Reconfigurable Hardware Generation of Multigrid Solvers with Conjugate Gradient Coarse-Grid Solution

CHRISTIAN SCHMITT, MORITZ SCHMID, SEBASTIAN KUCKUK,
HARALD KÖSTLER, JÜRGEN TEICH, and FRANK HANNIG

Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany

ABSTRACT

Not only in the field of high-performance computing (HPC), field programmable gate arrays (FPGAs) are a soaringly popular accelerator technology. However, they use a completely different programming paradigm and tool set compared to central processing units (CPUs) or even graphics processing units (GPUs), adding extra development steps and requiring special knowledge, hindering widespread use in scientific computing. To bridge this programmability gap, domain-specific languages (DSLs) are a popular choice to generate low-level implementations from an abstract algorithm description. In this work, we demonstrate our approach for the generation of numerical solver implementations based on the multigrid method for FPGAs from the same code base that is also used to generate code for CPUs using a hybrid parallelization of MPI and OpenMP. Our approach yields in a hardware design that can compute up to 11 V-cycles per second with an input grid size of 4096×4096 and solution on the coarsest using the conjugate gradient (CG) method on a mid-range FPGA, beating vectorized, multi-threaded execution on an Intel Xeon processor.

Keywords: Domain-specific language, Multigrid, Conjugate gradient, Code generation, FPGA, Accelerator, High-level synthesis, Pipeline, Stencil.

1. Introduction

Already today, a large percentage of clusters and supercomputers is equipped with accelerators and we expect that, in order to achieve exascale performance, the use of accelerator technologies will not only intensify, but will lead to a variety of new and different technologies, resulting in systems that are equipped with numerous accelerator technologies at the same time. However, the implementation of numerical solvers that unleash such a machine's full potential poses a great challenge, even for programming experts. They would not only require excellent knowledge of the different technologies but also of the mathematical and algorithmic implementation details.

A common solution to this challenge are domain-specific languages (DSLs). They decouple the algorithm from its implementation and allow domain experts, who not necessarily are programming experts, to formulate a description of the problem using concepts and terms native to them. This description then is transformed into a (binary) program by a DSL compiler. To add new optimizations, e.g., support for upcoming accelerator technologies, only the compiler has to be extended. Then, it

merely takes a recompilation step of the original DSL program to benefit from the compiler's improvements, where in traditional approaches the program has to be extended or often even re-written from scratch in order to be efficiently parallelized and optimized towards the specifics of a novel architecture.

This approach is used by project ExaStencils [1], which researches the feasibility of generating highly scalable numerical solvers based on the multigrid method [2, 3]. Multigrid methods are known to be one of the most efficient ways to solve systems of equations that arise from the discretization of elliptic partial differential equations (PDEs), which occur constantly in many application domains, such as physics, chemistry or materials science. ExaStencils proposes a multi-layered DSL with each layer tailored towards a certain user group. Furthermore, this approach—in combination with a description of the target platform and profound built-in domain knowledge—enables a great variety of possible optimizations that can be done automatically by the DSL compiler.

FPGAs have been a popular choice for the implementation of signal processing for a long time. Due to their high computational power in combination with excellent energy efficiency, they are increasingly drawing interest from users of other domains. Furthermore, newer approaches to supersede traditional hand-coding of register transfer level (RTL) designs have been matured: high-level synthesis (HLS) frameworks often provide an equal quality of results while achieving significant productivity gains through generating synthesizable hardware descriptions from behavioral algorithm descriptions on a higher abstraction level, e.g., C code. However, describing algorithms using such HLS-specific C code is still very specific towards a certain implementation, whereas DSLs allow to formulate algorithms in a much more abstract manner and thus enable even higher productivity improvements. Regardless of its development, the end product of such a hardware development is synthesized into a so-called intellectual property (IP) core, which then can be loaded onto a FPGA or integrated as part of an application specific integrated circuit (ASIC).

In this work, we demonstrate the feasibility of generating geometric multigrid solvers with solution at the coarsest grid via the conjugate gradient (CG) method from a domain-specific language. We substantiate our claims by providing results of generated solvers for the steady-state heat equation using constant and variable coefficients on an FPGA board. For our case study, a fine-grid size of 4096×4096 and a recursion depth of 8 was used. We compare the achieved performance to results from an Intel Xeon processor. Regardless of the fact that code for the reconfigurable systems has been processed by Vivado HLS, the presented approach is applicable to any C-based high-level synthesis in general.

The rest of this work is structured as follows: In Section 2, related work is reviewed. In Section 3, a brief introduction to multigrid methods is given. We provide a brief overview of our DSL and present its programming model in Section 4. In Section 5, the challenges and solutions arising from the shift towards code generation for FPGAs are expounded, whereas in Section 6 evaluation results of the actual hardware implementation are presented. Lastly, conclusions of the presented research are drawn in Section 7.

2. Related Work

Regarding the utilization of reconfigurable hardware in scientific computing, a number of different approaches have been evaluated. One example is Gu [4], who researched the acceleration of molecular dynamics simulations on FPGAs. Another example is Hu [5], who examined, among other numerical algorithms, the conjugate gradient (CG) method on reconfigurable hardware. The solution of Poisson's equation on a hybrid reconfigurable system consisting of a CPU and an FPGA is described by Gomes et al. [6].

For the development of applications on FPGAs, HLS is a popular choice and thus, numerous solutions have been developed. A popular approach is to start from a simple imperative programming language, e.g., a subset of C, and then translate it by stepwise refinement into a synthesizable hardware description language (HDL). Commercial examples include, besides the aforementioned Vivado HLS by Xilinx which is an advancement of the acquired HLS tool AutoPilot [7], Catapult[a] from Mentor Graphics, Cynthesizer [8] from Cadence (before Forte), NEC's CyberWorkbench [9], and Synopsys' Synphony C Compiler (formerly PICO Express [10] by Synfora).

For specific application fields, programming aids in the form of libraries are available and often are shipped with HLS frameworks. For the domain of image processing, a partial port of the computer vision library OpenCV is shipped with Vivado HLS[b].

