This is the author's version of the work. The definitive work was published in *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations (HiStencils), Amsterdam, The Netherlands, Jan 20, 2015.*

Generation of Multigrid-based Numerical Solvers for FPGA Accelerators

Christian Schmitt[‡], Moritz Schmid[‡], Frank Hannig[‡],
Jürgen Teich[‡], Sebastian Kuckuk[†], and Harald Köstler[†]

[‡]Hardware/Software Co-Design, Department of Computer Science
[†]System Simulation, Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

ABSTRACT

Not only in the field of High-Performance Computing (HPC), Field Programmable Gate Arrays (FPGAs) are a soaringly popular accelerator technology. However, they increase the heterogeneity of clusters, which might be equipped already today with accelerators, such as GPUs. This results in having to combine expertise from different fields, e. g., mathematical, algorithmic and technical experts are needed to create numerical solvers for such systems. 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 the generation of implementations of numerical solvers based on the multigrid method for FPGAs from the same codebase 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 12 V-cycles per second with an input grid size of 4096×4096 on a mid-range FPGA, beating vectorized, single-threaded execution on an Intel i7 by a factor of almost three.

1. INTRODUCTION

Already today, a large percentage of clusters and supercomputers are equipped with accelerators and we expect that, in order to achieve exascale performance, the use of such technologies will intensify and different solutions will be employed 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, allowing to formulate a description of the program in native terms. 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 [7], which research the feasibility of generating highly scalable numerical solvers based on the multigrid method. As input, a multi-layered DSL is proposed, making it possible to tailor each layer especially 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 a Application Specific Integrated Circuit (ASIC).

In this work, we demonstrate the feasibility of generating hardware-based multigrid solvers from a Domain-Specific Language via High-Level Synthesis. We substantiate our claims by providing results of a solver that has been generated from our DSL, called ExaSlang 4, and processed by Vivado HLS. However, the presented approach is applicable to any C-based High-Level Synthesis in general.

HiStencils 2015
Second International Workshop on High-Performance Stencil Computations
January 20, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://www.exastencils.org/histencils/2015/

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

For HLS, 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, Catapult by Calypto, Forte (now Cadence) Cynthesizer and Synopsys' Synphony C Compiler.

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 [16].

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 [4], 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 [5].

In previous work, the benefits of domain-specific optimization have been shown in various domains. *SPIRAL* [9], 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* [15] and *FFTW* [3] 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* [2], which adds abstractions to Java to ease stencil computations for unstructured problems, and *Pochoir* [14], which employs a divide-and-conquer skeleton on top of the parallel C extension Cilk to make stencil computations cache-oblivious. *PATUS* [1] 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* [10] generates, among others, CUDA and OpenCL code from a DSL embedded into C++. The same description can be transformed to Verilog by *Darkroom* [6].

HIPAcc [8] provides native support for multigrid methods by offering appropriate language elements. 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 [12].

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). One of the most researched PDEs is Poisson's equation which is used for modeling diffusion processes, e. g., in the simulation of temperature distributions.

The V-cycle, one variant of a multigrid method, is shown in Algorithm 1. 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. Low-frequency components are transformed into high-frequency components by restricting them to a coarser level, thus making them good targets for the smoother once more.

On the coarsest level, direct solving of the remaining linear system of equations is possible due to its low number of unknowns. However, it is also possible to apply a number of smoother iterations. In the case of a single unknown, one smoother iteration corresponds to directly solving the problem.

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}
  $e_H = V_H\,(0, A_H, r_H, \nu_1, \nu_2)$ {recursion}
  $e_h = P e_H$ {interpolate 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 $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A_h, f_h, \nu_1, \nu_2\right)$.

