FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Department Informatik

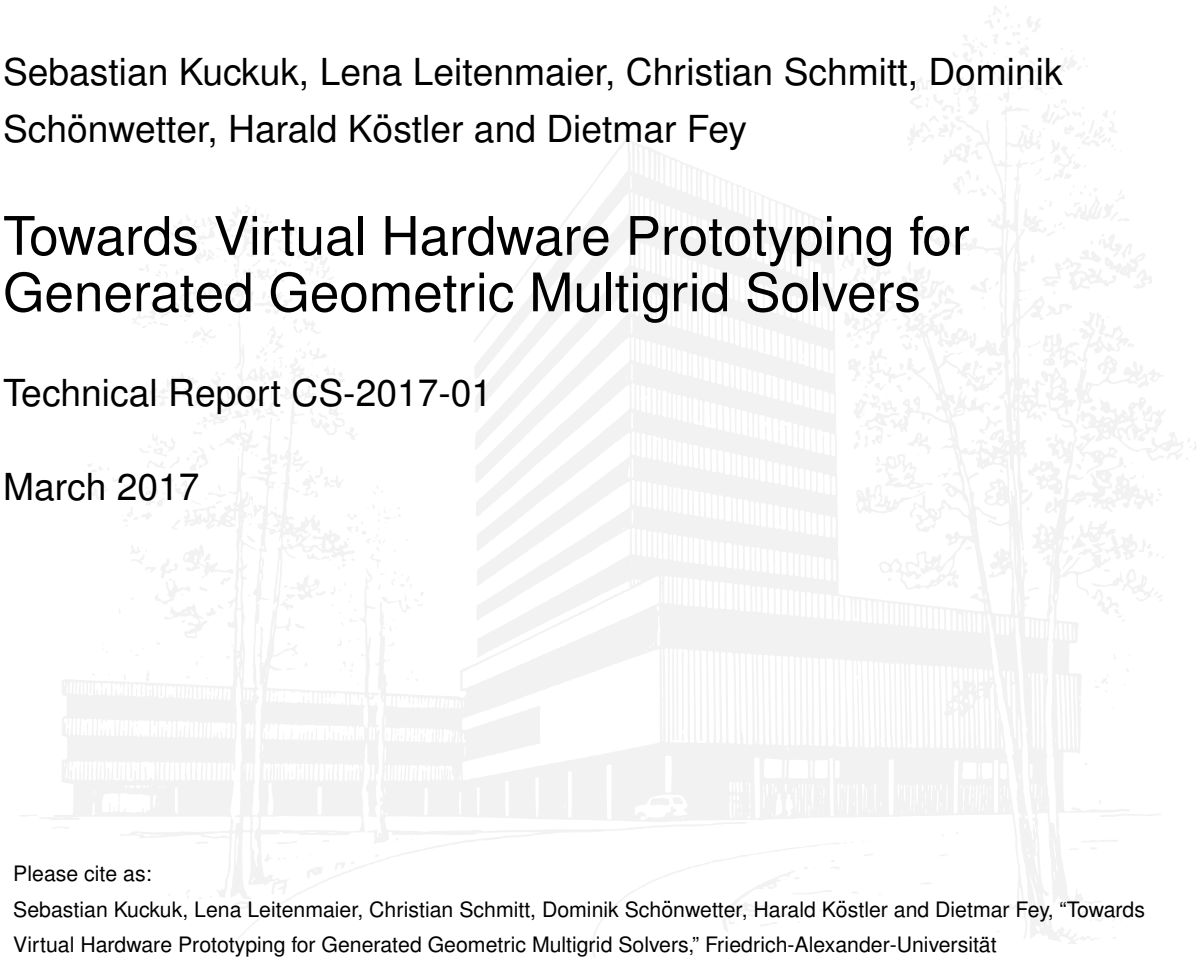
Technical Reports / ISSN 2191-5008

Sebastian Kuckuk, Lena Leitenmaier, Christian Schmitt, Dominik Schönwetter, Harald Köstler and Dietmar Fey

Towards Virtual Hardware Prototyping for Generated Geometric Multigrid Solvers

Technical Report CS-2017-01

March 2017



Please cite as:
Sebastian Kuckuk, Lena Leitenmaier, Christian Schmitt, Dominik Schönwetter, Harald Köstler and Dietmar Fey, “Towards Virtual Hardware Prototyping for Generated Geometric Multigrid Solvers,” Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Technical Reports, CS-2017-01, March 2017.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

Towards Virtual Hardware Prototyping for Generated Geometric Multigrid Solvers

Sebastian Kuckuk, Lena Leitenmaier, Christian Schmitt,
Dominik Schönwetter, Harald Köstler and Dietmar Fey

