a wealth of options and parameters, many researchers investigate the automation of finding the best combination of all these factors.

A. Library-Based Approaches

Frequently, library-based approaches provide a familiar programming environment to developers and allow the reuse of existing code, but lack the automation of the selection of suitable algorithmic components and implementation optimization. In contrast to the approach of ExaStencils, a sophisticated transformation framework and source-to-source compilation are not necessary for their development and usage. Considering such approaches, one well-established and mature project is DUNE [9], a C++ template library that provides support for different types of grids and offers many linear algebra building blocks such as BLAS routines and corresponding data structures as well as various input and output data formats. *hypre* [16] is another well-known library for the solution of large, sparse linear systems of equations for Fortran, C, C++, Python, and Java. It focuses on multigrid solvers for both structured and unstructured grids. Its installation can be customized for the target platform, e.g., by exchanging underlying libraries and frameworks, such as MPI or BLAS libraries, or by modifying configurations for certain solvers.

FETK [60] is an ANSI C-based bundle of libraries supporting the implementation of finite element methods (FEMs). Among others, it provides modules for meshes and refinement, numerical computations (including specialized libraries to solve equations such as Poisson-Boltzmann or Smoluchowki), and visualization of data. In contrast, ExaStencils employs at present finite difference (FD) and finite volume (FV) discretizations. Chombo [14] is a library

for FD and FV methods on block-structured adaptively refined meshes that mixes two programming languages: it uses C++ for high-level abstractions and Fortran for calculations on regular patches. Like ExaSlang, it has a hierarchical architecture that offers complete and parallel implementations of applications, solver libraries, mesh operators, data operators, and finally utility functions such as data I/O. But it is a library, not a DSL.

Another well-known project is *PETSc* [7], a suite of data structures and routines for C, C++, Fortran, and Python.

B. Domain-Specific Languages

Coming to DSL-based approaches, PETSc forms the basis for Firedrake [48], an automated toolchain for solving PDEs specified in a domain-specific language embedded in Python. Internally, Firedrake employs the Unified Form Language (UFL) [3] and FEniCS Form Compiler (FFC) of the *FEniCS* project [40], which is another established collection of routines, data structures, and other components for the solution of differential equations, focused on the FEM. For mesh data structures and solver implementations, Firedrake relies on the PETSc as mentioned above, and on OP2 [44] for system assembly. OP2 and PyOP2 [49] target mesh-based simulation codes over unstructured meshes, generating code for MPI clusters, multi-core CPUs, and GPUs. Furthermore, the latter employs run-time compilation and scheduling and is also used by Firedrake for system assembly. In contrast, ExaSlang focuses on the use of FD and FV methods on regular grids.

STELLA [23] targets stencil codes on structured grids, with an OpenMP and a CUDA back-end currently under development. In contrast to ExaStencils, it uses an internal DSL embedded into C++ and is based on template metaprogramming. SDSLc [50] is an external DSL for stencil computations that also supports embedment of DSL blocks into C/C++ and MATLAB programs. Optimizations applied to SDSLc programs include (nested) tiling and improved register reuse in the case of high-order stencils. Compared to ExaSlang, there is no focus on multigrid methods and use of domain knowledge at the algorithmic layer. Hipacc [42] targets the domain of geometric multigrid applications on two-dimensional structured grids. It provides code generation for accelerators, such as GPUs (CUDA, OpenCL, RenderScript) and FPGAs (C or OpenCL code that is suited for high-level synthesis). Originating from image processing, there is no support for volume data or distributed-memory parallelism. Mint [63] is a programming model for GPUs and focuses on the source-to-source compilation of annotated C to CUDA. It does not provide special syntax or semantics for multigrid solvers or domain-specific transformations. SEJITS [12] is a Python-based approach that uses LLVM to optimize relevant parts of the code at run time. FLAME [22] derives loop-based algorithms from a custom notation for dense linear algebra while also checking the result for correctness. It focuses on the efficient implementation of basic algorithmic building blocks, instead of the holistic approach ExaStencils is pursuing.

FFTW [19] specializes in the Fast Fourier Transform and applies auto-tuning during installation to find the optimal implementation. Halide [47] focuses on image processing algorithms and generates code for a variety of platforms from a DSL embedded into C++. The same description can be transformed to Verilog code by Darkroom [26]. Halide is also commercially used by Google to improve pictures taken with Pixel 2 smartphones [35]. It is domain-agnostic but has been used in a number of case studies involving Laplace kernels on different hardware platforms. It is not a DSL for the domain of scientific computations and does not provide any domain-specific abstractions. Finally, SPIRAL [46] provides abstractions for linear transforms and other mathematical functions. In cooperation with ExaStencils, SIRAL has recently also been used to prototype a multigrid algorithm [10]. It is especially noteworthy as the main inspiration for ExaStencils with its hierarchy of domain-specific abstractions that are converted to highly efficient code by transformations and use of domain knowledge.

C. Auto-tuning

To improve the performance of hand-written or generated code, auto-tuning is a popular approach. It works by comparing the performance results of variants of code. These variants all provide the same functionality but differ in their implementation, e.g., in memory layouts or tiling factors. They might even use different technologies under the hood. Consequently, for many codes, this results in a search space so large that it is not feasible to check every possible combination. Thus, most auto-tuners provide heuristics to accelerate the process. In ExaStencils, it is our goal to provide the best prediction already at the code-generation phase, i.e., to have appropriate knowledge of hardware and performance models a priori.

A very well-known project is ATLAS [64] for data- and instruction-cache optimization that generates performance-portable implementations of BLAS and LAPACK linear algebra routines. Auto-tuning can also be used to compose a program from a number of user-supplied building-block variants. This approach is pursued, for example, by PetaBricks [5]—which has also been used to optimize multigrid implementations [13]—and Zettabricks [4]. Both are aiming at finding the most efficient composition to solve equations, e.g., PDEs. Related to Zettabricks, but without the particular focus on scientific computing, is OpenTuner [6], a generalized framework for building domain-specific multi-objective program auto-tuners.

D. Language Transformation and Compilation

On the side of code generation and transformation approaches, a highly popular project is LLVM [36], which essentially provides a compiler infrastructure. By defining an intermediate representation (IR) that can be compiled to a runnable binary on many popular hardware architectures, it can be used by different front-ends, of which Clang for the compilation of C/C++ indeed is the most well-known. Clang is a very popular target compiler for ExaStencils,

i.e., works on input generated from ExaSlang. Athariac's does also ship with an IR, but it is much more abstract compared to the low-level, single-assignment LLVM IR. ROSE [39] is a similar infrastructure custom-designed to build source-to-source program transformation and analysis tools for C/C++, Fortran, Java, and others. Kiama [58] is a lightweight language processing library for Scala centering around strategy-based rewriting, which in turn is based on the Stratego/XT language and library, now part of the Spoofax Language Workbench [27]. However, it does not provide its own IR and corresponding transformations. Another Scala-based approach is Lightweight Modular Staging (LMS) [52], a dynamic code generation library. It is used by Delite [11], a framework for parallel DSLs, and enables developers to employ abstractions at will, which are then, in the staging phase, automatically removed to retain performance. Compared to Athariac, LMS focuses on the optimization of DSLs embedded into Scala. Delite has supports for task-based parallelism, and consequently needs its own run-time system for execution of generation programs. Athariac, in turn, allows the user to implement custom parallelism schemes that do not need any dependencies for at run-time.

