2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing

ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers

Christian Schmitt[‡], Sebastian Kuckuk[†], Frank Hannig[‡], Harald Köstler[†], and Jürgen Teich[‡]
[‡] Hardware/Software Co-Design, Department of Computer Science
[†]System Simulation, Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—High-Performance Computing (HPC) systems are becoming increasingly parallel and heterogeneous. As a consequence, HPC applications, such as simulation software, need to be especially designed towards these systems to achieve optimal performance. This, in turn, leads to higher complexity, making software engineers and scientists require a deep knowledge of the hardware and its technologies. As a remedy, domain-specific languages (DSLs) are a convenient technology for domain experts to describe settings and problems they want to solve using terms and models familiar to them. This specification is transformed into a target language, i. e., source code in another programming language or a binary executable, by a specialized compiler. We propose ExaSlang, a language for the specification of numerical solvers based on the multigrid method targeting distributed-memory systems. Furthermore, we present the transformation framework that drives the corresponding source-to-source compiler. It emits C++ code utilizing a hybrid OpenMP and MPI parallelization. Moreover, we substantiate our approach with scaling results of our code scaling up to the complete JUQUEEN cluster, consisting of 28,672 nodes, with a total of 458,752 cores.**

I. Introduction

Implementing numerical solvers in an efficient way poses a great challenge, as it requires profound knowledge not only of the application domain and its mathematical models, but also a solid programming experience in programming today's increasingly heterogeneous HPC clusters. Such systems might consist of different CPUs, complex memory architectures and accelerators such as Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs) in a single cluster node. It takes months to years for a programmer to learn programming and tuning of the different technologies utilized in a certain machine. When the code is to be executed on another platform, e. g., because the previous cluster is replaced with newer technology after a few years, new technologies have to be adopted and optimizations have to be re-evaluated. Very often, this takes the same time as writing the original optimized program—and has led to purchase decisions that favored inferior systems (e. g., with respect to performance, cost, energy efficiency) because of software backward compatibility reasons.

A common solution of this problem is the separation of algorithm and implementation by using domain-specific languages (DSLs). They allow domain experts to specify an algorithm to solve a certain problem without having to pay attention to implementation details. Instead, they can rely on the DSL compiler to generate a program with good—or even optimal—performance. Naturally, hand-written program code is usually faster than automatically generated code. However, instead of putting hardware and optimization knowledge into each application's implementation and only optimizing a single application, all of the programming expert's efforts are put into the compiler and, consequently, every program benefits. Thus, when changing hardware, only the compiler needs to be adapted to generate new program codes from the algorithm specifications. This is in contrast to library-based approaches where a newer version of the library potentially may break backwards compatibility, requiring changes to the original program and leading to laborious dependency management. Additionally, new algorithms can be evaluated much faster by providing only a short specification, resulting in a huge productivity increase. Another advantage of generative approaches is the validation of models. By providing language elements with corresponding constraints, a great number of invalid models cannot be specified at all. Additionally, semantic validation can be done in the compiler, avoiding generation of invalid programs.

Based on the language definition, there are two categories of domain-specific languages: *internal* (*embedded*) and *external* DSLs. Languages of the first category use the syntax of a host programming language—often a general-purpose language such as C, C++, Java, or Python—and extend or restrict it by introducing new domain-specific language elements like special data types, routines or macros. On the other hand, since external DSLs introduce a completely new syntax and semantics, they are, in general, more flexible and expressive than internal ones at the cost of a higher design effort. At the same time, they constrain the models that can be specified, reducing the number of potential errors for the end-user. By providing a semantic model in their intermediate representation that is much more powerful than an abstract syntax tree (AST), they also enable broader (domain-specific) transformations, e. g., optimizations.

Domain-specific languages, just like general-purpose programming languages, may be represented in a textual or in a visual way. Very often, however, visual programming languages like LabVIEW[1] and Simulink[2] are domain-specific languages in reality. Their visual representation often is the most natural way of describing scenarios in certain domains and thus, in the spirit of the intended use, further eases development for domain experts. A more complete overview over motives and techniques for the design and implementation of DSLs can be found in [19].

Multigrid methods are very popular for the solution of partial differential equations (PDEs). In ExaStencils[3], we work on a language and a source-to-source compiler to generate high-performance multigrid code for upcoming exascale supercom-

[1]http://www.ni.com/labview/
[2]http://www.mathworks.de/products/simulink/
[3]http://www.exastencils.org

978-1-4673-6757-8/14 $31.00 © 2014 IEEE
DOI 10.1109/WOLFHPC.2014.11

puters. Our goal is to process a high-level, abstract specification of the numerical solver into low-level C++ code. The resulting program uses a hybrid parallelization based on OpenMP and MPI, enabling it for large-scale clusters. A more extensive project description can be found in [15]. In previous work, we have demonstrated the feasibility of this approach by showcasing a simple prototype compiler for the generation of parallel multigrid solvers with low variability [12]. It featured a lower expressivity in terms of language elements and large, monolithic program code transformations, hindering the introduction of new language features and program optimizations. This showed the need for the flexible language processing framework presented in this work. In further previous work, HIPAcc—a DSL for image processing supporting multigrid methods by providing corresponding language elements [18]—was demonstrated to generate multigrid code for accelerators [25].

This paper is structured as follows: In Section II, we provide an overview of other work done in the field of domain-specific language engineering for mathematical problems and stencil codes. In Section III, we introduce basics and terms of the multigrid idea that are used throughout this paper. In Section IV, we introduce our general approach towards a multigrid DSL. In Section V, we give a more detailed description of the proposed language and its elements. We provide an introduction to the compiler and the framework in Section VI. We present a glance of first results of a scalability evaluation on a large HPC system in Section VII, whereas we address future research and implementation work, building on top of the work presented in this paper, in Section VIII. Finally, in Section IX, we conclude this work.

II. Related Work

In previous work, the benefits of domain-specific optimization have been shown in various domains. *SPIRAL* [22], for example, is a widely recognized framework for the generation of hard- and software implementations of digital signal processing algorithms (linear transformations, such as FIR filtering, FFT, and DCT). It takes a description in a domain-specific language and applies domain-specific transformations and auto-tuning techniques to improve runtime performance on a target hardware platform. *ATLAS* [30] and *FFTW* [11] are examples for the generation of mathematical code from abstract descriptions for specific applications such as FFTs, where further optimizations are done by auto-tuning.