Dept. of Computer Science, University of Erlangen, Germany
{sebastian.kuckuk, harald.koestler}@fau.de

***Abstract*—Many applications in scientific computing require solving one or more partial differential equations (PDEs). For this task, solvers from the class of multigrid methods are known to be amongst the most efficient. An optimal implementation, however, is highly dependent on the specific problem as well as the target hardware. As energy efficiency is a big topic in today's computing centers, energy-efficient platforms such as ARM-based clusters are actively researched. In this work, we present a domain-specific approach, starting with the problem formulation in a domain-specific language (DSL), down to code generation targeting a variety of systems including embedded architectures. Furthermore, we present an approach to simulate embedded architectures to achieve an optimal hardware/software co-design, i.e., an optimal composition of software and hardware modifications. In this context, we use a virtual environment (OVP) that enables the adaptation of multicore models and their simulation in an efficient way. Our approach shows that execution time prediction for ARM-based platforms is possible and feasible but has to be enhanced with more detailed cache and memory models. We substantiate our claims by providing results for the performance prediction of geometric multigrid solvers generated by the ExaStencils framework.**

I. Introduction

PDEs are omnipresent in many application domains ranging from, e.g., physics, chemistry and biology to material sciences, computational fluid dynamics and electrical engineering. One subclass of these problems are elliptic PDEs, which are usually discretized using finite differences, elements, volumes or some similar method. After the discretization, the arising system of equations needs to be solved. For this task, one of the most efficient ways is employing a solver from the class of multigrid methods. However, the actual algorithmic layout and final implementation are highly specific towards a given problem and target hardware. Beyond the differing implementations, various optimizations, that are usually highly specific to the target platform, are necessary to attain the best performance possible. Consequently, performance portability is highly desirable, but in reality very difficult to achieve. One possibility to tackle this challenge is employing DSLs and code generation techniques. Here, the problem can be specified in an abstract way while a code generator takes care of generating and optimizing solver code. In the domain of geometric multigrid solvers, the DFG-funded ExaStencils project[1] aims at delivering exactly such a tool [12]. Furthermore, ExaStencils envisions the usage of advanced performance prediction techniques to choose performance-optimal parameter configurations [3]. In the beginning of the project, ExaStencils was primarily focused on high performance codes [9], [19], i.e., codes implementing solvers that deliver the shortest time to solution. Usually, this is achieved by targeting large scale clusters such as JUQUEEN located at the Jülich Supercomputing Centre [10]. However, since the importance of energy usage and efficiency is growing steadily, target architectures beyond the classical examples of supercomputers are being focused on more and more. This is also shown in a recent study, which demonstrated the viability and potential of generating code targeting FPGA platforms using the ExaStencils framework [21]. Motivated by this success, we now aim at extending our range of supported architectures even more by moving to ARM processor architectures.

While our code generation techniques aim at generating an optimal solver variant for a given processor architecture, advanced simulation methods yield the potential to do just the opposite, i.e., finding a hardware layout that is optimal in terms of performance or energy consumption for a given algorithm or even a class of algorithms. In particular, virtual hardware prototyping

[1] http://www.exastencils.org

has the potential of determining optimal architectures for certain algorithms by providing a highly configurable virtual hardware. This includes, e.g., the size and hierarchy of caches, or the width of memory interconnects.

Our vision is to combine the two worlds, i.e., that such a hardware which is highly suitable for running geometric multigrid solvers can be conceptualized. Consequently, our code generator needs to be made aware of said model and thus can be enabled to apply unique performance enhancing transformations. As a first step towards this ambitious goal, we examine our frameworks capabilities to deal with ARM architectures as well as the applicability and precision of our simulation approach. We focus on a fast simulation environment to make multicore simulations tangible.

The remainder of the paper is structured as follows: In section II, we give a brief introduction into geometric multigrid methods and present key concepts as well as capabilities of the ExaStencils code generation approach. Next, we describe the environment used for our investigations in section III, consisting of simulation environment OVP as well as the virtual and actual reference hardware. After the two major components have been presented, we describe the test problem used to validate our approach and present results for said problem in section IV. Finally, after briefly looking into similar projects in section V, we conclude and give an outlook towards possible extensions in section VI and section VII, respectively.

II. Generating Solvers with the ExaStencils Approach

A. Multigrid Methods

In the following, we give a short introduction to multigrid methods. For a more in-depth review, we refer to respective literature, e.g., [4], [22].

The two main principles of any multigrid solver are the smoothing property and the coarse grid principle. In detail, classical iterative methods, such as Jacobi or Gauss-Seidel (GS) type solvers, are able to smooth the error within a small number of steps but require a large number of iterations to converge. Concurrently, a smooth function on a given fine grid can be approximated in a satisfactory fashion using only a reduced number of discretization points on a coarser grid. Combining these two ideas into a single iterative solver yields the multigrid algorithm which traverses between fine and coarse grids in a given grid hierarchy.