AnyDSL [37] is a framework for the implementation of DSLs. A core feature is the partial evaluation of abstractions to improve the performance of the generated code. It emits code for a number of back-ends, notably LLVM and GPUs via CUDA and OpenCL. Similar to Delite and in contrast to Athariac, programs rely on a run-time library.

IV. Code Transformation

To refine user programs of some ExaSlang layer at the next more concrete layer, and eventually emit C++ code, we have developed a flexible framework for external DSLs, called *Athariac*, that facilitates the easy specification of code transformations. Its name is assimilated from the Gaelic term *atharrachadh*, which can be translated to "the act of altering, changing, modifying, or varying." The framework builds on the concept of stepwise term rewriting, i.e., transformations work by replacing subterms of one formula with other terms. Consequently, a program is modified to its final form by a large number of small transformations that are applied sequentially. Based on an earlier evaluation [56], we have implemented it in Scala [45] and we make extensive use of advanced Scala features, such as parser combinators and pattern matching, as we explain in the following subsections. The framework has been used to implement the ExaSlang compiler but, in its core, is highly agnostic of the DSL's domain. Transformations and optimizations can be divided into two categories: ExaSlang-specific ones that are, e.g., crucial to our approach to parallelization and communication, and others that can be applied to a wide range of languages, such as constant propagation and constant folding. In the rest of this section, we describe the features of Athariac. We use the term *framework* as a synonym for Athariac, whereas *compiler* refers to the ExaSlang compiler, which has been implemented with Athariac.

A. Parsing

The first step in the refinement of a domain-specific program is the parsing of the source text. Scala provides a number of different parser implementations that can be combined via parser combinators. A parser combinator is a higher-order function that applies transformation functions to generate a parser from simpler parsers. Its output is a parse tree, called the *concrete syntax tree* (CST), which is composed of specialized data structures that usually must be implemented by the DSL developer. Next, this CST is transformed to an *abstract syntax tree* (AST) by collecting and adding bits of information, such as data types of variable accesses and function calls, or the resolution of language-specific syntactic sugar. In ExaSlang, one example of syntactic sugar is level specifications. An explanation and details of their refinement by our compiler are explained in Section V-A.

B. Data Structures

For the implementation of custom node structures of the program tree, Athariac offers the base type `Node`. It is not obligatory to inherit from it, but transformations will only be applied to subelements if the element does inherit. Additionally, inheriting from `Node` allows objects to be annotated, which comes in handy to mark parts of the tree or attach special information to certain nodes for use in subsequent transformations. The data structures offered by Athariac make up what we call the *intermediate representation* (IR); their names are prefixed with `IR_`.

Additionally to inheriting from `Node`, it is highly recommended to implement data structures in the form of Scala `case` classes, for which the Scala compiler emits pattern matching functions automatically (see Section IV-C), among others. This saves a lot of boilerplate code that would need to be written otherwise. Case classes have another advantage: a companion object of the same name is created automatically, acting like a factory that allows omitting the `new` keyword for object creation. This might seem a minor detail but it improves code clarity when nesting object instantiation, as depicted in Fig. 5.

For case classes, the `equals` function—for which the equality operator == is an alias—will compare objects, not by their reference, but structurally, which is actually used quite frequently, e.g., when comparing function calls, consisting of a function name, a number of typed arguments, and an expected return type. Thus, the automatic

```
1 var x = FunctionCall("sin",
2   BinaryExp("+", BinaryExp("*", IntConst(2),
3   DoubleConst(3.14)), IntConst(1))
```

Fig. 5: Clearer generation of nested objects via companion objects of case classes.

generation of such functions saves a lot of boilerplate code. However, case classes also have one drawback: careful class and inheritance design may be required because there is no inheritance from case classes.

For the implementation of ExaSlang, we rely on Athariac's data structures in combination with custom ones tailored especially for the domain, such as the representation of matrix and vector data types, specialized loop constructs, and communication-specific data structures. In turn, many of the custom data structures make use of Athariac's structures. For many of the custom data structures, companion objects have been defined to ease usage of some of the more complex types, or to provide appropriate conversion of instantiation arguments.

C. Transformations

Transformations make extensive use of Scala's rich pattern matching capability to follow Athariac's term-rewriting approach. (*Strategic*) *term rewriting* is a technique and corresponding formalism for performing analysis and modifications of tree-structured data [8]. One particularly powerful feature intensively used by Athariac is the so-called *deep matching* that facilitates the match of an entire subtree pattern at once. In Fig. 7, we defined a pattern to check whether `exp` matches a binary operation with the double constant of 3.14 as the right operand, regardless of the left operand or operator, as indicated by the wildcard symbol underscore. If the expression matches, it is assigned to variable `myexp` and output. A `match` statement can specify multiple patterns, as well as default cases, as shown in line 5 in Fig. 7.

Our transformations contain one or more patterns by giving their input and output type `Node`. A single node can be transformed to multiple nodes (aggregated in a list), e.g., when unrolling a for-loop. To remove a node from the tree, transformations may either return the empty list or Scala's special object `None`. An element can be modified or replaced with a newly instantiated one. Transformations may contain a default case but need not; in the latter case, the transformation is simply not applied to the node.

Transformations are reapplied recursively, i.e., even after a match, the matching node's subnodes are taken into account. The recursion can be disabled by setting a flag; however, the transformation will still be applied to other nodes, just not to the subnodes of the matching object. One use case is a transformation that should only touch the outer-loop of a loop nest.

Transformations can be restricted to a subtree by specifying the subtree's root node. This saves the effort of traversing potentially unwanted parts of the AST, and allows for the transformation of specific parts of the code while leaving others untouched.

```
1 case class BinaryExp(operator : String,
2   left: Expression, right : Expression)
3   extends Expression
4 case class FunctionCall(name : String,
5   arguments : List[Expression])
6   extends Expression
```

Fig. 6: Definition of Scala case classes.

```
1 exp match {
2   case myexp @ BinaryExp(_, _,
3     DoubleConst(3.14))
4     => println("Found Pi: " + myexp)
5   case _ => println("Did not match")
6 }
```

Fig. 7: Pattern using variable binding, deep matching, and a default case.