For the domain of stencil computations, many languages and corresponding compilers have been proposed: Examples include *Liszt* [8], which adds abstractions to Java to ease stencil computations for unstructured problems, and *Pochoir* [28], which employs a divide-and-conquer skeleton on top of the parallel C extension Cilk to make stencil computations cache-oblivious. *PATUS* [6] uses auto-tuning techniques to improve performance. In comparison to the approach presented in this work, they operate on a lower tier of abstraction and do not provide language support for multigrid methods.

Mint [29] and *STELLA* (STEncil Loop LAnguage) [21] are DSLs embedded in C, respectively C++, and consider stencil codes on structured grids. Mint's source-to-source compiler transforms special annotations into high-performance CUDA code, whereas STELLA additionally supports OpenMP for parallel CPU execution. For both, distributed-memory parallelization is currently not available.

Multigrid methods are used by a large variety of libraries. Examples are *Peano* [5], which is based on space-filling curves, the *Boomer AMG* [9] for unstructured grids and general matrices which is part of the highly scalable *hypre* [1] software package, a collection of preconditioners and solvers, and *DUNE* [2], a general software framework for the solution of PDEs.

In the past, several approaches for the generation of low-level stencil code from abstract descriptions have been made, however, to the best of our knowledge only few are centered around the multigrid methods for exascale machines.

Julia [3] centers around the multiple dispatch concept to enable distributed parallel execution. It builds on a Just-in-time (JIT) compiler and can also be used to write finite element codes. It works on a lower level of abstraction as the approach presented in this work.

Multi-stage programming is the key concept behind *Terra* [7], which embeds a statically typed DSL where memory is manually managed into the dynamically typed host language Lua, featuring automatic memory management. Terra builds on the Clang/LLVM infrastructure and uses LuaJIT for just-in-time compilation. It does not provide distributed memory support.

HIPAcc [17] is a DSL for the domain of image processing and generates OpenCL and CUDA from a kernel specification embedded into C++. It is based on the Clang/LLVM compiler infrastructure. It recently was extended to provide support for image pyramids which are data structures for multi-resolution techniques that are very similar to multigrid methods[4]. In contrast to this work, it supports only 2D operations and does not consider distributed-memory parallelization such as MPI.

The finite element method library *FEniCS* [16] provides a Python-embedded DSL called Unified Form Language (UFL). Multigrid support is available via PETSc, which provides shared-memory and distributed-memory parallelization via pthreads and MPI, as well as support for GPU accelerators. Our approach and domain-specific language presented in this work are focused on another class of users and work on a much more abstract level.

PyOP2 [24] uses Python as the host language. It targets mesh-based simulation codes over unstructured meshes and uses FEniCS to generate kernel code for different multicore CPUs and GPUs. Furthermore, it employs runtime compilation and scheduling. It does not feature the extensive, domain-specific, automatic optimizations that are one of the goals of the ExaStencils project as described in [15] and for which this work constitutes the working base.

Delite [4] is a compiler and runtime infrastructure for the construction of implicitly parallel DSLs in Scala. It provides parallel building blocks for shared-memory parallelization which can be mapped efficiently to parallel execution patterns. A DSL program can be split into such blocks to benefit from their re-combination and optimization. As an example, Deliszt is a re-implementation of the previously introduced Liszt [8] using Delite. In this work, an alternative approach for external, explicitly parallel DSLs for distributed-memory parallelization is presented.

The compiler infrastructure *ROSE* [23] eases analysis and source-to-source transformation of C/C++ and Fortan codes

[4] http://hipacc-lang.org/

by providing corresponding front ends, back ends, and data structures. As such, it can solely be used for embedded DSLs and has no explicit support for parallelization efforts.

Another compiler infrastructure is *LLVM* [14], forming the base for *Clang*, a new C++ compiler. Due to its modularity, it is popular for the creation of C++-embedded domain-specific languages and compilation to binary code. For example, it is the foundation of the above-mentioned HIPAcc and Terra. The framework is not suitable for the presented work, as the proposed DSL is not embedded into any host language and its intermediate representation works on a different level of abstraction compared to LLVM IR, since it is only partially AST based.

Kiama [27] is a Scala library for language processing and provides support for specification of program transformations and analysis. It is very similar to the implementation of transformation presented in this paper, but offers many more developer features currently not necessary for our framework. The work presented in the paper was designed with modularity in mind, so our internal transformation implementation could be changed to Kiama at a later stage without additional costs.

III. Multigrid

The multigrid idea is based on two principles: The smoothing property, i. e., that classical iterative methods like Jacobi or Gauss-Seidel (GS) are able to smooth the error after very few steps, and the coarse grid principle, i. e., that a smooth function on a fine grid can be approximated satisfactorily on a grid with fewer discretization points. Multigrid combines these two principles into a single iterative solver. The recursive algorithm traverses between fine and coarse grids in a grid hierarchy.

One simple multigrid iteration, the so-called *V-cycle*, is summarized in Algorithm 1. In the pre-smoothing step, high-frequency error components are damped first. Afterwards, a new error approximation, called residual, is calculated. Its low-frequency error components are then approximated on coarser grids (restriction of the residual). Returning from the recursive call, the residual is 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).

if *coarsest level* **then**
  solve $A^h u^h = f^h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A^h, f^h\right)$ {pre-smoothing}
  $r^h = f^h - A^h \bar{u}_h^{(k)}$ {compute residual}
  $r^H = R r^h$ {restrict residual}
  $e^H = V_H\left(0, A^H, r^H, \nu_1, \nu_2\right)$ {recursion}
  $e^h = P e^H$ {prolongate error}
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ {coarse grid correction}
  $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A^h, f^h\right)$ {post-smoothing}
end

Algorithm 1: Recursive V-cycle to solve
$$u_h^{(k+1)} = V_h\left(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2\right).$$

A note to the reader: In this paper, *level* will denote the *multigrid level* corresponding to the granularity of the grids, whereas *layer* will denote an element of the hierarchy of our programming language, as explained in the following section.