One possible iteration layout is the so-called v-cycle which is also illustrated in algorithm 1: At the beginning of every step, high-frequency error components are reduced through the application of a small number of a, e.g., Jacobi smoother. After the pre-smoothing, an error approximation, which is usually named residual, is updated and then approximated on a coarser grid in the restriction step. This approximation can then in turn be solved by recursive application of the multigrid algorithm. Returning from the recursive call, the current solution on the fine grid can be corrected with the values from the coarse grid in the correction step. At the end, newly arising high-frequency error components are again smoothed in the post-smoothing step.

if *coarsest level* **then**
  solve $A_h u_h = f_h$ directly or with specialized coarse grid solver
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}
  $r_H = R r_h$ {restriction}
  $u_H^{(k)} = 0$ {initialize solution}
  $e_H = V_H\left(u_H^{(k)}, A_H, r_H, \nu_1, \nu_2\right)$ {recursion}
  $e_h = P e_H$ {prolongation}
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e_h$ {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 $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A_h, f_h, \nu_1, \nu_2\right)$.

B. Domain-specific Modeling and Code Generation

1) ExaSlang: Short for ExaStencils language, it is a DSL for modeling multigrid algorithms [19]. A DSL is a computer language designed towards a certain domain, in our case the domain of multigrid algorithms. Typically, a DSL re-uses the target domain's concepts, abstraction mechanisms and terms by, e.g., providing the corresponding keywords and data types. DSLs can be subdivided into two classes: First, internal (or embedded) DSLs, where the new language is created by extending or restricting an existing language such as Java, C or C++ by introducing or removing keywords and data types. Implementation-wise, usually the language's standard compiler is modified to accept the new elements. Second, external (or standalone) DSLs, which are completely newly designed programming languages. This approach provides greater freedom and flexibility in terms of syntax and semantics for language designers, albeit resulting

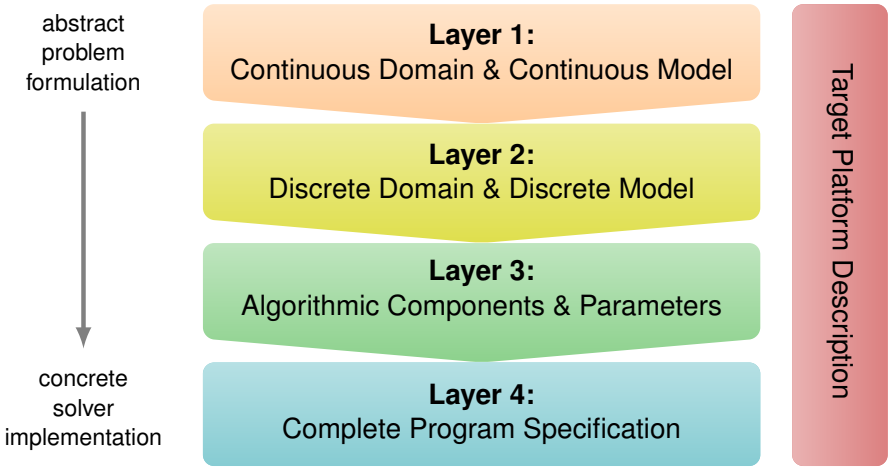


Fig. 1. ExaSlang consists of four layers of abstraction [19]. Orthogonal to the algorithm specification, a description of the target hardware is provided to generate optimal code.

in a larger implementation effort, as compilers have to be built from scratch.

ExaSlang is designed and implemented as an external DSL and features a hierarchical design, with the different layers of abstraction adjusted towards the conception of different classes of users. This structure is illustrated in figure 1. The complete development of an ExaSlang program would start with the formulation of an equation to be solved at ExaSlang 1 (short for ExaSlang Layer 1), followed by an appropriate discretization at ExaSlang 2, followed by the specification of stencils and multigrid components at ExaSlang 3, and concluded with the addition of implementation aspects at ExaSlang 4. Orthogonal to all the algorithmic layers, a description of the target hardware platform is specified to generate hardware-specific and thus efficient code. By design, these language layers are adjusted to the conception of different classes of users. Another benefit from this hierarchical structure is the large design space that is left to explore (and exploit) by ExaStencils' automatic optimization components, by, e.g., selecting a suitable discretization for equations specified at ExaSlang 1 or generating communication patterns at ExaSlang 4 for multigrid components specified in ExaSlang 3. A more thorough overview of ExaSlang and our framework for code generation is given in [19].