D. Strategies

Frequently, transformations must be processed in a fixed order, for instance, preliminary transformations that check for errors in the AST, add information to nodes, unify different types of nodes, or do similar work before subsequent transformations can be applied. Thus, it is desirable to group transformations in what we call *Strategies*. For the implementation of simple strategies that just apply a series of transformations consecutively, Athariac offers the base class `DefaultStrategy`. It acts as a container that guides, logs and prints transformation execution progress and some statistics about the number of matches. Sometimes, the sequential execution of a series of transformations is not sufficient, for example, when a certain transformation needs to be applied conditionally or repeatedly. In such cases, custom strategies may be implemented by inheriting from the default strategy class. Under the hood, strategies interact with Athariac's StateManager, which we explain in the next subsection. In our ExaStencils compiler, the *simplification strategy* is an implementation of a custom strategy. Among other purposes, it is used for constant folding, i.e., the compile-time evaluation of (constant) expressions. For example, an expression such as $3 * (1 + 2)$ is evaluated in two steps. Consequently, the strategy applies its transformations not only once, but as long as modifications take place.

E. StateManager

The *StateManager* controls the transformation process by providing the interface strategies used to apply transformations. When a transformation is to be applied to a node, the StateManager first checks its type. If it inherits from `Node`, the transformation can be applied directly. For any type of collection, such as lists, maps, or arrays, the transformation is applied to each element if it is of a compatible type (i.e., a node or collection). Notably, the StateManager will also iterate over all user-defined fields of the given node.

A frequent case is to check for the presence of, or work on, a list of certain nodes, regardless of their order or position in the AST. To this end, the StateManager offers functions such as `find` and `findAll` that allow the specification of

```
class VariableCollector extends Collector {
  private var ctx : List[HashMap[String, Datatype]] = Nil

  override def enter(node : Node) : Unit = node match {
    case _ : ForLoop => ctx ::= HashMap[String, L4_Datatype]()
    case x : VarDecl => if(!ctx.isEmpty) ctx.head += ((x.identifier.name, x.datatype))
    case _           => // else: do nothing
  }

  override def leave(node : Node) : Unit = node match {
    case _ : ForLoop => ctx = ctx.tail
    case _           => // else: do nothing
  }

  override def reset() : Unit = values.clear()

  def get(name : String) : Option[Datatype] = {
    for (it <- ctx) {
      val dt = it.get(name)
      if (dt.isDefined) return dt
    }
    return None
  }
}
```

Fig. 8: Definition of a context-building collector that keeps track of variable declarations in for-loops.

a predicate (i.e., a pattern) and return the first, respectively, all matching nodes of the entire AST or some subtree.

For other features of the StateManager, such as transactions and checkpoints, we refer the interested reader to previous work [54].

F. Collectors

Collectors implement the observer design pattern for AST traversal. Figure 8 shows a simple collector to aggregate variable declarations inside for-loops, i.e., to build what in compilers generally is called a context. Here, we create and remove context representations whenever the `enter` or `leave` function is called on a for-loop. When encountering a variable declaration, we extract identifier and data type and save them in the current context. However, most transformations do not need this kind of profound knowledge, so we can save the overhead of notifying observers for each traversal by allowing on-demand registration and removal of collectors.

G. Annotations

To attach extra knowledge to the AST, nodes can be annotated. Annotations consist of an identifier and an optional value, and can be set, modified, or removed anytime inside or outside a transformation. There is no limit to how many annotations can be added to a node.

A frequent use case is the specification of a node's origin, i.e., the line number and filename of the user-specified program that led to the instantiation of this node by the parser. Another example is presented in Fig. 10, in which annotations are used to generate unique names for temporary variables. Here, expressions are annotated with the matrix element's position they will yield so that the expressions can be substituted accordingly in subsequent transformations.

V. Transformation Workflow for ExaSlang

To demonstrate the specification and application of transformations in our framework, let us exemplify the refinement of vector-valued to scalar operations for the generated C++ code. We only have space to illustrate this at the most concrete ExaSlang layer. In ExaSlang 4, one may define and use higher-dimensional data types, such as vectors and matrices, for variables, constants, fields, and stencils. This feature is used in many application, e.g., in the computation of the optical flow [55] presented in Section VII.

A. Level Specifications

The very first step in the ExaStencils processing chain starting at layer 4 is a parse of the source file that yields a tree with a number of *accesses*, i.e., usages of identifiers that represent variables, stencils, or fields. Their types have to be inferred, so subsequent transformations have the information necessary. Additionally, accesses may carry level specifications that must be resolved appropriately, as explained in the next paragraph.

In ExaSlang, a *level specification* is one way of specializing the language for the targeted domain—here, multigrid—and allow the definition of grid-level-specific functionality or definitions. This mechanism also enables the specialization of certain aspects on a per-level basis. In the syntax, level specifications are suffixes to definitions and accesses and begin with the symbol `@`, followed, for example, by keywords (such as `all`, `current`, `coarser`, `finer`, …), a single integer referring to one specific grid size, a range of levels,

```
1  Func VCycle@(all but coarsest) () : Unit {
2    Smoother@current ()
3    CalculateResidual@current ()
4    Restriction@current ()
5    SetSolution@coarser (0)
6    VCycle@coarser ()
7    Correction@current ()
8    Smoother@current ()
9  }
10
11 Func VCycle@coarsest () : Unit {
12   /* ... solve directly ... */
13 }
```

Fig. 9: ExaSlang 4 implementation of the multigrid V-cycle using level specifications to exit recursion.

a list of levels, or any combination thereof. A frequent use case is the implementation of the multigrid method itself, as shown in Fig. 9, which implements the algorithm in Fig. 3 for $\gamma = \nu_1 = \nu_2 = 1$. Here, the first function, starting at line 1, works on all grid sizes other than the smallest (i.e., coarsest). A function is free to reference elements at other levels as well as, for example, in the restriction of the residual. In line 6, the `VCycle` function for the next more concrete multigrid is called. If this happens to be the most concrete multigrid level, the function defined at line 11 is called, which terminates the recursion and invokes the coarse-grid solution.

In an ExaSlang 4 program, one object reference using `@current` might refer to varying object declarations, depending on the multigrid level. Since our goal is to optimize the code as best as possible, we need to convert these compact statements to independent functions, e.g., in order to take grid sizes and, thus, loop bounds into account. This requires a chain of preparation and execution steps:

- We start with a function definition for all levels. Inside its body, we specify a range of levels and iterate from the next-to-coarsest to the finest level in this range. The range excludes the finest level of all, level 4. The lower and upper bounds of the range will be resolved by our code generator in the following step.

```
Func foo@all () : Unit {
  Var x@(coarsest + 1 to finest, not(4))
    : Int = 0
}
```

- In the first refinement step, the constants `@coarsest` and `@finest` are replaced by values specified by the user in a separate file or determined automatically by other means such as a convergence analysis, while `@all` is replaced by the corresponding constant range.
- Expressions can now be evaluated.
- Next, exclusions from ranges can be resolved by simply removing the according items from the list.
- At this point, the only level specifications left will be constant values, or lists thereof, and the `@current` specification. To resolve these, functions and declarations such as (global) variables, fields, stencils, etc. are duplicated for each level specification.
- Finally, remaining `@current` specifiers, which—by definition—can only be used in scopes that carry an absolute level specification, are resolved accordingly.