However, extending such a library can become quite a burden and poses problems when porting to new hardware. In contrast, DSL-based approaches separate algorithms from their implementation and provide greater flexibility by allowing easier extension to new platforms. PARO [11], for instance, is a HLS environment for the domain of image processing and provides domain-specific augmentations for border treatment and reductions such as median filtering. It is also capable of adaptive multiresolution filtering in medical imaging [12].

In previous work, the benefits of domain-specific optimization have been shown in various domains. *SPIRAL* [13], 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). In case of hardware generation, soft IP cores in synthesizable RTL Verilog are emitted. *ATLAS* [14] and *FFTW* [15] are examples for the generation of mathematical code from abstract descriptions for specific applications such as FFTs, where further optimizations are selected via auto-tuning.

In the field of scientific computing and especially for stencil computations, numerous approaches building upon domain-specific languages and code generation exist. Examples include *Liszt* [16], which adds abstractions to Java to ease stencil computations for unstructured problems, and *Pochoir* [17], which employs a

[a] https://www.mentor.com/hls-lp/catapult-high-level-synthesis/
[b] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

divide-and-conquer skeleton on top of the parallel C extension Cilk to make stencil computations cache-oblivious. *PATUS* [18] uses auto-tuning techniques to improve performance. However, none of the aforementioned approaches support code generation for FPGAs.

Computations in image processing often are similar to stencil computations and also another popular area for DSLs. *Halide* [19] generates, among others, CUDA and OpenCL code from a DSL embedded into C++. The same description can be transformed to Verilog by *Darkroom* [20].

HIPAcc [21] provides native support for multigrid methods by offering appropriate language elements [22]. It generates low-level accelerator such as CUDA, OpenCL and Renderscript code from a DSL embedded into C++ and was recently extended to emit code that can be processed to IP cores using Vivado HLS [23].

3. Multigrid Methods

In scientific computing, multigrid methods are a popular choice for the solution of large systems of linear equations that may stem from the discretization of partial differential equations (PDEs). The fundamental idea behind multigrid algorithms is to use a hierarchy of discretizations to accelerate the convergence of a basic iterative method by correcting the solution approximation on the fine grid globally, produced by the solution on the coarse grid. Since the iterative method reduces high-frequency errors, it is also called smoother or relaxer. The coarse problem is similar to the one on the finer grid, but consists of a smaller number of unknowns and thus is cheaper to solve. Furthermore, previously low-frequency errors are now high-frequency errors that are relaxed by the iterative method. The solution approximation can be improved by again going to a coarser grid, exhibiting the recursive nature of the multigrid algorithm.

One instance of a multigrid method to solve a simple equation such as $-\Delta u = f$ is shown in Figure 1, where u is the unknown to be solved for and f is the equation's right-hand side. The first step is to discretize the continuous computational domain and the continuous equation using, for example, the finite differences method. We obtain a grid with physical coordinates corresponding to the computational domain's coordinates and a discretized version of our equation, such as $-A_h u_h = f_h$. Here, index h represents the finer grid, whereas H denotes the coarser grid. A_h and A_H are used to describe the discretized mathematical operator, e.g., the Laplace operator. On the top-most multigrid level, i.e., the finest grid, u_h denotes the unknown function. Respectively, on the other multigrid levels, it denotes the current approximation. The right-hand side f is described by f_h. The parameter γ determines the number of recursive steps. $\gamma = 1$ results in a so-called V-cycle, i.e., approximation on the coarser grid is calculated only once. By setting $\gamma = 2$, a W-cycle is implemented, resulting in a better convergence albeit at a higher computational cost. In the pre- and post-smoothing steps, high-frequency components of the error are damped by smoothers such as the *Jacobi* or the *Gauss-Seidel* methods. In the algorithm, ν_1 and ν_2 denote the number of smoothing steps that are applied.

On the coarsest level, solving the remaining linear system of equations approximately by a specialized solver—or even directly—is more efficient than employing further levels of recursion due to its low number of unknowns. A widely used algorithm for this purpose is the CG method. However, it is also possible to apply a number of smoother iterations. In the case of a single unknown, one smoother iteration corresponds to solving directly.

if coarsest level **then**
  solve $A_h u_h = f_h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} \leftarrow \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A_h, f_h\right)$ ▷ 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**
    $e_H^{(j)} \leftarrow \mathrm{MG}_H\left(e_H^{(j-1)}, A_H, r_H, \gamma, \nu_1, \nu_2\right)$ ▷ recursion
  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
end if

Fig. 1. Recursive Multigrid algorithm to compute $u_h^{(k+1)} = \mathrm{MG}_h\left(u_h^{(k)}, A_h, f_h, \gamma, \nu_1, \nu_2\right)$.

4. Programming Model

ExaStencils is a project researching the generation of efficient and scalable numerical solvers based on multigrid methods from a description of the problem formulated in a domain-specific language. During the translation process, domain-specific and hardware-specific optimizations are applied to generate high-performance, scalable C++ code.

ExaSlang, short for ExaStencils language, is a domain-specific language consisting of four layers of abstraction, geared towards different classes of users from diverse domains. ExaSlang 4 constitutes the most concrete layer and allows to specify standard and custom multigrid cycles. It is a procedural and statically typed programming language, featuring control structures such as functions, loops and conditions. Furthermore, this layer is explicitly parallel by providing very simple communication statements to specify data to be communicated. A more thorough description of the language and its code generation and transformation framework can be found in [24, 25] and in [26].

4.1. *Language Elements*

As ExaSlang targets the description of multigrid-based numerical solvers, it combines elements of procedural languages, such as functions and loops, with domain-specific elements, e.g., stencils and communication-enabled memory arrays.

