An Architecture Framework for Porting Applications to FPGAs

Fabian Nowak, Michael Bromberger and Wolfgang Karl
Karlsruhe Institute of Technology
Chair for Computer Architecture and Parallel Processing
76128 Karlsruhe, Germany
Email: <lastname>@kit.edu

***Abstract*—High-level language converters help creating FPGA-based accelerators and allow to rapidly come up with a working prototype. But the generated state machines do often not perform as optimal as hand-designed control units, and they require much area. Also, the created deep pipelines are not very efficient for small amounts of data. Our approach is an architecture framework of hand-coded building blocks (BBs). A microprogrammable control unit allows programming the BBs to perform computations in a data-flow style. We accelerate applications further by executing independent tasks in parallel on different BBs. Our microprogram implementation for the Conjugate-Gradient method on our data-driven, microprogrammable, task-parallel architecture framework on the Convey HC-1 is competitive with a 24-thread Intel Westmere system. It is $1.2\times$ faster using only one out of four available FPGAs, thereby proving its potential for accelerating numerical applications. Moreover, we show that hardware developers can change the BBs and thereby reduce iteration count of a numerical algorithm like the Conjugate-Gradient method to less than $0.5\times$ due to more precise operations inside the BBs, speeding up execution time $2.47\times$.**

I. Introduction

Once being too small in terms of area and too slow in terms of clock rate, FPGAs are now sufficiently large and fast to accelerate complex algorithms. They are employed in several different target domains, ranging from digital signal processing to scientific computations. In the latter domain, processing floating-point data is of paramount importance.

Porting algorithms to FPGAs and implementing them requires great efforts. The developer has to cope with implicit parallelism, synchronization issues, clock rates, and data paths. As a result, tools for converting sequential high-level language code to hardware descriptions came up [6], trying to ease hardware development and to make FPGA hardware programmable for high-level application programmers. Each of these converters makes assumptions about data exchange, synchronization, hardware capabilities and interfaces. Much work is required to successfully interface with the surrounding hardware. Taking this into account, the generated designs are not portable from one FPGA environment to the other. To exchange data between interfaces and the generated or instantiated arithmetic-logical circuits, state machines are generated by the converters, which consume much space due to many required states when controlling pipelining and targeting data-driven execution. Although development is facilitated, the time-consuming hardware synthesis is still required, and the development cycle still consists of describing the algorithm, simulating, emulating, synthesizing and finally testing the resulting hardware design. Overall, the existing methodologies for creating hardware designs are quite cumbersome.

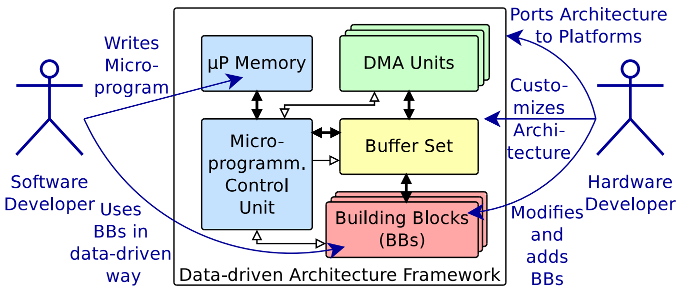
Writes Micro-program
µP Memory
DMA Units
Ports Architecture to Platforms
Custo-mizes Archi-tecture
Micro-programm. Control Unit
Buffer Set
Software Developer
Hardware Developer
Uses BBs in data-driven way
Building Blocks (BBs)
Modifies and adds BBs
Data-driven Architecture Framework

Figure 1. Data-driven architecture framework for FPGAs and responsibilities of hardware and software developers.

Our approach to this problem is microprogramming the control of medium-grained, preconfigured, existing building blocks (BBs), and calling the microprogram from software side on the host. We propose an architecture framework as depicted in Fig. 1 that consists of a microprogram memory, control unit, functional and data-transferring blocks, and of a decoupling buffer set to enable both data-driven and task-parallel execution. Once an FPGA is configured with our framework, an application developer needs to come up with microprogram implementations of the to-be-ported algorithm, but can ignore hardware constraints such as streaming, caching, data-parallel and task-parallel execution, resource usage, and clock rates. As changes to the microprogram can be tested within seconds, the developer is able to debug at software level and is freed from waiting several hours for the synthesis.