IV. ExaStencils DSL

When creating a new programing language—especially a DSL—it is of utmost importance to keep the user experience in mind. A language that is too complex will not be used by novice users, whereas a very abstract language will not be used by expert users. For our DSL, we identified three categories of users, each of which has a different expectation of an "optimal" language:

- Engineers & natural scientists: They expect a simple language to describe problems in a mathematical formulation. This could be done as an energy functional that is to be minimized or a partial differential equation to be solved on a given (computational) domain with corresponding boundary conditions. More advanced users of this category might want to make changes to the discretized equation, e. g., to compare convergence and correctness of different approaches.
- Mathematicians: These users have minor interest in the (physical) problem to be solved, but are more interested in the applicability and solution of mathematical models. Experts might want to make changes to parts of the (multigrid) algorithm, e. g., specify different components to be used. In general, however, they do not care about parallelization and low-level implementation details.
- Computer scientists: This category of users might be interested in the physical and mathematical models, but definitely cares about the software engineering approach and implementation details like communication and memory access patterns, as well as the optimal usage of the available hardware.

ExaStencils is a fundamental research project very focused on a single application domain. Implementation of large simulations involving a great diversity of different mathematical models or complex workflows is currently out of ExaStencils' scope. Our goal is to find out how to obtain optimal performance on tomorrow's highly heterogeneous HPC clusters in an automated way. As such, we regard the ExaStencils DSL *ExaSlang* and the framework behind it as a case study for the generation of heterogeneous application code from abstract specifications.

A. Multi-layered Approach

As pictured in Figure 1, our DSL consists of four layers that are derived from the user groups specified previously and the described overlap in user knowledge and expectancies. Thus, each layer features a different tier of abstraction.

At the most abstract layer, called Layer 1 or ExaSlang 1, the problem is defined in the form of an energy functional to be minimized or a partial differential equation to be solved, with a corresponding computational domain and boundary definitions. In any case, this is a continuous description of the problem. We propose this layer for use by natural scientists and engineers that have little or no experience in programming.

Layer 2 is a bit less abstract by making it possible to specify the problem in a discretized formulation. We deem this layer suitable for more advanced natural scientists and engineers as well as mathematicians.

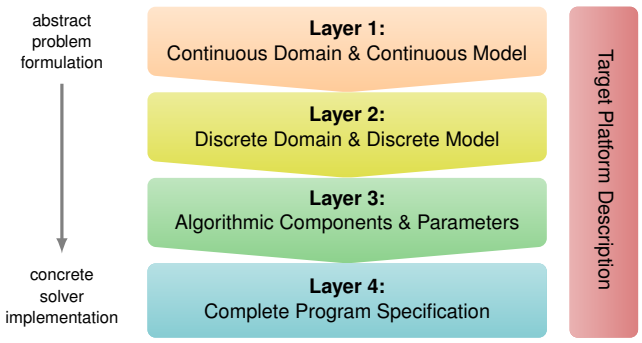


Fig. 1: Multi-layered approach of ExaSlang

The next layer is more concrete again. With ExaSlang 3, algorithmic components, settings and parameter values are modeled. As they build on the discretized problem of Layer 2, this is the first layer at which the multigrid method—which the ExaStencils approach is built on—is discernible. At this layer, it is possible to define smoothers and to modify the multigrid cycle. Computations are specified with respect to the complete computational domain. Since this is already a very advanced layer in terms of algorithm and discretization details, we mainly see mathematicians and computer scientists working at this layer.

The most concrete layer is ExaSlang 4, where user-relevant parts of the parallelization become accessible. Data structures can be adapted for data exchange and, through simple statements, communication patterns can be specified. We classify this layer as semi-explicitly parallel and see only computer scientists using these features. A detailed description of its key elements is given in the next section.

Unavailable to the user and thus not illustrated in Figure 1 is the internal representation following Layer 4. As it is the bridge between user input and compiler output, and exists only temporary in the compiler's memory, we call it Intermediate Representation (IR).

Orthogonal to the functional description of program is the Target Platform Description Language (TPDL), which specifies not only the hardware components of the target system such as CPUs, memory hierarchies, accelerators, and cluster topology, but also available software such as compilers or MPI implementations. However, its description is currently work in progress and out of the scope of this paper.

V. Language Elements

ExaSlang 4 is centered around the procedural programming paradigm and provides typical language elements, such as data types, functions, loops, variables, and constants. It is realized as an external domain-specific language to provide higher expressivity and readability to domain experts. In places, its syntax is inspired by Scala to ease adoption for novice users. In general, we deem it sufficiently different not to be confused with Scala.

In the following subsections, concepts and elements of the language are presented. Because ExaStencils is an active research project, some of these concepts, as well as concrete syntax, may still change at a later point in time.

A. Level Specifications

To define a mapping between functionality and multigrid levels, *Level Specifications* are used. Being a generic concept and stemming from its multigrid domain, this is used at many points in our language definition. In ExaStencils, we define multigrid level zero to work on the coarsest grid, hence a higher level number corresponds to a finer grid.

Levels can be specified in a number of ways: (a) The simplest case is to explicitly enumerate all levels, either by providing a complete object definition per level or by supplying a list of levels for the object definition. (b) As listing all levels can be cumbersome, ranges can be specified via lower and upper bounds, separated by the keyword `to`. (c) Levels to be excluded from a range can be specified via the keyword `not` that takes a single level, a list, or a range of levels, e. g., `@(0 to 8, not(4 to 6))`.

To ease addressing, keywords that mark special levels have been introduced: (a) `coarsest` and `finest` that correspond to the lowermost and the uppermost level, (b) `current` that can be used inside functions for accessing other objects on the same grid granularity, and (c) `coarser` and `finer` that address levels adjacent to the current one.

Furthermore, relative references further than one level away are possible by supplying a step size and a direction. Naturally, by increasing the level number, finer grids can be accessed, whereas decreasing the level number corresponds to accessing coarser grids. For example, to address the level that is two coarsening steps away, a notation like `@(current - 2)` may be used. Of course, different level specifications can be combined, yielding specifications such as `@(coarsest+1 to finest-1, not(coarest+3, finest-3))`.