2) Code generator: Based on a previous evaluation [20], the ExaStencils code generator has been implemented in Scala. User-specified input in one ExaSlang layer is read using Scala's parser combinator techniques and represented in form of a tree. For each language layer, specialized data structures exist that are instantiated while reading the user input. These data structures than can be progressed, i.e., converted, to the next language layer. ExaSlang 4 data structures are converted into another abstraction layer called intermediate representation (IR), which finally is emitted as C++ code.

Modifications to the program currently generated are done by applying a sequence of transformations. Technically, they employ Scala's powerful pattern matching capabilities to specify code modifications in a functional manner, where a pattern to be found and a corresponding replacement structure may be specified. Albeit each transformation tends to be small and very specialized, they can be aggregated into strategies to perform a certain task in the code generation process. An example is the resolution of expressions into constants, a typical compiler optimization. Mathematical expressions employing (numeric) constants and function calls might be evaluated at compile time. Thereby, first the function calls need to be evaluated. Second, mathematical constants need to be resolved to their numerical values. Finally, the mathematical operators need to be evaluated to replace the expression with its result.

3) Low-level optimizations: Besides generating solvers for a wide range of configurations and problem descriptions, our code generation framework is also able to apply optimization transformations [12]. These transformations allow generated programs to be automatically tuned towards target hardware platforms. Implemented optimizations can be classified into polyhedral and traditional optimizations [8]. The latter include *address pre-calculation* for reducing the number of integer operations carried out in specific kernels, *loop*

unrolling for mitigating the overhead introduced by loops around actual calculations, *arithmetic simplifications* similar to the ones carried out by target compilers, and *vectorization* in order to increase throughput.

For polyhedral optimizations, a model for each program part to be handled is extracted. Usually, this is done at loop level, that is, for each top-level loop a separate representation is set up. Using them, dependencies can be determined easily and various optimization are now possible, such as *merging loops* in order to reduce loop overhead and enable further optimization by, e.g., improving data locality, an extended version of *dead code elimination* to remove unnecessary operations, *tiling* to optimize caching behavior and thus reduce runtime of memory bound kernels, and *finding an optimal schedule* with respect to the detected dependencies by optimizing a given objective function as, e.g., the reuse distance.

III. Environment

The next sections describe our evaluation setup. Starting with an overview about the simulation environment OVP and our actual reference hardware in the first two sections, the last section describes the components of our corresponding virtual hardware.

A. Open Virtual Platforms

The simulation environment and technology from Open Virtual Platforms (OVP) was developed for high-performance simulation of embedded architectures. It permits analysis of virtual platforms containing multiple processor and peripheral models as well as debugging applications running on the virtual hardware. The OVP simulation technology is extensible and provides the ability to create new processor models and other platform components. This is done by writing C or C++ code that uses APIs and libraries supplied as part of OVP [6]. Figure 2 shows an overview about how a basic OVP simulation with one processor works. This example uses the C programming language for defining a hardware platform and for implementing the application to run on that platform.

The OVP simulator is an instruction-accurate simulator which means, that the functionality of a processor's instruction execution is represented without regard to artifacts like pipelining. Consequently, statements about a program's execution time cannot be given with high precision as any conversion to time has limited accuracy compared to actual hardware.

Apart from complete programs, it is also possible to measure how many instructions were executed for isolated code snippets. Assuming a perfect pipeline, where one single instruction, e.g., a floating point operation, is executed per cycle, the total execution time t_r is given by

$$t_r = \frac{\text{instruction count}}{\text{MIPS rate of processor}}, \tag{1}$$

where MIPS is million instructions per second.

In contrast to the single processor variant, OVP multiprocessor simulations are not working concurrently. Here, for efficiency, each processor advances a predefined number of instructions and then signals that it has finished its so-called quantum [6]. As the simulated time is moved forward only at the end of a quantum, simulation artifacts can be introduced. An example for this can be given by a processor spending its time in a wait loop while waiting for the quantum to finish. To avoid such artifacts, the quantum has to be set very low, in the worst case even to one instruction per processor, which will have a significant impact on simulation performance. The quantum can be adjusted in the simulator settings [7].

B. Reference Hardware

As reference hardware we chose an Altera's development kit board [1] with Cyclone V SX SoC-FPGA. It includes a hard processor system (HPS) consisting of a dual-core ARM Cortex-A9 MPCore processor, NEON coprocessor with double-precision FPU, multiport SDRAM controller with support for DDR2 as well as DDR3 and a set of peripheral controllers. The Cortex-A9 cores are connected to 1 GB DDR3 RAM operating at 400 MHz. These blocks are connected via multilayer AXI interconnect structure. Programmable logic part of the SoC is a high-performance 28 nm FPGA. The HPS and FPGA part of the chip are connected via high-bandwidth (> 125 Gbps) on-chip interfaces.