A key element and language feature directly stemming from the focus on multigrid methods are level specifications. Essentially, they are a suffix to identifiers such as function and variable names, tying that entity to one or more specific multigrid levels. A common usage pattern is to exit the multigrid recursion, as depicted in Figure 2. Here, the identifier `@all` in line 1 defines the `VCyle` function on every multigrid level, while the more precise specification using `@coarsest` in line 14 overwrites that function definition on the bottom multigrid level. The recursive call for the V-cycle on the next grid level is represented by the function call `VCycle@coarser ()` in line 8. Ultimately, in case of the penultimate multigrid level, this `@coarser` specification will resolve to the coarsest level, calling the specialized function that ends the recursion. Similarly, the `@current` specifications in lines 3 and 11 resolve to the specific multigrid level, calling the smoother functions for the appropriate multigrid level, and thus, grid sizes.

```
1  Function VCycle@all () : Unit {
2    repeat 3 times {
3      Smoother@current ()
4    }
5    UpResidual@current ()
6    Restriction@current ()
7    SetSolution@coarser (0)
8    VCycle@coarser ()  // recursive call
9    Correction@current ()
10   repeat 3 times {
11     Smoother@current ()
12   }
13 }
14 Function VCycle@coarsest () : Unit {
15   /* apply specialized solve, no recursion */
16 }
```

Fig. 2. Implementation of a V(3,3)-cycle in ExaSlang 4.

ExaSlang 4 features special data types such as **`Stencil`** and **`Field`**. We call this group algorithmic data types, as they can only be used in numerical calculations. In the next paragraph, we will introduce the two major data types.

Stencils are defined by listing the weights around the center using relative addressing. Weights can be represented by any type of expression, including binary expressions and function calls, but, naturally, constant values are possible as well. An example can be seen in Figure 3, where a weighted two-dimensional Jacobi smoother with constant coefficients is described. As the weight expressions include calls to the

built-in functions `meshWidth_x` and `meshWidth_y` (compare lines 2 to 7) to access distance between two grid points, suffixed with the level specification `@current`, in fact the stencil has different weights on every level, although it has been defined only once. In line 14, a built-in function named `diag` is called to extract the stencil's center weight. As a stencil represents a matrix, the matrix' diagonal corresponds to the central weights of every stencil, hence the function name.

```
1  Stencil Laplace@all {
2      [ 0,  0] => ((  2.0 / ( meshWidth_x@current() ** 2 )
3                   + 2.0 / ( meshWidth_y@current() ** 2 )))
4      [ 1,  0] => (( -1.0 / ( meshWidth_x@current() ** 2 )))
5      [-1,  0] => (( -1.0 / ( meshWidth_x@current() ** 2 )))
6      [ 0,  1] => (( -1.0 / ( meshWidth_y@current() ** 2 )))
7      [ 0, -1] => (( -1.0 / ( meshWidth_y@current() ** 2 )))
8  }
9
10 Function Smoother@all() : Unit {
11   loop over Solution@current {
12     Solution[next]@current = (1.0 - omega)
13       * Solution[active]@current
14       + ((( omega / diag(Laplace@current)))
15          * (RHS@current
16             - Laplace@current * Solution[active]@current))
17   }
18   advance Solution@current
19 }
```

Fig. 3. ExaSlang 4 example of a 2D stencil definition and its use as part of a weighted Jacobi smoother.

Fields represent data, e.g., the unknown to be solved for or the right-hand side. To define a field, first a layout is needed which essentially specifies all options for parallelization efforts, such as the size of ghost (halo) layers, the location (e.g., cell-based or node-based), or the data type. Fields then apply the given layout to the computational domain which is also specified in ExaSlang 4. Additionally, boundary conditions need to be specified. Both layouts and fields can be defined using level specifications, allowing for different options on different levels. A common use-case is the definition of different communication patterns and different boundary conditions on certain levels.

Slots are another feature of fields, allowing to define multiple copies of the same field. This is often used when implementing Jacobi solvers, where iterations alternate between two data grids. In Figure 3, their use can be seen in line 13 and following, where accesses to `Solution` have a suffix of `[active]` or `[next]`. The currently used slot may be shifted, similar to a ring-buffer, by using the **`advance`** statement, as depicted in line 18.

A language element crucial to this domain is the `loop over` statement, as used in Figure 3 in line 11. It is used to instantiate an iteration over the computational domain (or a part of it) by specifying a field that is used to determine the computational bounds.

4.2. *Code Generation*

Multigrid algorithms described in ExaSlang 4 are transformed into C++ code by a transformation framework written in Scala. The input file is parsed and transformed into an abstract syntax tree (AST), to which target platform specific alterations and optimizations are applied. We call this stage, where most of the transformations take place, the intermediate representation (IR), since it is only available in the compiler instance. After numerous alterations, the IR is emitted as C++ code that can be compiled with standard compilers such as gcc, Clang, IBM XL and MSVC.

5. Mapping to Hardware

To generate C++ code that can be mapped and synthesized efficiently to a hardware architecture for FPGAs, the transformation chain has to respect a number of specifics.

5.1. *Computational Model*

The conventional way to stencil computations is based on the temporal computing paradigm, i.e., to allocate a continuous chunk of memory and apply the stencils by iterating sequentially over the memory, i.e., a multigrid algorithm is realized as a sequence of smoother, residual calculation, restriction and prolongation stencils. Usually, these kernels are executed linearly, where application of a new kernel starts only after completion of the previous one. As a consequence, to improve the overall performance, kernel execution times have to be reduced.

In contrast, FPGAs offer a massively parallel hardware architecture which can achieve the best results in combination with data streaming. The concept of implementing the multigrid algorithm as a sequence of stencils can be carried over to the FPGA architecture by converting the computational kernels into hardware modules and laying them out in parallel on the chip. The modules are then interconnected by data streams to form a pipeline, through which data is streamed from one entity to another. Once the pipeline is completely filled, all of the computations are carried out in parallel, providing a continuous output flow of results from a continuous delivery of input data. Therefore, this paradigm is called spatial computing.