A common example is the use in the recursive multigrid call, where calls to the V-cycle function to work on the next coarser grid are made using the relative addressing `VCycle @coarser ()`, as illustrated in Listing 1.

```
1  Function VCycle @((coarsest + 1) to finest)
       () : Unit {
2      repeat 3 times { Smoother @current () }
3      UpResidual @current ()
4      Restriction @current ()
5      SetSolution @coarser (0)
6      VCycle @coarser ()
7      Correction @current ()
8      repeat 2 times { Smoother @current () }
9  }
10
11 Function VCycle @coarsest () : Unit {
12     /* ... solve directly ... */
13 }
```

Listing 1: Example of a V-cycle specification that maps directly to the algorithm presented in Algorithm 1. Solution at the coarsest grid is specified by overloading the `VCycle` function at the corresponding multigrid level.

B. Data Types and Variables

ExaSlang 4 supports the most common data types: `Real` that holds floating-point numbers and `Integer` for whole numbers,

`String` for the definition of character sequences, `Boolean` as the result of comparisons and `Unit` to define functions that do not return any value. These data types belong to the group of *simple data types*.

Furthermore, there is the category of *aggregate data types*, currently consisting of `Complex` for the definition of complex numbers where the underlying data type needs to be chosen between real and integer. As a generalization, we plan to add the data type `Vector`, which can hold an arbitrary (but small) number of elements.

The third class of data types stems from the multigrid domain and is used exclusively for numerical calculations. As such, it is called *algorithmic data types*. It consists of the types `Stencil` and `Field`. In a mathematical sense, they correspond to matrices and vectors, respectively. Both types are explained in more detail in the corresponding subsections.

Variables of simple and aggregate data types are declared using the keyword `Variable`, or shorter `Var`. It is followed by a colon and the designated data type. Optionally, it may be initialized by supplying a corresponding expression after an equals sign. Variables with a constant value, i. e., that cannot be changed after their definition, are defined by the keyword `Value` (short: `Val`), followed by a colon and a data type identifier. In case of values, initialization is mandatory. Example definitions are shown in Listing 2. Note that only simple and aggregate data types can be used for variables and values.

```
1 Variable alpha : Real = 0.1
2 Var beta : Real = calculateBeta() + 0.25
3
4 Value twoPi : Real = 6.28
5 Val six : Integer = 2 * 3
```

Listing 2: Examples of variable and constant definitions.

C. Fields and Layouts

Mathematically speaking, *Fields* are vectors, i. e., discretized variables. As such, they may be given by the user, e. g., as the right-hand side of a partial differential equation, be an unknown to be solved for, represent "temporary variables" like the residual and, of course, be the result of a calculation. In a technical sense, they are arrays of a certain size and, thus, their size is linked to the size of the computational domain.

For the definition of a field, a *Layout* is essential. It enables the field for communication among the domain partitions by specifying the number of boundary layers to be exchanged. In future implementations, extended layout options, such as color splitting, will be supported. To define multiple copies of the memory, e. g., when implementing a Jacobi smoother, *Slots* can be used.

In the multigrid algorithm, it is necessary that fields are instantiated at multiple levels. At Layer 4, this corresponds to using the level concept for field declaration and access, as described in Section V-A. Furthermore, fields usually vary in size for different levels, which, in turn, leads to level-specific field layouts as well.

D. Stencils

Stencils are the heart of many multigrid algorithms. They basically represent matrices in calculations and are used for vital tasks such as implementing smoothers or correction/prolongation functions. In Listing 3, a 5-point stencil using fixed weights is shown. In fact, any expression like (scalar) mathematical operations and function calls can be specified as weights.

Stencils are specified using an offset from the grid node either by specifying a fixed distance, e. g., ± 1 in certain directions, or by using binary expressions. Note that stencils can also be specified on a per multigrid level base by providing a level specification (see Section V-A), as shown in Listing 3.

```
1 Val kappa : Real = /* ... */
2 Stencil OperatorStencil @finest {
3      [ 0,  0] => (4.0 + kappa)
4      [ 1,  0] => -1.0
5      [-1,  0] => -1.0
6      [ 0,  1] => -1.0
7      [ 0, -1] => -1.0
8 }
```

Listing 3: Example of a 5-point stencil including a binary expression.

E. Control Flow

As expected of a procedural programming language, ExaSlang 4 provides language elements such as functions, branching and loops. Functions can take optional parameters and have a return type. `Unit` has to be specified in the case that no value is to be returned. Functions can optionally be assigned to specific multigrid levels by providing a level specification (see Section V-A). If a function does not carry a level specification, it is considered a utility function (as opposed to an algorithmic function) that is available at all levels. ExaStencils provides a number of basic mathematical functions such as `sin`, `cos`, `exp`, or `sqrt` as built-in, which can be called without previous definition in the DSL program.

By specifying a function with the signature `Function Application() : Unit`, a C++ `main()` method acting as the main entry point to the generated program is added to the output code. If this is omitted, the generated code is supposed to be part of a user application.

Branching is realized via the traditional if-else statement, followed by a boolean expression such as a comparison. To improve clearness of the DSL code and reduce developer errors, single-line if-statements are not allowed.

Loops are specified in a more functional way. For implementing a while-loop, e. g., for repeating iterations until the residual is below a certain threshold, ExaSlang provides the `repeat until` statement. A simple example is shown in Listing 4. While this also allows the specification of a fixed number of repetitions, declaring and incrementing a separate variable is unnecessarily complex. As syntactic sugar, the language provides the `repeat N times` statement, where `N` is an integer literal. An example can be seen in Listing 5.

```
1 Var eps : Real = 1
2 repeat until eps < 0.0001 {
3     eps = /* ... */
4 }
```

Listing 4: Example of repeating statements until a certain criteria is fulfilled.

```
repeat 10 times {
    /* ... do something ... */
}
```

Listing 5: Example of repeating statements a fixed number of times.

To iterate over the computational domain, the `loop over` statement is used. It is followed by a number of arguments:

1) The identifier of a field, including the multigrid level. This is used to determine the loop boundaries, i. e., the size of the iteration space. The optional `step size` parameter (not shown in the example Listing 6) specifies the size of sub-blocks into which the domain is to be divided. Furthermore, by the specification of conditions, advanced features, such as color splitting required for the implementation of red-black Gauss-Seidel smoothers, can be realized.
2) An optional reduction operation that, for example, can be the sum or the product of a variable that is used inside the loop body.