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 abstractness, 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 by providing appropriate language elements such as fields and stencils. It is a procedural 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

```
1  Stencil Laplace @all {
2    [ 0,  0] => 4.0
3    [ 1,  0] => -1.0
4    [-1,  0] => -1.0
5    [ 0,  1] => -1.0
6    [ 0, -1] => -1.0
7  }
8
9  Function Smoother @all () : Unit {
10   loop over Solution @current {
11     Solution2 @current = Solution @current +
       ((( 1.0 / diag(Laplace @current)) * 0.8)
       * (RHS @current - Laplace @current *
       Solution @current))
12   }
13 }
```

Listing 1: Example of a stencil definition and its use as part of a weighted Jacobi smoother in 2D.

```
1  Function Smoother @all () : Unit {
2    /* ... */
3  }
4
5  Function Smoother @(finest - 1) () : Unit {
6    /* ... */
7  }
8
9  Function VCycle @all () : Unit {
10   repeat 3 times {
11     Smoother @current ()
12   }
13   /* ... */
14 }
```

Listing 2: Example of overriding the default smoother function at a specific multigrid level.

communicated. A more thorough description of the language and its code generation and transformation framework can be found in [13].

4.1 Language Elements

Being a language that targets the description of multigrid-based numerical solvers, ExaSlang 4 combines elements of procedural languages, such as functions and loops, with domain-specific elements, e. g., stencils and communication-enabled memory arrays.

ExaSlang 4 features special data types that we call Algorithmic data types, as they stem from the language's domain. Therefore, `Stencil` and `Field` can only be as part of numerical calculations. They are explained in more detail in the next paragraphs.

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 Listing 1, where a weighted Jacobi smoother is described.

Furthermore, this example showcases the `loop over` statement, a language element that is used to instantiate an iteration over the computational domain (or a part of it) by defining a Field that is used to determine the computational bounds.

Fields represent the actual data that stencils are applied to. They may be given by the user, e. g., as the right-hand side of a calculation, or the unknown to be solved for. A field is defined by providing a layout which specifies further options, e. g., enables communication among the domain partitions. Furthermore, an underlying numerical data type has to be provided.

A key element of ExaSlang 4 are Level Specifications. They allow the definition of objects—such as functions, variables, stencils and fields—depending on multigrid levels and can be used to override defaults at specific levels, as illustrated in Listing 2. Here, a smoother that is different from the one called at other levels is implemented for the second-finest level. Identifiers such as `current` and `finest` can be used to ease modifying the depth of the multigrid recursion without rewriting large parts of the DSL code. Consequently, references to level-dependent entities, e. g., function calls, need to include the target level.

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), which target platform specific alterations and optimizations are applied to. 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 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. A key concept in hardware development is to design a component once and replicate it as often as necessary. For multigrid algorithms,

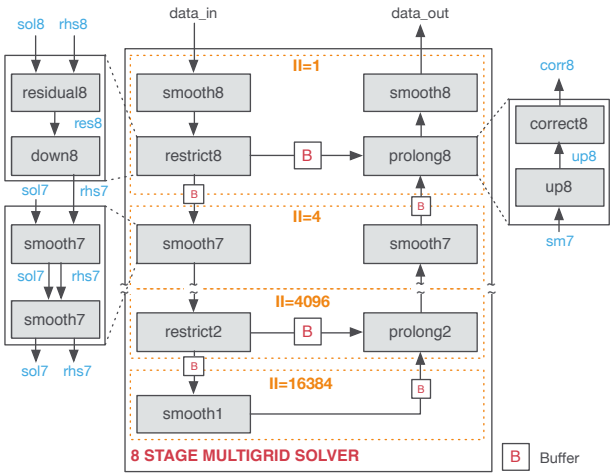


Figure 1: Structural representation of the multigrid algorithm implementation.

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 pipeline interval, also often called Iteration Interval (II), which describes the amount of clock cycles between the arrival of new data elements. To achieve a high performance, the top most 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, a dedicated operator must be instantiated for each operation of the algorithm, 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 1. 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 (a) after down samplers (part of restrict), (b) before up samplers (part of prolong), (c) 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 [11]. 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.

During the generation of CPU code, stencil weights are resolved directly into the calculations to reduce memory accesses and enable 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, as depicted in algorithm 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, in-

stantiate 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. Supporting these constructs inside of the V-cycle, for example for solving the coarsest grid is difficult, since it would require the repeated sequential execution of kernel, which would interrupt the streaming pipeline and require extensive buffering of the results of the higher levels. Another use case for the construct is the repeated application of the V-cycle until desired precision has been reached. However, this does not affect hardware generation.