A key concept in hardware development is to design a component once and replicate it as often as necessary. For multigrid algorithms, we can make use of this principle by designing one stage of the algorithm and replicate it to implement the recursion levels. An important fact to consider is that the lower stages always only have to process a fraction of the data of the next higher stage, e.g., a quarter in the case of 2D. Although this could be exploited in hardware by lowering the clock frequency of the lower stages, a much more sophisticated approach is to increase the

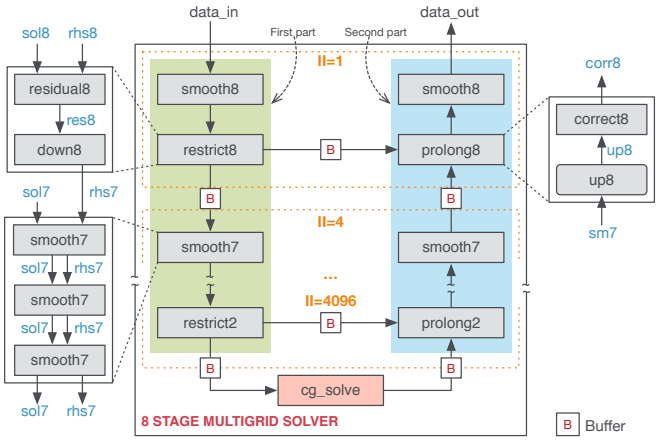

Fig. 4. Structural representation of the multigrid algorithm implementation.

pipeline interval, also often called Iteration Interval (II), which describes the amount of clock cycles between the arrival of new data elements. To obtain high performance, the topmost level uses a pipeline interval of one, which means the architecture can accept new input data in every clock cycle. In consequence, it also produces results in every clock cycle, after a certain latency. To achieve this, however, each operation such as addition or multiplication of the algorithm must be mapped to a dedicated operator in hardware, and therefore the implementation requires a large amount of hardware resources. If the pipeline interval is increased on the lower levels, hardware operators can be shared among the operations, which leads to significantly lower resource requirements. An overview of the structure for a multigrid solver is shown in Figure 4.

In addition to the actual implementation of each stage, the figure also shows the II for the stages, data streams, and indicates which connections require buffering (depicted as B). Although it is possible to instantiate buffers on every stream, there are actually only three cases where the interconnection requires buffering. These are

(1) after down samplers (part of restrict),
(2) before up samplers (part of prolong),
(3) before nodes that combine data streams and have different path lengths.

The necessity for buffering after downsampling and before up sampling is due to different iteration intervals between the stages. The buffer requirement for combining nodes, such as the correction step of the prolongation, becomes evident from the structure of the accelerator. For example, the connection between restriction and prolongation requires a very large buffer, since prolongation must wait until data arrives after having traversed all of the lower stages. Other interconnections in the architecture can be set to simple registered handshake connections, which will

lower the total amount of hardware resources required for buffer implementation. A limitation of the current HLS approach is that the re-use of streams is problematic. In case a data stream is needed as input for multiple kernels, e.g., the result of the residual computation is used for downsampling and for correction, it has to be duplicated accordingly by inserting kernels that copy, or split, the stream into the required number of output streams. Currently, HLS tools do not automatically duplicate such streams or insert appropriate copy operations. Thus, the required split kernels need to be identified and placed into the pipeline at the correct positions. Proceeding in this section, we will explain how code generation must be altered to achieve efficient hardware accelerators by HLS.

5.2. *Stencils and Kernels*

As already explained, computations are implemented by kernels and connected via streams that represent input and output elements. In previous work, we have introduced a library of standard components to facilitate code generation for C-based HLS [27]. In addition to support point and local operators with an arbitrary number of input and output ports, the library also provides operations to handle data streams in complex pipelines. Stencils can be expressed by local operators, where the center corresponds to the output element being calculated and neighboring points are addressed in a relative manner, similar to the way stencils are defined in ExaSlang 4.

As a consequence of the shift towards the stream processing model, iterations over the computational domain need to be transformed into separate kernels. As an iteration is declared by the `loop over` statement and all computational domain sizes are known at compile time, a corresponding kernel function with the correct number of stream elements can be derived directly and transformed into a computational kernel. The original loop statement is replaced with an instantiation and call of the kernel. Vivado HLS synthesizes each instantiated kernel into a dedicated hardware module and generates the data streaming interconnect fabric.

However, Vivado HLS currently is unable to connect the separate kernels to build up the data flow necessary for algorithm implementation. Therefore, the kernels need to be connected by the code generator by the memory streams. As per Vivado HLS' philosophy, a memory stream is consumed as soon as it used as input for a kernel and cannot be used multiple times. Consequently, to enable stream re-use, so-called copy kernels (also split kernels) have to be introduced. They multiply one stream into multiple identical output streams which then can be used as input into the corresponding kernels. This approach is used in numerous places throughout the multigrid solver implementation, e.g., to feed the right-hand-side stream into the six smoother kernels per multigrid level. To add these copy kernels automatically, a corresponding dependency analysis has been developed as part of our code generator.

For the implementation of the CG method used for coarse-grid solving, on-chip block RAM (BRAM) is used, which easily can hold the coarsest grid size of 32×32 values also in double precision. However, the implemented CG method for coarse-grid solution comes with its own drawbacks: executing a high number of iterations would

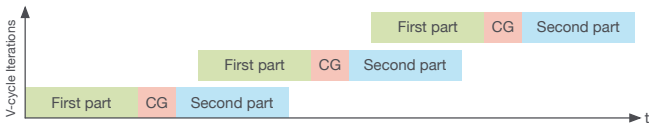

Fig. 5. Schematic illustration of overlapping multiple V-cycle iterations.

result in an an enormous increase in latency, as the first half of the pipeline must be fully executed in order to obtain all of the input data for the CG solver and the second part of the pipeline can only start once the CG has finished all of its iterations. As the key advantage of FPGAs for multigrid is that once the pipeline is filled, all of the kernels can run in parallel, involving an iterative solver at the coarsest level will effectively more than double the execution time. Furthermore, all of the data computed in the first half of the pipeline must be retained while the CG is running until it can be consumed. Since the second part of the pipeline is idle while the first part is computed and vice versa, it is possible to overlap successive V-cycle iterations, as shown in Figure 5. Here, the first part of the pipeline can be overlapped with the second part as soon as the second part starts producing output data which is fed straight back into the input of the first part.