Duplicating entities for individual multigrid levels, and optimizing each level separately, obviously has the drawback of code explosion and, consequently, increased workload for subsequent transformations and the production compiler. On the other hand, it permits more effective optimizations since level-specific knowledge can be taken into account. For duplication, there are two different approaches: the developer may either implement and call his/her own copying functions, or use the framework-supplied cloning methods via the `Duplicate` object, which works with reflection. Scala's compiler also generates default `copy` functions for case classes, but code frequently works on references to abstract base classes or traits, for which such functionality is not available, which calls for an implementation by the DSL developer. In the future, Athariac will provide a cleverer solution to this problem than code duplication.

B. Conversion to Scalar Data Types

After having resolved the level specifications, we can now concentrate on the specifics of the higher-dimensional data types. Here, the main transformation challenges are functions and in-line matrix expressions. Since user-specified functions may have side-effects, such as a modification of global variables per function call, we must maintain the exact number of function calls as specified by the user. Thus, our next step is to extract the function calls from any expression tree before unfolding it to scalars, so a function is still called only once, and not as part of every scalar operation. We ensure this by replacing every function call returning a matrix by an access of a temporary variable holding the call's result. Furthermore, this has to be done recursively, since one function call might appear as an argument of another. The actual implementation is divided into a number of transformations. Preparatory transformations look at expression trees and copy any function call to a temporary variable before the expression is used, and annotate the original function call with an ID that is used to construct the unique variable name. Next, the actual replacement transformation, shown in Fig. 10, replaces the function call with the access. Since the workflow of extracting in-line anonymous matrix expressions is quite similar, we usually implement the resolution of both with the same transformations.

Since we would like to flatten the higher-dimensional data types to scalars, functions can no longer return a matrix but are modified to write their result to the temporary variable that is now passed by reference to the function. Accordingly, the function call is modified. For the function itself, all `return` statements specifying a return value are replaced by appropriate assignments and return statements without value.

Finally, variable, field, and matrix accesses in higher-dimensional expressions are dissolved into scalar operations

```
this += Transformation("Replace by variable accesses", {
  case call : IR_FunctionCall if (call.hasAnnotation("ResolveMat_counter")) => {
    IR_VariableAccess("_fct" + call.popAnnotation("ResolveMat_counter"), call.datatype)
  case exp : IR_MatrixExpression if(exp.hasAnnotation("ResolveMat_counter")) => {
    IR_VariableAccess("_mat" + call.popAnnotation("ResolveMat_counter"), exp.innerDatatype)
}})
```

Fig. 10: Definition of a transformation that replaces function calls and in-line matrix expression with variable accesses by checking the objects' annotations.

by duplicating them, annotating the target element's position, i.e., row and column, and then reducing them to scalars in a subsequent transformation.

VI. Target-Specific Optimizations

Another essential part of our code generator is the optimization of the generated code [31]. The supported optimization strategies vary both in complexity and potential benefit. The classic transformations, such as loop unrolling or address pre-calculation [2], are also part of some production compilers. But, since we explicitly support several different architectures and systems, we cannot focus on a single compiler, which justifies implementing these classic optimizations directly in our code generator.

The more advanced techniques contain, e.g., polyhedral optimizations and different redundancy eliminations [30]. The latter focus on reducing the computational effort, while the former can be used to apply a temporal blocking to increase the performance of memory-bandwidth bound codes. Additionally, the results of an exact polyhedral data dependence analysis are input to several other strategies, e.g., vectorization.

As an example of how these optimizations can be implemented in and benefit from the features of our framework, let us take a closer look at a specialized vectorization strategy. It consists of a sequence of four steps:

1) the detection of loops with vectorization potential,
2) the generation of vectorized versions of the loops,
3) the elimination of redundant load operations,
4) the selection of vector instructions suitable for the target architecture.

Steps 2 and 3 should not be merged since other strategies may potentially apply between them, such as loop unrolling. There exist several more advanced vectorization strategies [29, 50, 51], which could be implemented, too. However, their effectiveness depends on the structure of the loops to be vectorized. Another option would be to implement multiple techniques and select the best fit for each loop individually.

To provide insight into how the vectorization works and how it makes use of the features of Athariac, let us take a closer look at the following example:

```
for (int x = start; x < end; x++)
  out[x] = 1.2 * in[x] - 0.2 * in[x + 1];
```

A. Detection of Loops to be Vectorized

The first step, the detection of loops to be vectorized, uses deep pattern matching to select only parallel innermost loops with an appropriate loop header. Figure 11 shows a small excerpt of what this pattern matching looks like. Line 1 contains the match for the loop node. The first field of an `IR_ForLoop` stores the initialization part of a classic C-like loop. Only loops with an integer variable declared and initialized (the last field of the declaration must be `Some(..)`, not `None`) in the loop header are being matched here. Then, the upper bound is extracted from the exit condition if it is either a lower (line 4) or a lower-equal (line 7) expression and its left-hand side is an access to the variable declared in the initialization part (lines 5 and 8).

The loops detected in this process are not certain to be vectorizable, but the loops passed over are certain not to be. This means that, if any construct occurs that cannot be handled, one must be able to cancel the transformation for this loop and restore the original version. This is achieved by working on a copy of the original syntax tree until the vectorization can be completed. Note that ExaSlang does not support aliasing; thus, no special checks are required in this case.

B. Loop Vectorization

The second step performs the actual modifications of the code. Even though the vectorization process is, in general, architecture-independent, the vector sizes must be known and the decision of whether to allow unaligned memory accesses must be taken at the beginning. The first store operation in the example code accesses `out[x]`. The framework ensures that the array itself is aligned properly, which means that the first value of the loop iterator must also be evenly divisible by the vector size if an aligned store is required. This can be achieved by splitting the loop and creating a prologue. Additionally, if the number of loop iterations is not evenly divisible by the vector size, a second split creates an epilogue.

Next, the loop stride is updated and the expressions in the loop body are replaced by their vectorized counterparts. Variable and array accesses are replaced by new temporaries, which are declared directly before their first usage. This reduces the number of load instructions required and allows the reuse of values already present in a register.