A complete example of a `loop over` statement is presented in Listing 6.

```
Field Residual /* ... */

Function L2Residual() : Real {
    Var res : Real = 0
    loop over Residual @current with reduction(+ : res) {
        res += Residual @current * Residual @current
    }
    return ( sqrt(res) )
}
```

Listing 6: Example of a loop over the computational domain, with a reduction operation to calculate the residual.

F. Communication Statements

An important aspect of specifying parallel programs is to define points of communication, i. e., spots in the program execution where data between fields needs to be synchronized. Here, it is vital to strike a good balance between automatic deduction of information and giving users the power to express their concepts and ideas. If all communication is added automatically and hidden from the user, confidence in the correctness of the program might decrease. At the same time, it prevents users from evaluating different communication patterns and their effects on convergence and execution time. On the other hand, it is not feasible to model every detail explicitly in the DSL.

Thus, we have decided to provide basically two different options to specify communication. Both of them can be generated automatically when setting up Layer 4, but still be reviewed and adapted by the user later on. The first possibility is to specify a synchronous data exchange providing the identifier of the field to be handled and, if applicable, a corresponding slot, as depicted in Listing 7.

Please note that "synchronous" in this context means that the `Communicate` operation can be treated as a single operation. The actual implementation, i. e., if synchronous or asynchronous MPI operations for send and receive operations are used or when OpenMP parallel data transfers are performed, is still decided by the compiler.

```
Communicate Solution[0] @current
```

Listing 7: Example of a (synchronous) communication statement to synchronize data in a certain slot of a field on a specific level.

The second alternative is required to enable overlap between communication and computation. To this end, the statement shown previously is split into two new ones, namely `StartCommunication` and `FinishCommunication`, resulting in code as shown in Listing 8.

```
// Solution data becomes available
StartCommunication Solution[0] @current
// do some work until Solution data is required
FinishCommunication Solution[0] @current
```

Listing 8: Example of two basic communication statements enabling overlap between computation and data exchange.

VI. Compiler and Framework

To refine user input from one DSL layer to another and to finally generate C++ code, a flexible transformation framework to power the compiler is needed.

The compiler for ExaSlang and its empowering framework is written in Scala, based on the results of a review of different programming language implementation technologies done earlier [26]. Scala is an object-functional general-purpose programming language [20], which means it is both object-oriented (e. g., like Java or C++) and functional (e. g., like Haskell) at the same time. It is actively developed at EPF Lausanne and runs inside the Java Virtual Machine (JVM). Parsers, which are software components that match streams of tokens according to predefined rules—the grammar—and execute code accordingly, can be implemented effortlessly by the very powerful feature of Parser Combinators. As Scala is binary compatible to standard Java code, a tremendous number of existing libraries can be employed. Furthermore, this also enables access to the Java Native Interface (JNI), which allows to call binary code available in the form as platform-specific binary shared libraries. Because of features like Scala Reflection and Macros, at least Scala 2.11 is required.

When designing the compilation framework, we not only derived tasks from the workflow as illustrated in Section IV, but additionally defined requirements that lay the ground for future research:

- Traceability: We care not only about the generated output code, but also about the path of transformations taken from the input to the output code. Therefore, the decisions made by the compiler modules and the intermediate states of the generated program are of great interest to us and should still be available after code generation is done. We not only use this for debugging purposes, but we can also improve future generated programs by constructing a feedback loop.
- Variant Generation: We will generate a large number of programs where each code will differ only in a small percentage to build a learning base. Therefore, we do not want to re-create the whole program every time but save time simply rolling back to a previous program

state, then continuing code generation by applying another transformation than in the previous compilation run.

A. Architecture

The architecture of the compiler is arranged into several modules, where specialized data structures are used as interfaces. This not only improves re-usability of the modules, but also enables a better collaboration between the developers in our project.

With respect to functionality, the language processing framework is organized in a number of components. In the code, this results in a clear division of the program into several Scala namespaces. The entities providing the core functionality of the compiler framework, e. g., everything that deals with program transformations, can be found in the namespace `core`. Basic functionality such as diagnostic output or handling of compiler settings (e. g., output paths for the generated) also belongs to this category. Another vital part is the reading of input files which is done by the parsers in the namespace `parsers`.

B. Data Structures

Crucial to any compiler are data structures. The four layers of our language are represented in a number of data structures that are assigned to corresponding Scala namespaces, namely `datastructures.{l1, l2, l3, l4}`. To ease development, the data structures that are created during parsing are a close match to each statement before being processed.

As most of the transformations occur at the IR layer, there is also a namespace `datastructures.ir` housing the common data structures. It also has—by far—the highest number of data structures, ranging from data structures that are very similar to the ones that exist at Layer 4, but also data structures that are close to the generated C++ code. Highly specialized node types, e. g., nodes handling parallelization aspects, can be found in distinct namespaces, which usually also house the corresponding transformations and strategies.

In general, we try to model as few different node types as possible but as many types as needed. Because of this adaptive approach, no single level of abstractness can be found throughout the compiler.

C. Transformations

In compilers, *Transformations* are a key element in refining the input program to the output program. They alter the program by modifying, adding, removing or replacing elements.

Formally, transformations consist of two parts: an input element, which is the element to be searched for, and an output element, which replaces the input element. This output element can take one of three forms: (a) an empty element, which means the input element will be removed from the program, (b) a list of elements to replace the original single input element—however, this may not be applicable in all cases, or (c) another element that replaces the input element, which may also be the modified input element. In any case, it has to inherit directly or indirectly from the abstract base class `Node`. If so, all writable member variables of the object are considered for modification. The same applies to container data types such as lists, arrays, or maps.

Two examples are presented in Listing 9. In the first transformation, the input element is modified and returned, whereas in the second transformation, a node that is matched is replaced by a completely new one, also removing the sub-nodes `left` and `right` from the program.

Every transformation carries an identifier to reconstruct the generation path of a program and to learn about optimal combinations of program configurations and target hardware. From this, we build a feedback loop to improve the programs generated.

Furthermore, the input element can be matched at a finer grain by using so called *Guards*. They extend the matching capabilities by providing access into the objects themselves and allow to check member variables of the input elements. Only if the guard is evaluated positively, the input element is matched and processed further. An example can be found in the first transformation in Listing 9, where the target function's name has to be exactly `foo`.

D. Strategies and Transactions

To execute transformations, *Strategies* are needed. They can be thought of as containers that cluster all transformations related to a specific step in the program generation process.