6. CASE STUDY

To evaluate our approach, we consider a typical example problem: A solution to a finite differences (FD) discretization of Poisson's equation with Dirichlet boundary conditions. We have implemented a typical multigrid solver in ExaSlang 4 using a V(2,2) cycle and a recursion depth of 8. For smoothing, a weighted Jacobi with a pre-calculated optimal ω is used. Solution on the coarsest grid of 32×32 is approximated by multiple smoother steps.

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 mid-range Xilinx Kintex 7 (xc7k325t) FPGA. Although interconnecting individual modules using FIFO streams and executing these concurrently is supported by the tool using the `data flow` directive, the buffer size is statically set to 1 and there is no mechanism for automatic determination of the necessary depth of the buffer. To allow uninterrupted execution, buffers on all levels must provide enough space to not produce back-pressure on the pipeline, which might result in a deadlock, once the upper most level is affected. In addition to static delays between the interconnected functions, due to the latencies of the hardware modules, a complex pipeline consisting of multiple up- and down-sampling steps also produces runtime delays, which vary according to the chosen architecture and are hard to anticipate. As under provisioning may at least decrease the performance and over provisioning may waste resources, we have used logic simulation to determine the necessary buffer sizes, which are shown in Table 1. 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 2. 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 maximum amount of memory space that can be allocated to a 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 2. 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. Table 2 shows evaluation results of the hardware synthesis from Vivado HLS, which give a rough estimate a conservation approximation of the resource requirements of the design. Indeed, after externalizing the top four largest buffers for results and RHS, the design can be fit onto the chip. Table 3 lists the post place and route (PPnR) results after integrating the modified accelerator into the described FPGA support system and shows that the multigrid solver can be implemented on a Kintex 7 and

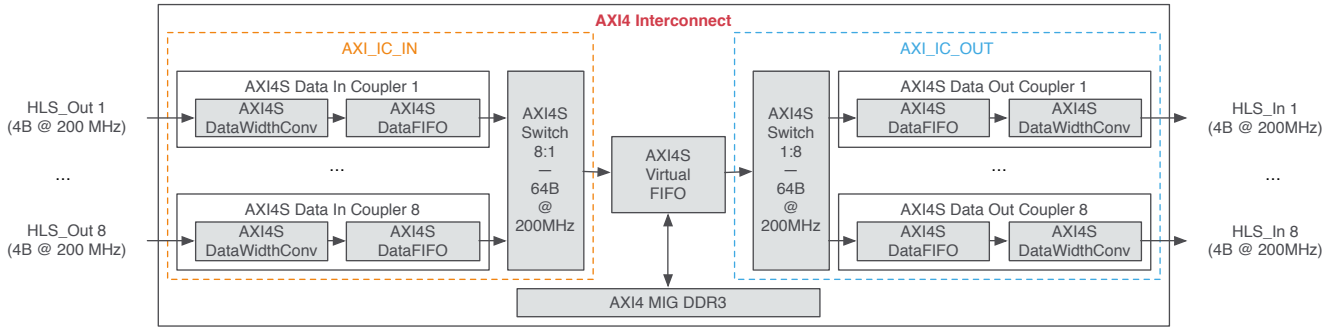
Level	RHS buffer	Result buffer	4KB Pages
Stage 8	4430946	4430936	4328
Stage 7	1094407	1094397	1069
Stage 6	266942	266932	261
Stage 5	63728	63718	63
Stage 4	14443	14435	NA
Stage 3	2866	2856	NA
Stage 2	338	328	NA
Stage 1	1	64	NA

Table 1: Buffer sizes for interconnecting the stages of the V-cycle.

Table 2: HLS resource estimates for the multigrid solver design comparing on-chip and external buffering.

Resource	On-Chip	External	Available
FF	256368	106311	407600
LUT	880314	177379	203800
BRAM	11812	323	445
DSP48	442	442	840

Figure 2: Structural representation of the FPGA support design.



the design achieves a maximum clock frequency of over 200 MHz. We have evaluated the PPnR hardware results on the

Table 3: PPnR resource requirements of the complete multigrid solver design.