If a function call is found, the vectorization of this loop is canceled, unless a call to a function from the math

```
case IR_ForLoop(IR_VariableDeclaration(IR_IntegerDatatype, itName, Some(init)),
                condExpr, incrExpr, body, /* ... */) =>
  val upperBoundExcl : IR_Expression = condExpr match {
    case IR_Lower(IR_VariableAccess(bName, IR_IntegerDatatype), upBndExcl)
      if itName == bName =>
        upBndExcl
    case IR_LowerEqual(IR_VariableAccess(bName, IR_IntegerDatatype), upBndIncl)
      if itName == bName =>
        IR_Addition(upBndIncl, IR_IntegerConstant(1))
    case _ => /* abort vectorization for this loop */
  }
```

Fig. 11: Excerpt from the loop header testing for the vectorization strategy.

```
// prologue [...]
simd_vec vec0 = simd_set1(2.0);
simd_vec vec2 = simd_set1(1.2);
for (int x = vecStart; x < vecEnd; x += 4) {
  simd_vec vec1 = simd_load_aligned(&in[x]);
  simd_vec vec4 =
    simd_load_aligned(&in[x + 4]);
  simd_vec vec3 =
    simd_conc_shift(vec1, vec4, 1);
  simd_vec vec5 =
    simd_madd(vec0, vec1,
      simd_mul(vec2, vec3));
  simd_store_aligned(&out[x], vec5);
}
// epilogue [...]
```

Fig. 12: Vectorized loop using intermediate vector instructions.

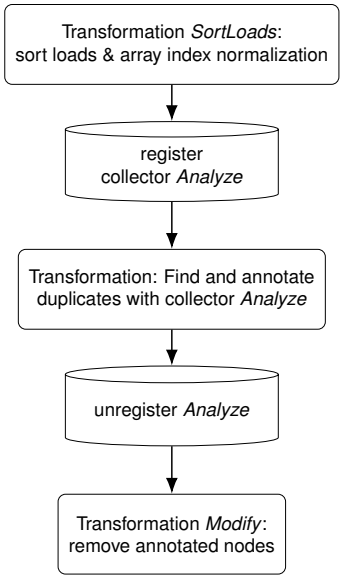


Fig. 13: Basic structure of the custom strategy to remove duplicate vector loads.

library is found for which a vectorized version is externally available. Figure 12 presents a vectorized version of the example code using architecture-independent intermediate instructions. The special instruction `simd_conc_shift` is necessary since, in this example, vectors starting at both `in[x]` and `in[x + 1]` are needed, which can obviously not be aligned both at the same time. Therefore, this instruction consumes two vectors, concatenates them, and returns a new vector that contains the elements starting at the position given by the third argument. For example, `simd_conc_shift(vec1, vec4, 1)` returns a vector that contains all but the first element from `vec1` in positions 1 to 3, while its last element is the first of `vec4`. The resulting loop is also annotated with information about the epilogue loop. In case of a subsequent unrolling, this information suppresses the generation of an additional epilogue.

C. Redundancy Elimination

The third step aims at removing redundant load operations inside a single and between subsequent loop iterations. This applies to variables `vec1` and `vec4` in Fig. 12: `vec4` in iteration `x` is equivalent to `vec1` in iteration `x+4`. Removing such a redundancy is straight-forward, but the detection might require some kind of prior normalization of the index expressions. To this end, we have implemented a separate, advanced normalization technique that can also be applied to a small subtree as part of the vectorization or any other optimization [30]. Other redundancies introduced by unrolling are also removed.

The basic structure of this elimination strategy is shown in Fig. 13. The first transformation uses the partial function `SortLoads` to permute the load instructions in a way that eases the elimination of redundancies. This means that, if a vector loaded in the previous loop iteration should be reused, the variable holding it must not be overridden before its value can be stored in another temporary. The actual redundancy detection is performed by the collector `Analyze`. It does not modify the syntax tree, but adds annotations specifying necessary changes. The collector is derived from Athariac's `StackCollector` class, whose purpose is to maintain a stack of ancestor nodes. This provides an easy access to the statement and the surrounding loop of a load expression. The last transformation `Modify` executed in this strategy finally incorporates all changes prepared by the previous analysis.

D. Selection of Target-Platform Instructions

The final step of the vectorization replaces the intermediate instructions by a suitable set of target-

specific ones. For example, on newer Intel processors with an FMA3 extension, `simd_madd(a, b, c)` is replaced by `_mm256_fmadd_pd(a, b, c)`. For processors without FMA3 extension, the combination `_mm256_add_pd(_mm256_mul_pd(a, b), c)` applies. The intermediate instruction `simd_conc_shift` requires a more careful selection, since AVX does not have a corresponding machine instruction. Depending on its third argument, a combination of shuffle, permute, and blend instructions is generated. `simd_conc_shift(a, b, 1)`, for example, results in `_mm256_shuffle_pd(a, _mm256_permute2f128_pd(a, b, 0x21), 0x5)`. In contrast to Intel processors, IBM's BlueGene/Q has a corresponding instruction that can be used directly. As a huge benefit of the architecture-independent intermediate instructions, code can be generated for the vector units of all x86-based processors, including the Intel Xeon Phi, as well as for the IBM BlueGene/Q and ARM architectures featuring the Neon extension. Additionally, adding support for a new vector unit is straight-forward.

VII. Evaluation

In previous work, we have already demonstrated that we are able to generate code for a variety of applications and target platforms. We have demonstrated scalability and runtime results for up to the full JUQUEEN machine of 458,752 cores across 28,672 nodes for Poisson's equation, employing several parallelization and communication strategies [54, 33]. We have demonstrated the feasibility of implementing imaging algorithms for execution on workstations [55], and we have conducted case studies of code generation for more exotic platforms such as embedded devices featuring ARM cores [34] and even FPGAs [57]. Here, we focus on productivity gains achieved with the DSL ExaSlang and its code generator by exploiting the large difference in software complexity of source and target code. To this end, we have selected a number of applications on which we have previously reported in the literature:

- The Poisson equation with constant factors on a two-dimensional unit square [54], parallelized with OpenMP. This application uses seven different sizes of grids with up to 4,225 unknowns. A number of V(3,3)-cycles with Jacobi smoothers are executed until the convergence criterion is met.
- The same application with variable coefficients on the three-dimensional unit cube [34]. Parallelization is done via MPI and OpenMP for a multigrid hierarchy of four levels with up to 35,937 unknowns. Again, V(3,3)-cycles with Jacobi smoothers are executed until the convergence criterion is met.
- Optical Flow in 3D [55], shared-memory parallelized with OpenMP. It uses seven multigrid levels for a total of $2,1 \cdot 10^6$ unknowns that are solved by executing five V(3,3)-cycle with Jacobi-style smoothers.
- Simulation of non-Newtonian fluids [32] using five multigrid levels (35,937 unknowns), parallelized with OpenMP. V(3,3)-cycles with red-blacks Gauss-Seidel smoothers are executed for each of the five fields with 35,937 unknowns until the convergence criterion is met.

For each example, we switch automatic vectorization on and off.