Just like transformations, each strategy carries a name to make its execution traceable. There are two types of strategies:

(a) The default strategy merely is a container to which transformations can be added. When executed, it applies the transformations in the order they have been added.
(b) Custom strategies allow fine-grained control over the execution of transformations. They support conditional and looped execution of transformations.

Application of strategies can only occur in the scope of a *Transaction*. It provides exclusive access to the program's current state so no concurrent transactions can write at the same time, which could otherwise lead to data corruption. A transaction can be successfully completed by committing, while aborting leads to an end of the current compilation process and—if applicable—resets the program to a state saved previously.

```
1 var s = DefaultStrategy("example strategy")
2
3 // rename a certain function
4 s += Transformation("rename fct", { case x :
       FunctionStatement if(x.Name == "foo")
      => x.name = "bar"; x})
5
6 // evaluate additions
7 s += Transformation("eval adds", { case
      AdditionExpression(left :
      IntegerConstant, right : IntegerConstant
      ) => IntegerConstant(left + right) })
8
9 s.apply // execute transformations
      sequentially
```

Listing 9: Example of a default strategy containing two transformations. The workflow is illustrated in Figure 2.

E. Collectors

A generalization of scopes or contexts—known from compiler frameworks such as LLVM and GCC—are *Collectors*. In general, they provide context information when traversing and modifying the program's current state, e. g., a list of variables accessible within a function. In our framework and by default,

a collector is a data structure that keeps track of the parent nodes of the node that is currently traversed.

A use case would be checking the definition of variables in nested scopes, which is traditionally done with the use of contexts. For example, imagine a variable definition using a binary expression inside a `repeat` loop, which, in turn, is part of a function body. In this case, the newly declared variable is available neither in the binary expression defining it nor in the outer scopes, i. e., the function body. The compiler has to diagnose such situations and, in case of errors, emit corresponding messages to the user.

F. Annotations

To attach extra information to nodes, *Annotations* are used. There is no limit as to how many annotations can be added to a node. In its simplest form, it is a key/value pair, where the key is a (unique) identifier and the value can be of any data type available in Scala. Transformations solely based on the recognition of annotations are possible as well.

A common example is the location in the input file (i. e., file name, line number, and column) that is added to the data structures during parsing and is used for error messages when doing semantic validation.

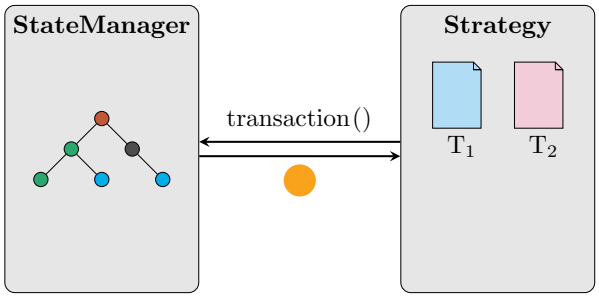
G. StateManager

This entity is the key object in the transformation framework and is the only component that has direct access to the generated program's state. Thus, it provides the transaction interface that strategies use. Transactions prevent concurrent strategies on the same program state, thus preventing the modification of intermediate states between transformations of a strategy that might lead to corruption of the generated program.

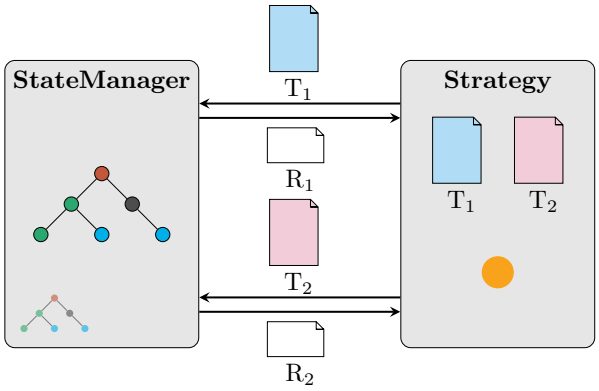
Furthermore, it also provides means to create checkpoints of the program to be restored in case a transformation fails, or to save code generation time when creating many similar programs by providing a base from where to transform the program into another direction compared to previous compilation runs. Thus, this functionality is similar to checkpointing of simulation codes during runtime. However, there is no relation to checkpoint/restart capabilities that might be added to the generated program. The duplication facilities can also be used to generate a large number of program variants whose performance results serve for the optimization of successively generated programs. This is a direct result of the aforementioned requirement of variant generation. The same cloning functionality can also be applied to smaller portions of the program, which is a common use case, e. g., for function inlining where the function body replaces the calls to the very function.

Figure 2 illustrates the interaction between a strategy and the StateManager when applying transformations. In Figure 2a, the strategy opens a new transaction with the StateManager. A transaction token, permitting access to the transformation interface, is returned and the strategy is now free to execute its transformations. Internally—and depending on the compiler configuration—the program has been saved automatically using the aforementioned checkpointing functionality. In Figure 2b, a copy of the program can be seen in the lower left corner of the StateManager box.

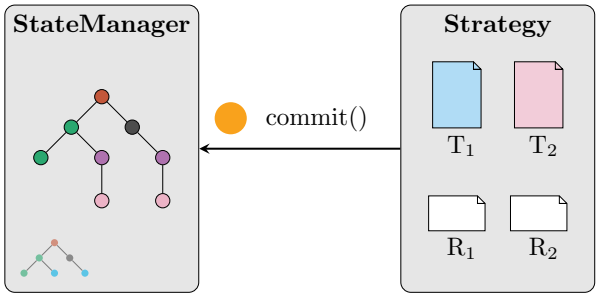
No specific order of transformation execution is enforced, e. g., a strategy may also conduct only a subset of its transformations, or perform the transformation numerous times. Transformations are exclusively executed by the StateManager, which returns some statistics to the strategy, e. g., the number of nodes that matched the transformation's pattern. This is illustrated in Figure 2b. When the strategy is done, it can either close the transaction successfully—as illustrated in Figure 2c—by handing back the token to the StateManager, or abort the transaction. Depending on the configuration of the compiler, a previous program state can be loaded, or the complete code generation process is canceled.



(a) A strategy opens a new transaction with the StateManager. A transaction token is returned.