All the tests presented in this paper use only the HPS part of the board, the FPGA part is not used. The Cortex-A9 MPCore runs a Linux kernel version 3.16.0.

C. Virtual Hardware

Our OVP virtual hardware platform implements just a part of the actual Altera Cyclone V SoC reference hardware [5]. Specific and unnecessary hardware parts, e.g., the FPGA block, are not implemented. Yet, all components required for running a Linux kernel and guaranteeing correct hardware functionality for our test cases are available. Figure 3 shows a subset of implemented hardware components. The virtual hardware allows to boot the same Linux kernel as the actual hardware does.

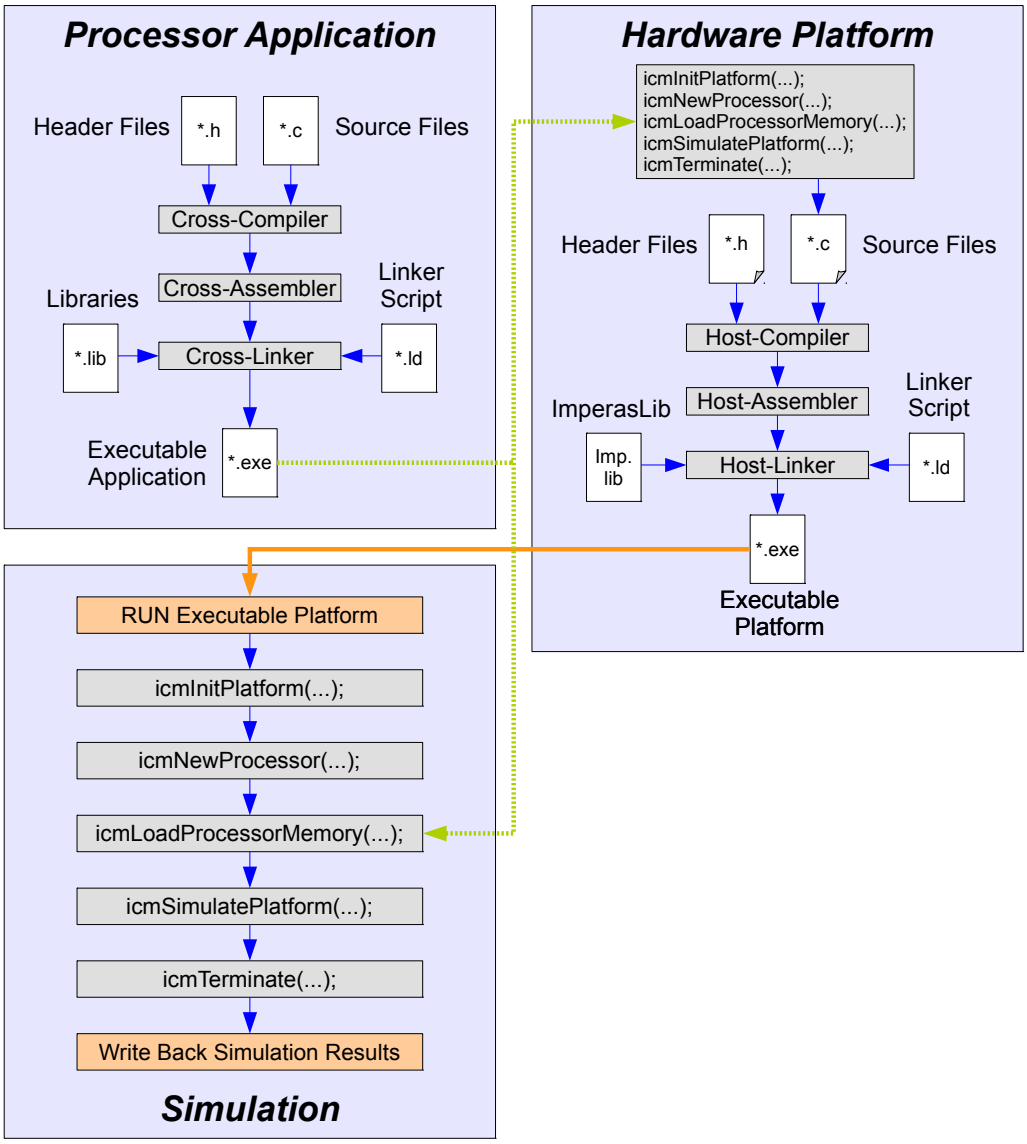


Fig. 2. Operating principle of a basic Open Virtual Platforms simulation.