To quantify productivity gains for users, we rely on Halstead's complexity measures [25], which quantify software based on the number of operators and operands. Halstead's metrics are an approach to an objective basis for comparing different codes, independently of programming styles of individuals or enforced by the programming language. Therefore, they are much more significant than the popular lines of code (LoC). Basically, Halstead works with the total and the unique number operators (N_1 and η_1) and operands (N_2 and η_2) in a program. With these four numbers, different metrics can be formed. In Table I, we use three of Halstead's metrics:

- Volume: $V = (N_1 + N_2) \cdot \log_2(\eta_1 + \eta_2)$
  The "size" of the program.
- Difficulty: $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$
  The difficulty of writing or understanding the program.
- Effort: $E = V \cdot D$
  The effort of writing or understanding the program. This can be translated to a time span by division with a constant. Halstead proposed $T = \frac{E}{18}$ seconds. However, this is a controversial choice, so we disregard it here.

In fact, many more metrics can be calculated, such as the time required for implementation, or the number of bugs delivered. To get an impression of code complexity that feels more "natural" to programmers, we also included the LoC, as well as the number of defined methods and corresponding calls, in Table I. Because ExaSlang's source-to-source compiler emits C++ code, the Halstead numbers listed are comparable for any supported target platform.

Note that code generation times (T_G) are in the range of seconds for every test case, making the generation of different code variants feasible, e.g., for a search-space exploration to find the best-performing code for a given combination of application and platform. Additionally, compilation times using a single thread ($T_{C,1}$) and four, respectively, eight parallel threads ($T_{C,4}$, $T_{C,8}$) were measured, and we see that code-generation run times are, at most, of the same order of magnitude. The speedup factor of 4 for compilation times can be attributed to the test system's four physical cores, which Power8 SMT cores. Another observation is that the activation of vectorization increases code sizes (LoC) and numbers of method calls dramatically. Difficulty drops, which is easily explained when looking at its definition: it is the product of the number of distinct operators multiplied by the average occurrence of all operands. As vectorization introduces numerous short-lived operands, the average drops, leading to the counter-intuitive observation that vectorized programs are less difficult than non-vectorized ones. The effort is defined as the product of difficulty and volume. Thus, an increase in volume, which is a side-effect of vectorization, is not always able to repeal this effect. Another effect of vectorization is the huge

TABLE I: Halstead complexity measures, number of files, lines of code (LoC), method and method calls, and generation and compile times of ExaSlang 4 DSL programs and compared to generated target code.

	Poisson		**Var. Coeff.**		**Opt. Flow**		**Fluids**	
ExaSlang 4								
Files		1		1		1		1
LoC		240		296		351		902
Methods		12		15		17		15
Calls		66		101		57		288
Volume [$\cdot 10^5$]		0.10		0.23		0.20		0.81
Difficulty		111		178		216		159
Effort [$\cdot 10^7$]		0.11		0.41		0.44		1.29
Target code								
Vectorized	no	yes	no	yes	no	yes	no	yes
Files	60	60	70	70	96	96	224	224
LoC	2,475	3,362	12,916	15,490	5,225	9,032	9,623	15,154
Methods	59	59	69	69	95	95	223	223
Calls	215	964	1,241	3,246	124	6,190	591	3,173
Volume [$\cdot 10^5$]	2.4	3.1	16.4	18.8	16.4	24.6	20.9	29.8
Difficulty	203	180	323	280	562	418	403	326
Effort [$\cdot 10^7$]	5.0	5.6	53.0	52.5	91.9	10.3	84.4	97.0
Times								
T_G [s]	4.3	5.0	13.3	13.4	11.4	12.5	22.3	24.2
$T_{C,1}$ [s]	14.9	26.2	49.9	60.7	101.9	120.3	60.0	104.5
$T_{C,4}$ [s]	4.4	7.3	14.5	17.2	27.7	32.3	18.1	29.3
$T_{C,8}$ [s]	3.6	5.8	11.5	13.5	23.2	26.7	14.4	23.5
MFLOPS	708.4	764.3	1,935.0	2,073.8	3,098.2	4,507.7	2,003.0	2,085.2
Bandwidth [MB/s]	598.5	728.4	199.8	163.3	2,263.1	6,553.5	720.6	890.2
Solved Unknowns [1/s]	$3.52 \cdot 10^6$	$3.84 \cdot 10^6$	$1.00 \cdot 10^6$	$1.03 \cdot 10^6$	$0.32 \cdot 10^6$	$0.93 \cdot 10^6$	3,748.9	3,631.9

Target code file count and lines of code do not include auto-generated header files, support libraries (e.g., image handling for the optical flow test case), and other auxiliary files such as Makefiles. Code generation (T_G) and compilation times were measured for sequential ($T_{C,1}$) and parallel (4 threads, $T_{C,4}$ and 8 threads, $T_{C,8}$) on an Intel i7-6700 CPU using Scala 2.12 and Java 1.8 for the framework, and GCC 7.2.0 as back-end compiler. To illustrate the impact of vectorization, MFLOPS and bandwidth were measured on one core of an Intel Xeon E5-2630 v2 CPU, using GCC 4.8.5 as back-end compiler.

increase in the number of (unique) operators, because vector intrinsics appear as method calls to the parser. Since the parser works at a purely syntactical level, it is unable to consolidate a scalar operation (such as an addition using the + operator) and a corresponding vector intrinsic (e.g., `simd_madd`). MFLOPS and memory bandwidth for the generated solvers were measured using the likwid tool suite [61]. To highlight vectorization impacts, OpenMP and MPI parallelizations were disabled for these measurements. As a measurement of memory bandwidths was not available on the Intel Skylake platform, the benchmarks for MFLOPS and bandwidth were run on one core of an Intel Xeon E5-2630 v2. For comparison with results of established benchmarks, HPGMG [1] achieved 1134.1 MFLOPS and a memory bandwidth of 40.1 MB/s on the same machine, solving 70,456.5 unknowns per second. The problem that HPGMG computes is equivalent to the ExaSlang Variable Coefficients test case. These measurements were restricted to the pure benchmark runs, i.e., after the warm-up phase, using a configuration similar to our test cases: A V-cycle structure with Jacobi smoothers and solution on the coarsest grid using the CG method. Again, these numbers were measured using likwid. In a future publication, we will show and discuss detailed performance results. Here, we merely intend to make the point that our approach works and yields usable target code with reasonable performance. Additionally, it is a further illustration of the feasibility of code generation: compared to ExaSlang 4 code (see Table I), HPGMG's source code[1] is more complex and lower-level, making it hard to implement or improve for unexperienced users.