(b) The strategy executes transformations T_1 and T_2. The StateManager returns statistics S_1 and S_2. Note the checkpoint of the program state that has been saved by the StateManager.



(c) The transaction is being committed, i. e., ended successfully by the strategy. The transformation token is returned to the StateManager. The previously made checkpoint is still available for later use.

Fig. 2: Workflow of a strategy applying transformations via the StateManager. Example code for this workflow is shown in Listing 9.

H. Prettyprinting

The output of our compiler is C++ code to be compiled with standard C++ compilers such as Clang or GCC as well as vendor-supplied ones like the IBM XL C/C++ compiler. To enable parallel compilation, functions are distributed across many smaller files. This helps reduce compilation times, e. g., on JUQUEEN by one order of magnitude when running the C++ compiler with recommended optimization flags.

In the code, an entity called PrettyprintingManager keeps track of files and their dependencies. When prettyprinting, it adds header guards where needed and resolves dependencies to corresponding preprocessor declarations. Program state nodes are assigned to classes, which are mapped directly to output files.

To cope with the number of generated files, accompanying Makefiles are generated automatically, where compiler flags are taken from the hardware description. Makefile generation is modeled in an abstract way, allowing output to other formats such as CMake.

VII. Preliminary results

To verify the functionality of our framework and roughly assess the performance of the generated code, we regard a simple example problem. In our case, this corresponds to solving a 3D finite difference discretization of Poisson's equation, a representative elliptic PDE used to model diffusion processes. As such, its usage in real-word application is widespread.

Our generated geometric multigrid solver has the following characteristics: a V(3,3)-cycle using a Gauss-Seidel smoother, a parallel Conjugate Gradient (CG) solver for handling the coarsest grid, 4 threads per core equivalent to 64 threads per node and 1 million unknowns per core.

For our benchmark, we have performed a basic weak scaling on the JUQUEEN supercomputer located at Jülich, Germany. Our generated code has been evaluated up to the full machine consisting of 28,672 nodes (458,752 cores). As evident from the results shown in Figure 3, scalability is satisfactory for this kind of setup. A more in-depth analysis suggests that the increase in runtime is caused almost exclusively by the CG solver, a behavior that matches previous findings [13]. With a growing number of threads, the number of unknowns and thus the number of required coarse-grid iterations increases, while, additionally, required global communication operations also become more costly.

However, it is also important to see these preliminary results in perspective: On the one hand, this code is not optimized yet, i. e., common techniques such as tiling, temporal blocking, vectorization, loop unrolling, or address pre-calculation, are not taken into account. Therefore, no comparison to hand-tuned code or other available multigrid frameworks was made and a performance model has been omitted. On the other hand, the present setup yields a very low ratio between the execution times of the mostly perfectly scalable components, like the smoother and the residual calculation, and the coarse grid solver. Thus, for more complicated problems, where this ratio shifts, the observable overall scalability would improve.

In table I, the lines of code of typical programs formulated in ExaSlang 4 have been compared to the generated C++ code. Every program consists of a V(4,2) cycle, the same number of levels and uses the Conjugate Gradient method for solving on the coarsest grid. As the table illustrates, output sizes depend vastly on the utilized communication implementation, whereas the lengths of the input files do not differ. Overall, this demonstrates the effectivity of the generative approach that is the heart of ExaStencils.

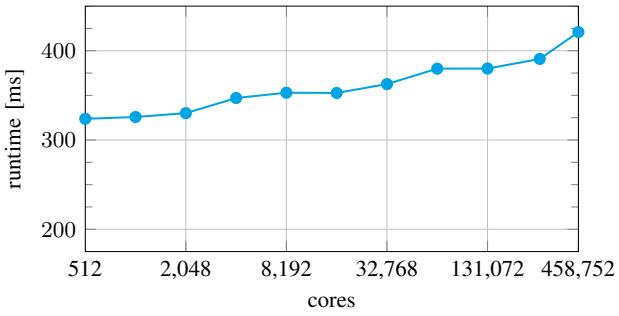


Fig. 3: Weak scaling for the mean time of one V(3,3)-cycle on JUQUEEN, using up to the complete machine of 28,672 nodes (458,752 cores). Runtimes rise with an increasing number of nodes because of iteration numbers and communication costs.

Smoother	Stencils	Comm.	DSL	C++
Jacobi	constant	simple	231	12,821
Jacobi	constant	extended	231	28,331
Jacobi	varying	simple	292	19,330
Jacobi	varying	extended	292	47,814
Gauss-Seidel	constant	simple	227	12,726
Gauss-Seidel	constant	extended	227	28,230
Gauss-Seidel	varying	simple	288	19,229
Gauss-Seidel	varying	extended	288	47,713

TABLE I: Overview of the sizes of typical ExaSlang Layer 4 programs and generated output code in lines of code.

VIII. Future work

While we have shown that highly scalable low-level code can be generated from the high-level DSL, it is essential to apply further optimizations. We plan to apply optimizations at the different stages of the transformation pipeline, e. g., on a high level, we might decide to generate CPU or accelerator code automatically depending on the performance numbers provided in the TPDL. On a low level, we may apply vectorization to the code. Furthermore, communication patterns can be chosen automatically by the compiler in correspondence to the hardware and algorithm. Of great interest to us is polyhedral optimization [10]. We see a high potential to apply and to benefit from such transformations in the compilation process. However, more investigation is needed to find out the best points of entry for such techniques.

For the overall framework, increasing transformation performance especially for large DSL programs is a goal not only benefiting users, but also enabling us as developers to do more comprehensive and thorough evaluations of the code generation path as well as the generated program's performance characteristics. This can be used as input for the aforementioned feedback loop to increase future program generation. Another task is the implementation of heuristics to find performance-

optimal program configurations in this huge design space stemming from the vast degrees of freedom.

Language specifications of the higher—i. e., more abstract—layers of our multi-layered DSL need to be finalized. Research in this field centers around the main language concepts and elements to express the desired functionality. Program transformations between layers are an interesting research topic directly resulting from this task.

IX. Conclusions

In this work, we have introduced ExaSlang, a domain-specific language for solving partial differential equations utilizing multigrid methods. We introduced key concepts and elements of our language. An overview of our framework for transformation, optimization and code generation was given. To demonstrate our approach's effectiveness, we provided scalability results from evaluation runs on up to the full machine JUQUEEN, i. e., 28,672 nodes, equivalent to 458,752 cores.