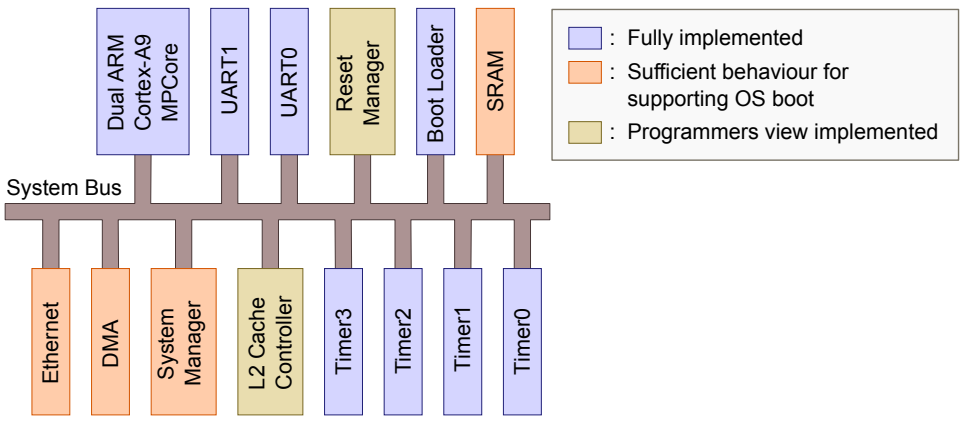


Fig. 3. Virtual Cyclone V SoC.

IV. Case Study

A. Problem Description

For our experiments, we choose a diffusive type PDE as it arises in many application domains such as physics, material sciences, computational fluid dynamics and many more. In detail, we consider the steady-state heat equation with Dirichlet boundary conditions on the unit square:

$$\begin{aligned} -\operatorname{div}(a \operatorname{grad} u) &= f \quad \text{in} \quad \Omega\,, \\ u &= g \quad \text{on} \quad \partial\Omega\,. \end{aligned} \tag{2}$$

Here, $a : \mathbb{R}^d \to \mathbb{R}$ describes, e.g., the thermal conductivity of the material. Furthermore, $\Omega = (0,1)^d$, $\operatorname{div} : \mathbb{R}^d \to \mathbb{R}$ is the divergence and $\operatorname{grad} : \mathbb{R} \to \mathbb{R}^d$ is the gradient.

From this family of problems, we chose one specialized case which is given by

$$\begin{aligned} f(x,y,z) &= -2k\big((y-y^2)(z-z^2) \\ &\quad + (x-x^2)(z-z^2) + (x-x^2)(y-y^2)\big) \\ a(x,y,z) &= -e^{k(x-x^2)(y-y^2)(z-z^2)} \\ u(x,y,z) &= 1 - e^{-k(x-x^2)(y-y^2)(z-z^2)}\,. \end{aligned} \tag{3}$$

Additionally, k and g are chosen to be 10 and 0, respectively.

The arising equation is then discretized on a regular grid with $2^8 + 1$ grid points per dimension using finite differences. This, in consequence, results in a linear system of equations (LSE) with roughly 16 million unknowns, which is then solved using our generated geometric multigrid solver. The solver itself uses a V(3,3)-cycle with a damped Jacobi smoother and a conjugate gradient (CG) coarse grid solver. Optimization of the implementation is carried out using the techniques described in section II-B3 except of vectorization, which is deferred to future work.

We execute the solver generated by the ExaStencils framework and compiled for the chosen target platform as described in section III-B. Due to the (very) limited memory, fitting our test problem onto the hardware leads to some problems. For example, the stencil coefficients, which vary for every grid point, are usually stored and re-used in order to decrease computational intensity. However, in our context, this approach is not possible which is why we switch to computing the coefficients on the fly at the cost of additional computational overhead.

B. Results

When generating and executing the solver described previously, we have the option of adding parallelization based on OpenMP. For our test problem, an almost identical behavior and nearly perfect scaling can be observed for both cases (with and without parallelization). Thus, for reasons of clarity, we choose to limit our presentation to the OpenMP case. For our measurements, the generated solver is compiled and, using the same binary, executed on the real and the simulated platform, respectively.

As can be seen in figure 4, a high timing prediction accuracy is achieved for compute-bound parts of the problem, namely smoothing and updating the residual. In detail, we measured a deviation of less than 20 %, which, in turn, can be mostly attributed to non-optimal pipeline usage in the computation of the exponential function. As said computation is currently done using a library