In contrast, when applying a number of smoothing steps to solve on the coarsest level, the fully pipelined execution model can continue and it is possible to start the second half of the pipeline before the first part has finished its computations. Yet, this approach usually results in a worse overall convergence rate, as most often the number of smoother iterations applied is smaller than the number needed to reach the same accuracy as the CG method. Consequently, this increases the number of V-cycles needed to solve the system of equations to the same precision by multiple orders of magnitude. Actually applying the smoother as often as needed for direct solution is impossible, as there simply are not enough resources available to do so on FPGAs, thus a specialized coarse-grid solver must be used.

During transformation of the DSL stencil calculations, stencil weights are resolved directly into the calculations, and thus, mathematical expressions can be statically evaluated and simplified. For CPU code, this reduces memory accesses and enables further optimizations. The same approach is used for the HLS code generation, where placing the coefficients directly into the code yields a lower number of memory streams that need to be processed, i.e., streams that only provide constants are not generated.

5.3. *Loops and Recursion*

To enable pipelining and parallel execution of the kernels, the loop and recursion constructs of ExaSlang 4 must be unrolled and flattened. The principle can be easily applied to the `repeat N times` loop, as N is a constant integer. An example for this is the repeated application of the smoother. Since the number of applications

is known at compile time, we can simply unroll the loop and generate appropriate kernels.

A control structure that requires more attention is recursion. For example, it is used to define the V-cycle (see Figure 1). However, due to ExaSlang's static approach, also this information is available at compile time which enables us to unroll the recursion and instantiate appropriate kernels. In addition, we must adjust the iteration interval and the loop bounds of the individual kernels, instantiate restriction and prolongation operators, as well as duplicate data streams where necessary.

Moreover, ExaSlang contains `repeat until` loops, which are executed until a certain criteria is fulfilled. A common use case for this language construct is the repeated application of the V-cycle until desired precision has been reached. Supporting this use case is difficult, since it would require the repeated sequential execution of the complete process, which would interrupt the streaming pipeline and require extensive buffering of the results at the end of one V-cycle. Of course, this dilemma might be solved if convergence rates were known a priori to starting the V-cycle execution. As local Fourier analysis (LFA), a technique to approximate convergence rates for given equations and solvers, is a crucial part of ExaStencils' approach, we hope to predict the number of V-cycles necessary for solving and, thus, replace the original loop with a fixed number of iterations. Consequently, this loop can be unrolled again. For other use cases, it might be viable to do a dependency analysis to check if the condition can be statically evaluated, in order to again replace the loop with a fixed number of executions to be finally unrolled. In the case that no evaluation can be done, the compilation process currently is aborted, prompting the user to re-write that portion of the program.

6. Case Study

To evaluate our approach, we consider the steady-state heat equation with Dirichlet boundary conditions on the two-dimensional unit square, which is given by

$$\begin{aligned} -\nabla \cdot (a \nabla u) &= f \quad \text{in} \quad \Omega\,, \\ u &= g \quad \text{on} \quad \partial\Omega\,, \\ \Omega &= (0,1)^2 \end{aligned} \tag{1}$$

Here, $a : \mathbb{R}^2 \to \mathbb{R}$ $(a \geq 0)$ describes the material's thermal conductivity in the region. Furthermore, $\nabla \cdot w$ is the divergence of w and ∇u is the gradient of u.

For our numerical experiments, we discretize the heat equation using finite differences and choose f and a from Figure 6. Furthermore, we choose $k = 10$ and $g = 0$ to represent a constant or smoothly changing thermal conductivity. For these examples, an analytic solution can be found which is beneficial for verifying the correctness of an implementation. The analytical solution u is also depicted in Figure 6.

We implemented suitable solvers in ExaSlang 4 using the V-cycle scheme for constant and variable coefficients, both with a recursion depth of 8 and starting

Constant coefficients (2D)

$$f(x,y) = 2k\big((x-x^2)+(y-y^2)\big)$$
$$a(x,y) = 1$$
$$u(x,y) = k(x-x^2)(y-y^2)$$

Variable coefficients (2D)

$$f(x,y) = 2k\big((x-x^2)+(y-y^2)\big)$$
$$a(x,y) = e^{k(x-x^2)(y-y^2)}$$
$$u(x,y) = 1-e^{-k(x-x^2)(y-y^2)}$$

Fig. 6. Family of solutions of Equation (1) for $g = 0$.

on grid sizes of 4096×4096. For smoothing, a weighted Jacobi (JOR) with a precalculated optimal relaxation factor ω is used. The solution on the coarsest grid of size 32×32 is done by a CG solver, also implemented in ExaSlang 4. Because of buffer and memory size constraints on the FPGA, we had to restrict the case study to 2D and to a V(2,2)-cycle for variable coefficients. For constant coefficients, a V(3,3)-cycle was used.

The code generated by ExaSlang 4 was used to infer a hardware description of the multigrid solver using Xilinx Vivado HLS v14.2. As evaluation hardware, we have chosen a Xilinx Virtex 7 (xc7vx485t) FPGA. Although interconnecting individual modules using FIFO streams and executing these concurrently is supported through the `data flow` directive, the tool currently does not determine the buffer sizes automatically, but requires them to be set manually. Since we need to execute the full first part of the pipeline before we can start the CG solver at the coarsest grid, all intermediate values must be retained for the second part of the V-cycle. For the chosen grid size of 4096×4096 floating point values and 8 recursion levels, the buffers on the top four levels become very large and would overwhelm the amount of available resources, even on very large FPGAs. A solution to allow the fastest possible execution and keep within the maximum amount of available resources is to offload the most challenging buffers to external DDR3 memory. A drawback is that HLS tools, such as the here used Vivado HLS, do not support this from within the tool, but require an FPGA support design to facilitate this. We add input and output arguments for the streams to be externalized to the function definition and specify their type as AXI4-Streaming (AXI4S). In this way, we obtain a high-performance interface to the FPGA fabric for each data connection and do not need to make extensive modifications to the actual accelerator source code. The FPGA support design uses an AXI4S interconnect (IC) built on top of a virtual FIFO as an abstraction to an off-chip DDR3 memory. A structural overview of the design is shown in Figure 7. The virtual FIFO is an IP core from Xilinx and can be configured to support up to eight full duplex AXI4S data channels using word widths of up to 128 bytes. The core uses round robin arbitration between the channels which can be weighted in terms of how many data bursts are executed in sequence before arbitrating to the next channel. The size of the data burst defines the