Furthermore, we demonstrate that DSLs are not only applicable in the field of single machines, i. e., shared-memory systems or single accelerators, but enable the use of modern HPC systems by domain experts without the need for a interdisciplinary team of scientists to implement and tune the simulation's program code for months.

X. Acknowledgments

This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 "Software for Exascale Computing" in project under contracts TE 163/17-1 and RU 422/15-1. We thank the Jülich Supercomputing Center for providing access to the supercomputer JUQUEEN. Finally, we thank Christian Lengauer for his valuable contributions and inspiring discussions.

References

[1] A. Baker, R. Falgout, T. Kolev, and U. Yang. "Scaling hypre's multigrid solvers to 100,000 cores." In: *High-Performance Scientific Computing*. Springer, 2012, pp. 261–279.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. "A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE." In: *Computing* 82 (2 2008), pp. 121–138.

[3] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. "Julia: A fast dynamic language for technical computing." In: *The Computing Research Repository (CoRR)* (2012), 27 pp. arXiv: 1209.5145.

[4] 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. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2011, pp. 89–100.

[5] H. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl. "The PDE framework Peano applied to fluid dynamics: An efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids." In: *Computational Mechanics* 46.1 (2010), pp. 103–114.

[6] M. Christen, O. Schenk, and H. Burkhart. "PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures." In: *Proc. IEEE Int. Parallel & Distributed Processing Symp. (IPDPS)*. IEEE. 2011, pp. 676–687.

[7] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. "Terra: A multi-stage language for high-performance computing." In: *ACM SIGPLAN Notices*. Vol. 48. 6. ACM. 2013, pp. 105–116.

[8] Z. DeVito et al. "Liszt: A domain specific language for building portable mesh-based PDE solvers." In: *Proc. Conf. on High Performance Computing Networking, Storage and Analysis (SC)*. Paper 9, 12 pp. ACM, 2011.

[9] R. Falgout, V. Henson, J. Jones, and U. Yang. *Boomer AMG: A Parallel Implementation of Algebraic Multigrid*. Tech. rep. UCRL-MI-133583. Lawrence Livermore National Laboratory, 1999.

[10] P. Feautrier and C. Lengauer. "Polyhedron model." In: *Encyclopedia of Parallel Computing*. Ed. by D. A. Padua. Springer, 2011, pp. 1581–1592.

[11] M. Frigo and S. G. Johnson. "The design and implementation of FFTW3." In: *Proc. IEEE* 93.2 (Feb. 2005), pp. 216–231.

[12] H. Köstler, C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, and U. Rüde. "A Scala prototype to generate multigrid solver implementations for different problems and target multi-core platforms." In: *The Computing Research Repository (CoRR)* (June 20, 2014), 18 pp. arXiv: 1406.5369.

[13] S. Kuckuk, B. Gmeiner, H. Köstler, and U. Rüde. "A generic prototype to benchmark algorithms and data structures for hierarchical hybrid grids." In: *Proc. Int. Conf. on Parallel Computing (ParCo)*. IOS Press, 2013, pp. 813–822.

[14] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis and transformation." In: *Proc. Int. Symp. on Code Generation and Optimization*. (San Jose, CA, USA). Mar. 2004, pp. 75–88.

[15] C. Lengauer et al. *ExaStencils: Advanced Stencil-Code Engineering – First Project Report*. Tech. rep. MIP-1401. Department of Computer Science and Mathematics, University of Passau, June 2014.

[16] 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. Springer, 2012, pp. 1–723.

[17] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. "Generating device-specific GPU code for local operators in medical imaging." In: *Proc. IEEE Int. Parallel & Distributed Processing Symposium (IPDPS)*. IEEE. Shanghai, China, May 21–25, 2012, pp. 569–581.

[18] 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: *Journal of Parallel and Distributed Computing* (Nov. 2014), 32 pp.

[19] M. Mernik, J. Heering, and A. M. Sloane. "When and how to develop domain-specific languages." In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344.

[20] M. Odersky et al. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. EPFL Lausanne, Switzerland, 2004, pp. 1–20.

[21] C. Osuna, O. Fuhrer, T. Gysi, and M. Bianco. "STELLA: A domain-specific language for stencil methods on structured grids." In: Poster Presentation at the Platform for Advanced Scientific Computing (PASC) Conference, Zurich, Switzerland. June 2–3, 2014.

[22] M. Püschel, F. Franchetti, and Y. Voronenko. "SPIRAL." In: *Encyclopedia of Parallel Computing*. Ed. by D. A. Padua. Springer, 2011, pp. 1920–1933.

[23] D. Quinlan and C. Liao. "The rose source-to-source compiler infrastructure." In: *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*. Vol. 2011. ACM, 2011.

[24] 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. Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. Nov. 2012, pp. 1116–1123.

[25] M. Schmid, O. Reiche, C. Schmitt, F. Hannig, and J. Teich. "Code generation for high-level synthesis of multiresolution applications on fpgas." In: *Proceedings of the First International Workshop on FPGAs for Software Programmers (FSP)*. (Munich, Germany). Sept. 1, 2014, pp. 21–26. arXiv: 1408.4721.

[26] C. Schmitt, S. Kuckuk, H. Köstler, F. Hannig, and J. Teich. "An evaluation of domain-specific language technologies for code generation." In: *Proc. Int. Conf. on Computational Science and its Applications (ICCSA)*. (Guimaraes, Portugal). IEEE Computer Society, June 30–July 3, 2014, pp. 18–26.

[27] T. Sloane. "Experiences with domain-specific language embedding in Scala." In: *Domain-Specific Program Development* (2008), pp. 1–6.

[28] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. "The Pochoir stencil compiler." In: *Proc. 23rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2011, pp. 117–128.

[29] D. Unat, X. Cai, and S. B. Baden. "Mint: Realizing CUDA performance in 3d stencil methods with annotated C." In: *Proc. Int. Conf. on Supercomputing (ISC)*. ACM. 2011, pp. 214–224.

[30] R. C. Whaley, A. Petitet, and J. J. Dongarra. "Automated empirical optimization of software and the ATLAS project." In: *Parallel Computing* 27.1 (2001), pp. 3–35.