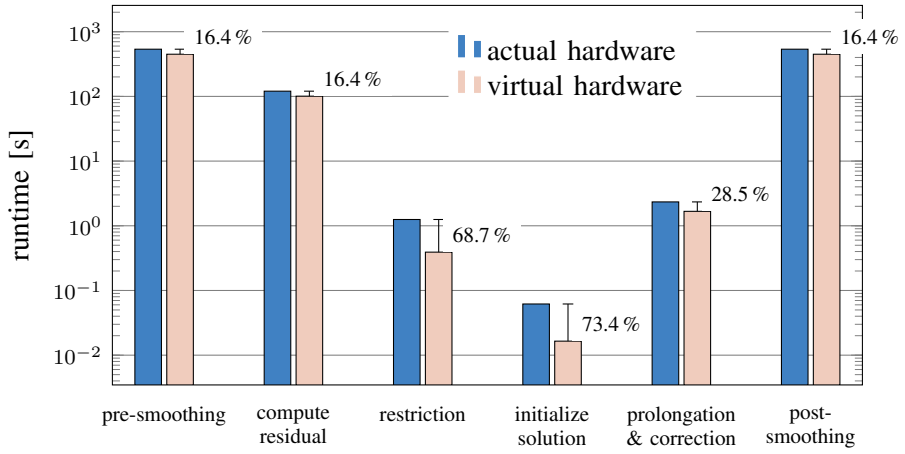


Fig. 4. Measured and simulated times of multigrid components for a whole solving step on a log scale. Names of computational kernels map directly to the algorithm description in Algorithm 1.

routine, the resulting discrepancy is constant.

Since the utilized OVP simulation does not yet contain a sophisticated memory model of the platform including, e.g., private and shared caches, there are bigger deficits when targeting computational kernels that are memory bound. In our tests, this is reflected in the times for restriction and setting the solution as depicted in figure 4. However, considering that our simulation model is still in an early development stage, and that a detailed memory model will enhance applicability, our simulation results motivate our approach.

V. Related Work

One of the most well-known projects about code generation is SPIRAL, who optimize by applying linear transformations. In recent work [11], support for the ARM vector extensions NEON was added.

Generating code based on algorithmic descriptions in DSLs is widely used in the domain of imaging processing. For instance HIPAcc is able to utilize (embedded) CPUs and GPUs [13] and also FPGAs [18]. Its DSL is embedded into C++, while the compiler back-end uses Clang and LLVM as its foundation. Albeit being possible to implement multigrid algorithms, certain smoother types cannot be described. Furthermore, computational domains may only be two-dimensional. Another example from the same domain is Halide [15], which also utilizes C++ as its host language for a functional representation of the algorithm. It can also generate code using ARM NEON vector extensions.

More generally, Renderscript [2], a framework for running computationally intensive tasks on Android devices, generates code for the very architecture it is running on, based on LLVM bytecode. This device-specific code is cached permanently. Renderscript is implemented as an internal DSL based on C99, which has been extended by support for vector types.

Looking at development in HPC hardware, a lot of effort has gone into the research of architectures consisting of embedded processors. This topic is not only interesting to academia [14], [16], but also large industrial manufacturers are researching or already building such systems.

VI. Conclusion

We demonstrated that, using the ExaStencils framework, code generation starting from an abstract DSL representation is possible and viable also for ARM-based target platforms. For this target, no changes in the DSL code are necessary, which allows for an efficient evaluation of different hardware platforms by end users.

OVP-based simulations of our generated solvers attain a prediction errors below 20 % for compute-bound parts, which is reasonably good for our setup. In contrast, the insufficient memory model of the simulation tool reflects in the results for memory bound parts of our solvers. Nevertheless, we think that this deficiency can be resolved in the future and that accurate predictions for the complete application are possible.

This further motivates our long-term goal of designing potential future hardware highly attuned to the needs of special classes of applications, such as our multigrid solvers, and gives us the opportunity for rapid design space exploration of low power hardware.

VII. Future Work

Following our results, the next step necessary is to extend our simulation model to make the results even more accurate. Firstly, we plan to add a memory model that allows getting precise results concerning chache hierarchies and memory accesses. Secondly, an energy model is of high interest to us. With it, precise statements about energy consumption in specific parts of the code, or in the whole application, can be obtained. The foundation for these extensions special to our ARM model is already given as the OVP platform currently supports both, modeling of memory accesses [7] and energy [17].

For the code generation process, support for ARM NEON will be added to increase performance—and most probably—energy efficiency of the resulting code. Furthermore, our code generator is already capable of emitting code parallelized not only with OpenMP but also with MPI. Thus, it would be highly interesting for us to evaluate the performance of generated solvers on small and large scale ARM clusters like the ones developed by the Mont-Blanc project [16].

Since the platform used for evaluation in this work features a reconfigurable component, investigating the hardware/software partitioning, i.e., doing a hybrid parallelization using both ARM cores and the FPGA chip at the same time, would be a worthwhile research goal.

VIII. Acknowledgments

This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 Software for Exascale Computing, under the contracts RU 422/15-1 and TE 163/17-1.