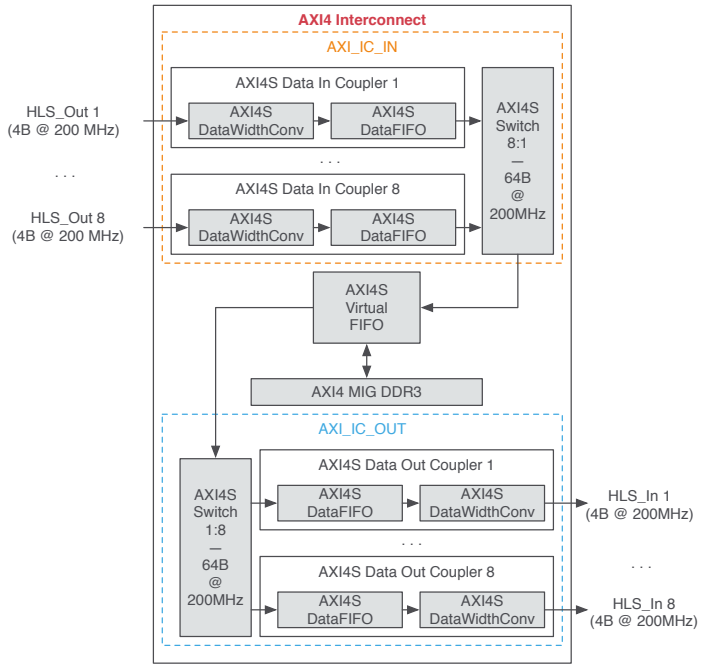

Fig. 7. Structural representation of the FPGA support design. *HLS_Out* and *HLS_In* denote directions coming from respectively routed towards the IP core generated by HLS.

maximum amount of memory space that can be allocated to each channel. For our application, we use a 64 byte word width, as this is also the word width supported by the underlying DDR3 memory and select the smallest available burst size of 128 bytes. As the individual channels have different data production rates, we assign different weights for appropriate bandwidth allocation.

To allow uninterrupted data exchange between the DDR3 and the accelerator, the AXI IC aggregates data to a very large word width of 64 bytes before passing it to the Virtual FIFO. To adjust the data from the accelerator to the requirements for buffering on the off-chip memory, we have designed an AXI4-Streaming interconnect, as shown in Figure 7. It uses an internal 64 byte data bus and is situated in the same clock domain as the memory interface. Incoming data streams are first adjusted to the internal data width before they are transferred to the internal clock domain using asynchronous FIFO buffers. An AXI4S switch merges the data stream onto the interface of the virtual FIFO, which stores the data in the channel's memory region according to the destination identifier of the data stream. An equal path is used in reverse order to transfer data from the external memory back to the output ports of the interconnect and from there to the accelerator.

Figure 8 shows evaluation results of the hardware synthesis from Vivado HLS, which give a rough approximation of the amount of on-chip resources required for

	Single Precision		Double Precision		
Resource	On-Chip	External	On-Chip	External	Available
LUTs	2 153 613	235 054	3 000 651	314 729	303 600
FFs	593 187	247 469	613 870	306 251	607 200
DSPs	1 369	1 369	1 597	1 597	2 800
BRAMs	58 550	982	117 100	1 964	2 060

Fig. 8. HLS resource estimates for the V(3,3)-cycle multigrid solver design using constant coefficients, comparing on-chip and external buffering.

the design. Indeed, after externalizing the top four largest buffers for results and RHS, the design can be fit onto the chip. Note that even though the approximated amount of required LUTs for the double precision version with externalized buffers still exceeds the amount of available resources slightly, the actual implementation process can reduce the resource requirements. Figure 9 lists the post place and route (PPnR) results for single and double precision floating point arithmetic.

Resource	Single Precision		Double Precision	
Constant coefficients				
LUTs	171 274	(56.4 %)	225 095	(74.1 %)
FFs	217 448	(35.8 %)	260 641	(42.9 %)
DSPs	1 369	(48.9 %)	1 597	(57.3 %)
BRAMs	985	(47.8 %)	1 936	(94.0 %)
Slices	66 046	(87.0 %)	69 766	(91.9 %)
F_{max}[MHz]	197.2		140.8	
Variable coefficients				
LUTs	212 774	(70.1 %)	523 105*	(172.3 %)
FFs	225 727	(37.2 %)	479 857*	(79.0 %)
DSPs	1 349	(48.2 %)	4 068*	(145.3 %)
BRAMs	841	(40.9 %)	1 676*	(81.4 %)
Slices	66 790	(87.9 %)	NA	NA
F_{max}[MHz]	193.4		NA	

Fig. 9. PPnR resource requirements of the complete multigrid solver design on the Virtex 7. Values marked with * are estimates by Vivado HLS and could not be synthesized onto the FPGA, as they exceeded the available resources.

Moreover, we have evaluated hardware designs for variable coefficients. Due to the very high resource requirements, however, we were only able to fit a V(2,2)-cycle design using single precision floating point arithmetic on the Virtex 7 FPGA. Figure 9 therefore only lists PPnR results for the single precision version, whereas we can only provide synthesis estimates for the double precision version. Also here, we must

externalize the buffers on the top four levels of the V-cycle to be able to fit the design onto the FPGA. The large difference between both implementations is due to the exponential function required to compute the variable coefficients, which consumes a substantial amount of resources. Another factor that makes the difference in resource usage appear much higher is that the double precision implementation did also not undergo the optimization steps of the actual hardware implementation process, which can deliver noticeable reductions only if the design can also fit on the chosen FPGA.