LUT	FFs	DSPs	BRAMs	Slices	F_{max}[MHz]
105951	135442	460	808	39147	202.34

Kintex-7 FPGA to measure the performance in terms of how many clock cycles it takes it takes to process a 4096×4096 grid of floating point values. 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. Using the concept of data streaming, we can also hide the communication time with a host that supplies the input data completely, as current high-speed serial interconnects, such as PCI express and others, can achieve data rates that exceed the throughput of the accelerator by far.

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. The evaluation was done on a single core of an Intel i7-3770, which is clocked at 3.40 GHz and features L2 and L3 cache sizes of 1 MB, respectively 8 MB. For the CPU, AVX vectorization was enabled. Table 4 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

Table 4: Comparison of the performance of the multigrid solver on different hardware targets.

Target	Latency [ms]	Throughput [Vps]
Kintex-7	83.1	12.3
Intel i7	223.1	4.5

mentioning that it is irrelevant to the performance of the accelerator, whether the input data uses only single-precision floating point or is implemented for double-precision input data. Although the chosen mid-range Kintex-7 FPGA cannot provide the necessary amount of logic and memory resources, switching to a larger FPGA, such as a member of the Virtex-7 family, here an XC7VX485t, can easily solve this issue. To underline this, we have generated source code for HLS of the multigrid solver using double precision floating point arithmetic, for which the estimated resource requirements are listed in Table 5.

Table 5: HLS synthesis estimates for implementation of the multigrid solver using double precision arithmetic. Values are given as percentage of available resource type.

FPGA	LUT	FFs	DSPs	BRAMs	F_{max}[MHz]
Kintex-7	140	43	111	124	232.0
Virtex-7	73	29	33	53	229.4

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 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 onto two different hardware targets, a Kintex-7 FPGA and an Intel i7 CPU. Both implementations were generated from the same code base in ExaSlang. Additionally, evaluation numbers show that employing FPGAs in HPC is a promising approach to increase computing power, while, at the same, to reduce the energy footprint of clusters.

8. FUTURE WORK

While ExaSlang can be used to describe multigrid-based solvers for three and more dimensions and code generation works, the underlying concept of streaming and buffering needs some refinement. 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.

Automatic insertion of split kernels to duplicate streams is currently unavailable, as data flow and depency analysis as part of the ExaStencils transformation framework is not yet finished. From such information, an information-rich call graph could be build up, easing generation of linearized kernel executions.

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.

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 in project under contracts TE 163/17-1 and RU 422/15-1.

References

[1] 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.

[2] 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.

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

[4] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich. "PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications." In: *Proc. of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*. (London, United Kingdom). Vol. 4943. Lecture Notes in Computer Science (LNCS). Springer, Mar. 26–28, 2008, pp. 287–293.

[5] F. Hannig, M. Schmid, J. Teich, and H. Hornegger. "A deeply pipelined and parallel architecture for denoising medical images." In: *Proc. of the IEEE International Conference on Field Programmable Technology (FPT)*. (Beijing, China). IEEE, Dec. 8–10, 2010, pp. 485–490.

[6] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. "Darkroom: Compiling high-level image processing code into hardware pipelines." In: *ACM Transactions on Graphics (TOG)* 33.4 (July 2014), 144:1–144:11.

[7] 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.

[8] 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.

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

[10] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." In: *ACM Transactions on Graphics (TOG)* 31.4 (July 2012), 32:1–32:12.

[11] M. Schmid, N. Apelt, F. Hannig, and J. Teich. "An image processing library for C-based high-level synthesis." In: *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL)*. (Munich, Germany). Sept. 2–4, 2014.

[12] 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.

[13] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. "Exaslang: a domain-specific language for highly scalable multigrid solvers." In: *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC). To appear*. (New Orleans, LA, USA). IEEE, Nov. 17, 2014, 10 pp.

[14] 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.

[15] 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.

[16] Xilinx Inc. *Vivado Design Suite User Guide – HLS*. User Guide. 2013.