VIII. Conclusions and Future Work

Building on the techniques presented, among others, we were able to generate code for a large spread of different problems, platforms, and variants from the very same ExaSlang 4 code. However, there is room for many further exciting features to be realized with the ExaStencils approach. For our domain-specific compiler, we are looking at extending the application range to support also the solution of linear elasticity problems. Another goal worth pursuing would be the implementation of other communication libraries, such as GPI [21] or libOMPSS [17] as an alternative to the MPI+OpenMP platform. For accelerators, we already support NVIDIA graphics cards by emitting CUDA code. Here, OpenCL might prove a viable alternative such that not only AMD GPUs can be addressed, but also Altera FPGAs that are programmed using a vendor-specific OpenCL dialect [15].

At the user front-end, we are already exploring means to lower the entry barrier to our language and compiler

[1] https://bitbucket.org/hpgmg/hpgmg

for novice users. One approach is to provide a web-based graphical user interface (GUI) similar to IPython[2] for Python scripts, JSFiddle[3] for HTML and JavaScript experiments, or Overleaf[4] for typesetting documents with LaTeX that invite users to interact with the language and develop directly in a web browser, without the need for any modification of the local system. For our project, this would mean that the user would get the resulting C++ code from a remote ExaStencils compiler as a simple download via mouse click, or even the opportunity to execute the code in the browser's context. Going further in this direction, we could explore the possibilities of enabling more natural input in ExaSlang 1 such as hand-written mathematical expressions, written on a mobile device or even on a white board and submitted to the web front-end via a mobile app.

For the transformation framework Athariac, we plan to release a stand-alone version such that other DSL developers can use it for implementation of their languages or in education. At the same time, it might be worth looking at the possibility of providing a pretty-printer to emit code for a number of different platforms as part of the framework. This will require a generalization of some of the data structures in our ExaStencils compiler, and modification of existing transformations and optimizations that will be shipped as part of the framework.

IX. Acknowledgements

This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 "Software for Exascale Computing" in Project ExaStencils under contract numbers LE 912/15 and TE 163/17. We are grateful to all ExaStencils members for reviewing this manuscript and providing valuable feedback. Finally, our thanks go to Daniel Vahle for providing the software complexity analysis framework referred to in Section VII.

References

[1] M. F. Adams, J. Brown, J. Shalf, B. V. Straalen, E. Strohmaier, and S. Williams. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems*. Tech. rep. LBNL-6630E. LBNL, 2014.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. 2nd. Addison-Wesley, 2007.

[3] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. "Unified Form Language: A Domain-specific Language for Weak Formulations of Partial Differential Equations." In: *ACM Trans. on Mathematical Software (TOMS)* 40.2 (Mar. 2014), 9:1–9:37.

[4] S. Amarasinghe. *ZettaBricks: A Language Compiler and Runtime System for Anyscale Computing*. Tech. rep. DOE-MIT–0005288-1. MIT, 2015.

[2] https://www.ipython.org
[3] https://www.jsfiddle.net
[4] https://www.overleaf.com

[5] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice." In: *Proc. PLDI*. ACM, June 2009, pp. 38–49.

[6] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. "OpenTuner: An Extensible Framework for Program Autotuning." In: *Proc. PLDI*. ACM, Aug. 2014, pp. 303–316.

[7] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries." In: *Modern Software Tools in Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Birkhäuser Press, 1997, pp. 163–202.

[8] M. Bezem, J. W. Klop, and R. de Vrijer. *Term Rewriting Systems*. Vol. 55. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003. 884 pp.

[9] M. Blatt and P. Bastian. "The Iterative Solver Template Library." In: *Applied Parallel Computing. State of the Art in Scientific Computing*. Ed. by B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski. Vol. 4699. Lecture Notes in Computer Science (LNCS). Springer, 2007, pp. 666–675.

[10] M. Bolten, F. Franchetti, P. H. J. Kelly, C. Lengauer, and M. Mohr. "Algebraic Description and Automatic Generation of Multigrid Methods in SPIRAL." In: *Concurrency and Computation: Practice and Experience* 29.17 (2017). Special Section on "Advanced Stencil-Code Engineering", 4105:1–4105:11.

[11] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. "A Heterogeneous Parallel Framework for Domain-Specific Languages." In: *Proc. PACT*. IEEE. Oct. 2011, pp. 89–100.

[12] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. "SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization." In: *Proc. PMEA*. Available at ResearchGate. 2009.

[13] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman. "Autotuning Multigrid with PetaBricks." In: *Proc. SC*. ACM, Nov. 2009, 5:1–5:12.

[14] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. van Straalen. *Chombo Software Package for AMR Applications Design Document*. 2003.

[15] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. "From OpenCL to High-Performance Hardware on FPGAs." In: *Proc. FPL*. IEEE, Aug. 2012, pp. 531–534.

[16] R. D. Falgout and U. M. Yang. "hypre: A library of high performance preconditioners." In: *Computational Science (ICCS)*. Ed. by P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra. Vol. 2330. Lecture Notes in Computer Science (LNCS). Springer, 2002, pp. 632–641.

[17] A. Fernández et al. "Task-Based Programming with OmpSs and Its Application." In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by L. Lopes et al. Springer, 2014, pp. 601–612.

[18] M. Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010.

[19] M. Frigo and S. Johnson. "The Design and Implementation of FFTW3." In: *Proc. IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[20] A. Grebhahn, N. Siegmund, H. Köstler, and S. Apel. "Systems of Partial Differential Equations in ExaSlang." In: *Software for Exascale Computing – SPPEXA 2013–2015*. Ed. by H.-J. Bungartz, P. Neumann, and W. E. Nagel. Vol. 113. Lecture Notes in Computational Science and Engineering (LNCSE). Springer, pp. 69–88.

[21] D. Grünewald and C. Simmendinger. "The GASPI API Specification and its Implementation GPI 2.0." In: *Proc. PGAS*. The University of Edinburgh, Oct. 2013, pp. 243–249.

[22] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. "FLAME: Formal Linear Algebra Methods Environment." In: *ACM Trans. on Mathematical Software (TOMS)* 27.4 (Dec. 2001), pp. 422–455.

[23] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess. "STELLA: A Domain-Specific Tool for Structured Grid Methods in Weather and Climate Models." In: *Proc. SC*. ACM, Nov. 2015, 41:1–41:12.

[24] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer-Verlag, 1985.

[25] M. H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier, 1977.

[26] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. "Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines." In: *ACM Trans. on Graphics (TOG)* 33.4 (July 2014), 144:1–144:11.

[27] L. C. L. Kats and E. Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs." In: *ACM SIGPLAN Notices* 45.10 (2010). Proc. ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 444–463.

[28] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[29] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. "When Polyhedral Transformations Meet SIMD Code Generation." In: *Proc. PLDI*. ACM, June 2013, pp. 127–138.

[30] S. Kronawitter, S. Kuckuk, and C. Lengauer. "Redundancy Elimination in the ExaStencils Code Generator." In: *Proc. ICA3PP*. Ed. by J. Carretero et al. Vol. 10049. Lecture Notes in Computer Science (LNCS). 1st Int'l Workshop on Data Locality in Modern Computing Systems (DLMCS). Springer, 2016, pp. 159–173.