We have evaluated the PPnR hardware results on the Virtex 7 FPGA to measure the performance in terms of how many clock cycles it takes to process a 4096×4096 grid of floating point values in single and double precision arithmetic. In combination with the clock frequency of the design, this yields an accurate measurement of the performance. Contrasting to a software solution, the pipelining principle also applies here, thus, it is not necessary to wait until the result is ready, but we can start processing a new grid, as soon as all of the input values of the previous grid have been consumed. In addition to waiting until both parts of the pipeline, before and after the CG solver have traversed the V-cycle, it is possible to overlap successive iterations and start a new V-cycle as soon as the second half starts producing output data.

In order to compare the hardware accelerator to state-of-the-art approaches, we have used the specification of the multigrid solver in ExaSlang to generate C++ code for a single machine.

For the CPU tests, an Intel Xeon E5-1620 v3, featuring 4 physical cores resulting in a total of 8 logical cores including Intel's hyperthreading, each clocked at 3.50 GHz, was used. It features 64 KB of L1 and 256 KB of L2 cache per core, and 10 MB of shared L3 cache. The system is equipped with 32 GB of DDR3 RAM. The evaluated code was generated with support for OpenMP parallelization and included vectorization, based on AVX-2. Compilation of the generated C++ code was done using the default gcc 4.7 shipped by openSUSE. For all benchmark runs, the processes were pinned to the CPU cores. Our parallelization concepts on single nodes currently favor numbers of cores that are a power of two, therefore only a single core, two, four and eight cores have been benchmarked.

Figure 10 lists the performance results in terms of latency in milliseconds for processing a single iteration of the V-cycle and the throughput in terms of how many iterations of the V-cycle can be processed per second ([Vps]), on average. It is also worth mentioning that the performance difference between running the V-cycle in single or double precision on the accelerator is not nearly as large as on the CPU. Unfortunately, the Kintex 7 FPGA used in the previous work [28] which is extended in this manuscript was unable to provide the necessary amount of logic and memory resources for the chosen multigrid configuration for the variable coefficients as well as for constant coefficients problem. Even the larger Virtex 7 was unable to provide the necessary amount of resources required for the calculation of the variable coefficients problem in double precision.

Target	Single Precision Runtime [ms]	Single Precision Through-put [Vps]	Double Precision Runtime [ms]	Double Precision Throug-put [Vps]
Constant coefficients				
E5-1620 v3, 1 thread	235.67	4.24	361.34	2.77
E5-1620 v3, 2 thread	121.71	8.21	217.11	4.61
E5-1620 v3, 4 threads	93.61	10.68	176.21	5.68
E5-1620 v3, 8 threads	94.62	10.57	182.54	5.48
Virtex 7, sequential	173.59	5.76	242.69	4.12
Virtex 7, overlapped	91.01	10.99	124.82	8.01
Variable coefficients				
E5-1620 v3, 1 thread	405.72	2.46	590.65	1.69
E5-1620 v3, 2 thread	206.52	4.84	330.61	3.02
E5-1620 v3, 4 threads	151.06	6.62	286.18	3.49
E5-1620 v3, 8 threads	148.81	6.72	287.00	3.48
Virtex 7, sequential	177.04	5.65	NA	NA
Virtex 7, overlapped	90.76	11.02	NA	NA

Fig. 10. Comparison of the performance of the multigrid solvers, i.e., constant and variable coefficients in single and double precision, on different hardware targets.

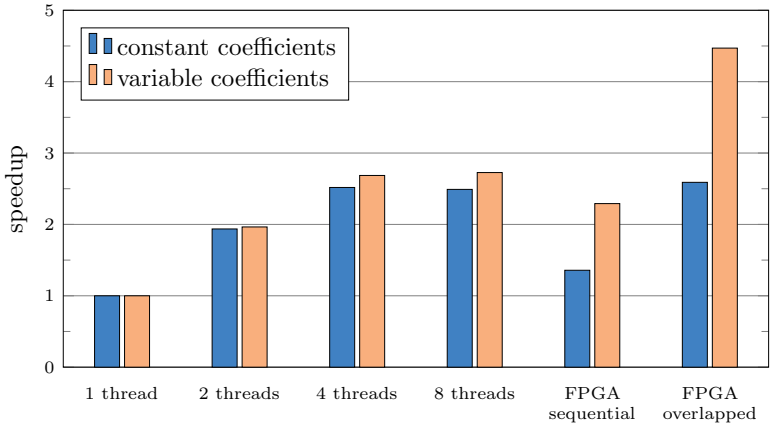

Fig. 11. Comparison of the speedup of the multigrid solvers, i.e., constant and variable coefficients in single precision, on different hardware targets.

7. Conclusions

In this work, we have presented an approach to map descriptions of multigrid algorithms in a domain-specific language to hardware designs for execution on FPGA by generating C++ code that can be used with C-based high-level synthesis

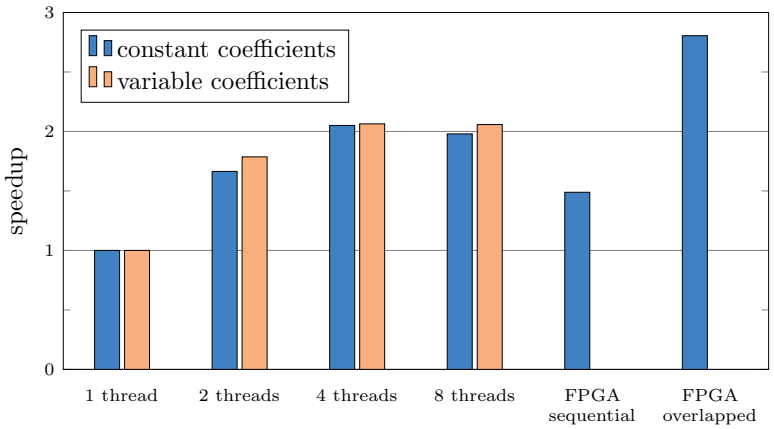

Fig. 12. Comparison of the performance of the multigrid solvers, i.e., constant and variable coefficients in double precision, on different hardware targets.

tools. Furthermore, we have outlined the specifics of implementing stencil-based calculations on FPGAs via HLS tools and highlighted differences from the process of code generation for traditional CPU-based programs. We verified our approach by synthesizing a multigrid-based solver for Poisson's equation including a CG solver for solution on the coarsest grid onto two different hardware targets, a Virtex 7 FPGA and an Intel CPU. Both implementations were generated from the same code base in ExaSlang 4. Additionally, evaluation numbers show that employing FPGAs in scientific computing is a promising approach to increase computing power. Since they are known to be much more energy efficient than CPUs or GPUs, this will also reduce energy footprints.