REFERENCES

[1] Altera Corporation: Cyclone V SoC Development Kit User Guide (2013), https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_cv_soc_dev_kit.pdf, last visit on 07.05.2015

[2] Google: RenderScript (2015), http://developer.android.com/guide/topics/renderscript/compute.html, last visit on 28.05.2015

[3] Grebhahn, A., Kuckuk, S., Schmitt, C., Köstler, H., Siegmund, N., Apel, S., Hannig, F., Teich, J.: Experiments on Optimizing the Performance of Stencil Codes with SPL Conqueror. Parallel Processing Letters 24(3), Article 1441001, 19 pages (2014)

[4] Hackbusch, W.: Multi-Grid Methods and Applications. Springer-Verlag (1985)

[5] Imperas Software Limited: Description of Altera Cyclone V SoC (2015), http://www.ovpworld.org/library/wikka.php?wakka=AlteraCycloneVHPS, last visit on 29.04.2015

[6] Imperas Software Limited: OVP Guide to Using Processor Models. Imperas Buildings, North Weston, Thame, Oxfordshire, OX9 2HA, UK (January 2015), version 0.5, docs@imperas.com

[7] Imperas Software Limited: OVPsim and Imperas CpuManager User Guide. Imperas Buildings, North Weston, Thame, Oxfordshire, OX9 2HA, UK (January 2015), version 2.3.7, docs@imperas.com

[8] Kronawitter, S., Lengauer, C.: Optimizations Applied by the ExaStencils Code Generator. Tech. Rep. MIP-1502, Faculty of Informatics and Mathematics, University of Passau (2015)

[9] Kuckuk, S., Gmeiner, B., Köstler, H., Rüde, U.: A Generic Prototype to Benchmark Algorithms and Data Structures for Hierarchical Hybrid Grids. In: Proc. Int. Conf. on Parallel Computing (ParCo). pp. 813–822. IOS Press (2013)

[10] Kuckuk, S., Köstler, H.: Automatic Generation of Massively Parallel Codes from ExaSlang. Computation 4(3), Article 27, 20 pages (Sep 2016), http://www.mdpi.com/2079-3197/4/3/27, special Issue on High Performance Computing (HPC) Software Design

[11] Kyrtatas, N., Spampinato, D.G., Püschel, M.: A Basic Linear Algebra Compiler for Embedded Processors. In: Proc. 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1054–1059 (2015)

[12] Lengauer, C., Apel, S., Bolten, M., Größlinger, A., Hannig, F., Köstler, H., Rüde, U., Teich, J., Grebhahn, A., Kronawitter, S., Kuckuk, S., Rittich, H., Schmitt, C.: ExaStencils: Advanced Stencil-Code Engineering. In: Euro-Par 2014: Parallel Processing Workshops. Lecture Notes in Computer Science, vol. 8806, pp. 553–564. Springer (2014)

[13] Membarth, R., Reiche, O., Hannig, F., Teich, J.: Code Generation for Embedded Heterogeneous Architectures on Android. In: Proc. Conf. on Design, Automation and Test in Europe (DATE). pp. 86:1–86:6 (2014)

[14] Padoin, E., de Oliveira, D., Velho, P., Navaux, P.: Time-to-Solution and Energy-to-Solution: A Comparison between ARM and Xeon. In: Proc. Workshop on Applications for Multi-Core Architectures (WAMCA). pp. 48–53 (2012)

[15] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In: Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI) (2013)

[16] Rajovic, N., Carpenter, P.M., Gelado, I., Puzovic, N., Ramirez, A., Valero, M.: Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In: Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (Supercomputing). pp. 40:1–40:12. ACM (2013)

[17] Rosa, F., Ost, L., Raupp, T., Moraes, F., Reis, R.: Fast energy evaluation of embedded applications for many-core systems. In: Proc. Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS) (2014)

[18] Schmid, M., Reiche, O., Schmitt, C., Hannig, F., Teich, J.: Code Generation for High-Level Synthesis of Multiresolution Applications on FPGAs. In: Proc. Int. Workshop on FPGAs for Software Programmers (FSP). pp. 21–26 (2014)

[19] Schmitt, C., Kuckuk, S., Hannig, F., Köstler, H., Teich, J.: 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). pp. 42–51. IEEE Computer Society (2014)

[20] Schmitt, C., Kuckuk, S., Köstler, H., Hannig, F., Teich, J.: An Evaluation of Domain-Specific Language Technologies for Code Generation. In: Proc. Int. Conf. on Computational Science and its Applications (ICCSA). pp. 18–26. IEEE Computer Society (2014)

[21] Schmitt, C., Schmid, M., Hannig, F., Teich, J., Kuckuk, S., Köstler, H.: Generation of Multigrid-based Numerical Solvers for FPGA Accelerators. In: Proc. Int. Workshop on High-Performance Stencil Computations (HiStencils). pp. 9–15 (2015)

[22] Trottenberg, U., Oosterlee, C.W., Schüller, A.: Multigrid. Academic Press (2001)