[31] S. Kronawitter and C. Lengauer. *Optimizations Applied by the ExaStencils Code Generator*. Tech. rep. MIP-1502. 10 pages. Faculty of Informatics and Mathematics, University of Passau, Jan. 2015.

[32] S. Kuckuk, G. Haase, D. A. Vasco, and H. Köstler. "Towards Generating Efficient Flow Solvers with the ExaStencils Approach." In: *Concurrency and Computation: Practice and Experience* 29.17 (2017). Special Issue on "Advanced Stencil-Code Engineering", 4062:1–4062:17.

[33] S. Kuckuk and H. Köstler. "Automatic Generation of Massively Parallel Codes from ExaSlang." In: *Computation* 4.3 (2016). Special Issue on High Performance Computing (HPC) Software Design, 27:1–27:20.

[34] S. Kuckuk, L. Leitenmaier, C. Schmitt, D. Schönwetter, H. Köstler, and D. Fey. *Towards Virtual Hardware Prototyping for Generated Geometric Multigrid Solvers*. Tech. rep. CS-2017-01. 8 pages. Dept. of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Mar. 2017.

[35] F. Lardinois. *A dormant chip in the Pixel 2 will soon let developers write better camera and AI apps*. 2017. URL: https://techcrunch.com/2017/10/17/googles-first-custom-consumer-chip-is-the-secret-behind-the-pixel-2s-camera-performance/ (visited on 11/02/2017).

[36] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proc. CGO*. IEEE Computer Society. Mar. 2004, pp. 75–88.

[37] R. Leißa, K. Boesche, S. Hack, R. Membarth, and P. Slusallek. "Shallow Embedding of DSLs via Online Partial Evaluation." In: *Proc. GPCE*. ACM, Oct. 2015, pp. 11–20.

[38] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt. "ExaStencils: Advanced Stencil-Code Engineering." In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by L. Lopes et al. Vol. 8806. Lecture Notes in Computer Science (LNCS). Springer, 2014, pp. 553–564.

[39] C. Liao, P.-H. Lin, D. J. Quinlan, Y. Zhao, and X. Shen. "Enhancing Domain Specific Language Implementations through Ontology." In: *Proc. WOLFHPC*. ACM. Nov. 2015, 3:1–3:9.

[40] A. Logg, K.-A. Mardal, and G. N. Wells, eds. *Automated Solution of Differential Equations by the Finite Element Method*. Vol. 84. Lecture Notes in Computational Science and Engineering (LNCSE). Springer, 2012.

[41] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. "Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets." In: *Proc. ICS*. ACM, June 2011, pp. 265–274.

[42] R. Membarth, O. Reiche, C. Schmitt, F. Hannig, J. Teich, M. Stürmer, and H. Köstler. "Towards a Performance-portable Description of Geometric Multigrid Algorithms using a Domain-specific Language." In: *J. Parallel and Distributed Computing (JPDC)* 74.12 (Dec. 2014), pp. 3191–3201.

[43] M. Mernik, J. Heering, and A. M. Sloane. "When and How to Develop Domain-Specific Languages." In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344.

[44] G. R. Mudalige, I. Reguly, M. B. Giles, C. Bertolli, and P. H. J. Kelly. "OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures." In: *Proc. InPar*. 12 pages. San Jose, California: IEEE, May 2012.

[45] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. 2nd ed. artima, 2011.

[46] M. Püschel, F. Franchetti, and Y. Voronenko. "Spiral." In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua et al. Vol. 4. Springer, 2011, pp. 1920–1933.

[47] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines." In: *ACM Trans. on Graphics (TOG)* 31.4 (July 2012), 32:1–32:12.

[48] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. "Firedrake: Automating the Finite Element Method by Composing Abstractions." In: *ACM Trans. on Mathematical Software (TOMS)* 43.3 (2016), 24:1–24:27.

[49] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. Kelly. "PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes." In: *Proc. WOLFHPC*. ACM, Nov. 2012, pp. 1116–1123.

[50] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. "SDSLc: A Multi-target Domain-specific Compiler for Stencil Computations." In: *Proc. WOLFHPC*. ACM, Nov. 2015, 6:1–6:10.

[51] O. Reiche, C. Kobylko, F. Hannig, and J. Teich. "Auto-Vectorization for Image Processing DSLs." In: *Proc. LCTES*. ACM, June 2017, pp. 21–30.

[52] T. Rompf and M. Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs." In: *Comm. ACM* 55.6 (June 2012), pp. 121–130.

[53] C. Schmitt, F. Hannig, and J. Teich. "A Target Platform Description Language for Code Generation in HPC." In: *Workshop Proc. ARCS*. VDE, Apr. 2018, pp. 59–66.

[54] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers." In: *Proc. WOLFHPC*. IEEE Computer Society, Nov. 2014, pp. 42–51.

[55] C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, H. Köstler, U. Rüde, and C. Lengauer. "Systems of Partial Differential Equations in ExaSlang." In: *Software for Exascale Computing – SPPEXA 2013–2015*. Ed. by H.-J. Bungartz, P. Neumann, and W. E. Nagel. Vol. 113. Lecture Notes in Computational Science and Engineering (LNCSE). Springer, pp. 47–67.

[56] C. Schmitt, S. Kuckuk, H. Köstler, F. Hannig, and J. Teich. "An Evaluation of Domain-Specific Language Technologies for Code Generation." In: *Proc. ICCSA*. IEEE Computer Society, July 2014, pp. 18–26.

[57] C. Schmitt, M. Schmid, F. Hannig, J. Teich, S. Kuckuk, and H. Köstler. "Generation of Multigrid-based Numerical Solvers for FPGA Accelerators." In: *Proc. HiStencils*. Jan. 2015, pp. 9–15.

[58] A. M. Sloane. "Lightweight Language Processing in Kiama." In: *Proc. GTTSE*. Ed. by J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva. Vol. 6491. Lecture Notes in Computer Science (LNCS). Springer. 2009, pp. 408–425.

[59] B. Subramaniam, W. Saunders, T. Scogland, and W. C. Feng. "Trends in Energy-Efficient Computing: A Perspective from the Green500." In: *Proc. IGCC*. 8 pages. IEEE, June 2013.

[60] *The Finite Element ToolKit (FETK)*. 2017. URL: http://www.fetk.org/ (visited on 10/20/2017).

[61] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments." In: *Proc. ICPPW*. IEEE Computer Society, Sept. 2010, pp. 207–216.

[62] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[63] D. Unat, X. Cai, and S. B. Baden. "Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C." In: *Proc. ICS*. ACM, June 2011, pp. 214–224.

[64] R. C. Whaley, A. Petitet, and J. J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project." In: *Parallel Computing* 27.1–2 (2001), pp. 3–35.