8. Future Work

While ExaSlang can be used to describe multigrid-based solvers for three and more dimensions and generation of code for HLS for FPGAs works, the underlying concept of streaming and buffering needs some refinement. Already for 2D and single-precision floating point numbers, buffers grow very large, up to the point where all available resources are used. While switching to double precision would consume twice the memory but leave buffer sizes—in terms of elements—intact, adding another dimension would result in enormous buffers. For large datasets in higher dimensions than 2D, current generation FPGAs boards lack sufficient large memories. Instead, data will have to be stored in the host's memory, utilizing the PCI express bus and drawing a huge performance penalty. Nevertheless, we will re-evaluate our case study with newer generations of FPGAs boards.

Additionally, to improve dataset sizes by incorporating on-board DDR3 RAM, memory bandwidths could be improved by employing multiple memory controllers into the design.

Partitioning of data for multiple FPGAs—similar to the way data is partitioned across cluster nodes—is another area worth looking into, especially in the light of HPC, where large datasets are common and different techniques for optimal distribution of data have been developed.

9. 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.

References

[1] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. ExaStencils: Advanced stencil-code engineering. In *Proceedings of Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science (LNCS)*, pages 553–564. Springer, 2014.

[2] Wolfgang Hackbusch. *Multi-Grid Methods and Applications*. Springer-Verlag, 1985.

[3] Ulrich Trottenberg, Cornelis W. Oosterlee, and Anton Schüller. *Multigrid*. Academic Press, 2001.

[4] Yongfeng Gu. *FPGA Acceleration of Molecular Dynamics Simulations*. PhD thesis, Boston University, Boston, MA, USA, 2008.

[5] Jing Hu. *Solution of partial differential equations using reconfigurable computing*. PhD thesis, University of Birmingham, December 2011.

[6] Vitor Gomes, Haroldo Campos Velho, and Andrea Charão. A fast poisson solver for hybrid reconfigurable system. In *Proc. Int. Conf. on Reconfigurable Computing: Architectures, Tools, and Applications (ARC)*, pages 47–58, Berlin, Heidelberg, 2013. Springer-Verlag.

[7] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter 6, pages 99–112. Springer, 2008.

[8] Michael Meredith. *High-Level SystemC Synthesis with Forte's Cynthesizer*, chapter 5. Springer, 2008.

[9] K. Wakabayashi and T. Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 19(12):1507–1522, dec 2000.

[10] Shail Aditya and Vinod Kathail. Algorithmic synthesis using PICO: An integrated framework for application engine synthesis and verification from high level C algorithms. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter 4, pages 53–74. Springer, 2008.

[11] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *Proc. Int. Workshop on Applied Reconfigurable Computing (ARC)*, volume 4943 of *Lecture Notes in Computer Science (LNCS)*, pages 287–293. Springer, 2008.

[12] Frank Hannig, Moritz Schmid, Jürgen Teich, and Heinz Hornegger. A deeply pipelined and parallel architecture for denoising medical images. In *Proc. IEEE Int. Conf. on Field Programmable Technology (FPT)*, pages 485–490. IEEE, 2010.

[13] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. SPIRAL. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1920–1933. Springer, 2011.

[14] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.
[15] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.
[16] Z. DeVito, N. Joubert, F. Palaciosy, S. Oakleyz, M. Medinaz, M. Barrientos, E. Elsenz, F. Hamz, A. Aiken, K. Duraisamy, E. Darvez, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proc. Conf. on High Performance Computing Networking, Storage and Analysis (SC)*. ACM, 2011. Paper 9, 12 pp.
[17] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and Ch. E. Leiserson. The Pochoir stencil compiler. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 117–128. ACM, 2011.
[18] 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)*, pages 676–687. IEEE, 2011.
[19] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)*, 31(4):32:1–32:12, 2012.
[20] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics (TOG)*, 33(4):144:1–144:11, 2014.
[21] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. HIPAcc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, PP(99), 2015.
[22] Richard Membarth, Oliver Reiche, Christian Schmitt, Frank Hannig, Jürgen Teich, Markus Stürmer, and Harald Köstler. Towards a performance-portable description of geometric multigrid algorithms using a domain-specific language. *Journal of Parallel and Distributed Computing (JPDC)*, 2014.
[23] Moritz Schmid, Oliver Reiche, Christian Schmitt, Frank Hannig, and Jürgen Teich. Code generation for high-level synthesis of multiresolution applications on fpgas. In *Proc. Int. Workshop on FPGAs for Software Programmers (FSP)*, pages 21–26, 2014.
[24] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. ExaSlang: A domain-specific language for highly scalable multigrid solvers. In *Proc. Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51. IEEE Computer Society, 2014.
[25] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Jürgen Teich, Harald Köstler, Ulrich Rüde, and Christian Lengauer. Systems of partial differential equations in ExaSlang. In *Software for Exascale Computing – SPPEXA 2013–2015*, volume 113 of *Lecture Notes in Computational Science and Engineering*. Springer, 2016.
[26] Christian Schmitt, Stefan Kronawitter, Frank Hannig, Jürgen Teich, and Christian Lengauer. Automating the development of high-performance multigrid solvers. *Proceedings of the IEEE*, 106(11):1969–1984, 2018.
[27] Moritz Schmid, Nicolas Apelt, Frank Hannig, and Jürgen Teich. An image processing library for C-based high-level synthesis. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2014.
[28] Christian Schmitt, Moritz Schmid, Frank Hannig, Jürgen Teich, Sebastian Kuckuk, and Harald Köstler. Generation of multigrid-based numerical solvers for FPGA accelerators. In *Proc. Int. Workshop on High-Performance Stencil Computations (HiStencils)*, pages 9–15, 2015.