When using reconfigurable accelerators, data transfer poses a limiting factor. However, with our data-driven architecture framework, this is no longer the case as data are stored internally and reused for further computations. Targeting numerical programs, the framework allows accelerating the Conjugate-Gradient method [10], for example, because it is no longer memory-bound. To port further algorithms to the architecture framework, application developers only need to provide a different microprogram for the same building blocks. As a hardware developer, it is possible to change implementations of the BBs, e.g. for increased accuracy to reduce iteration count without changing the microprogram, though synthesis is required then.

Approaches for helping in FPGA designs are discussed in Section II. Next, Section III explains our concept. To prove portability, we implemented the architecture on two different FPGA systems (Section IV). We present previous evaluation results [10] of the framework in Section V. To take previous work one step further, we evaluate ease of high-level programming and optimization, and possibility for low-level hardware adaptations. Through adaptations such as more precise internal operations inside the BBs, the convergence criterion of numerical algorithm is reached faster. This reduces iteration count to less than $0.5\times$. Section VI closes the article and gives a short outlook.

II. Related Work

For scientific computing, efficient floating-point operation is of great importance. Over the last ten years, FPGAs have become capable of performing floating-point calculations, which has been researched extensively with regard to area, clock rate, precision, pipelining [9]. Very helpful is the automated creation of floating-point cores [2]. With such hardware capabilities and software tools, FPGAs can provide benefit for scientific computing [3].

One major issue with FPGAs is to develop the design, then simulate it, synthesize and test it. Therefore, High-Level Languages (HLLs) evolved, and with fixed-point and floating-point cores for all kinds of operations, it became possible to translate arithmetic codes to hardware descriptions that instantiate available cores for arithmetic operations. Among such translators, ROCC Compiler [6] is probably known best. Besides the C-based translators, there are efforts to accelerate poorly performing Matlab code, coming up with a complete toolbox and design flow for FPGA acceleration [8].

Basis for many translators is the SUIF compiler [17] that creates an abstract syntax tree upon which the translators further produce state machines and wiring of the instantiated cores or generated arithmetic-logical circuitries. In tackling the large area consumption of state machines in FPGA designs with more than 30 states, microprogramming proved helpful and has found its way into FPGA research [1].

In the MORA architecture [16], no explicit control is necessary because the arithmetic instructions in the program code are translated to parameterized processing elements that are connected with one another according to the data graph of the code, and the elements execute in a data-driven fashion. But MORA lacks from being freely (re-)configurable to another application, again requiring high synthesis time for new application accelerators. Same applies to Altera's OpenCL path where the to-be-accelerated kernel function is translated to a data-parallel, deeply pipelined hardware description in Verilog [14].

Although the use of high-level languages and converters such as MORA, OpenCL and ROCCC facilitates the use of FPGAs by not requiring error-prone developing of hardware descriptions and no time-consuming Modelsim-based simulations, the general methodology remains the same. At first, an algorithm must be ported to the language itself. Secondly, especially with OpenCL, a suitable mapping must be found, i.e., optimal dimension and grid size. This task requires more than only basic understanding of the hardware. Third, the algorithm description must be simulated in its high-level-language version and then in its translated version, i.e. emulating the hardware. Fourth, hardware synthesis and place&route will take several minutes or even hours. Only then can the ported algorithm and its mapping be tested in real hardware, potentially detecting performance bottlenecks or even implementation errors. Unless data is written back to external memory after each single operation, tracking an error is rather difficult and requires the use of expert tools such as Chipscope. Every change to the high-level code requires another time-consuming hardware synthesis run instead of reusing previously translated and synthesized subcomponents such as a vector adder. This process must be repeated again and again until the performance goal is met and execution delivers correct results. Normally, this takes a couple of days. Programmers of scientific applications however need an efficient and easy-to-program means for exploiting FPGA-based accelerators in diverse floating-point-intense applications. SHARC [7] tries to close this gap by parameterizing the instantiated cores for further iterations, e.g. for processing different color values. The connection of the cores can be changed at runtime by instantiating and parameterizing additional switch modules. These parameters are extracted from Matlab code and cannot be programmed by the user in a convenient fashion.

As a further problem, data exchange with FPGA-external memories and internal data storage frequently pose a huge problem when employing accelerators. The high-level tools can only help to a limited amount in optimizing data transfer. Thus, the programmer has to care for efficient communication and data reuse, which is a non-trivial task that requires profound understanding of the underlying system. The average domain specialist application programmer should be freed from such optimization tasks. Aside, the resulting optimized hardware implementations are no longer portable from a performance point of view.

Much benefit is gained from FPGAs when implementing special implementations that operate at bit level or cannot be executed efficiently in general-purpose processors, such as a highly accurate dot product implementation [12].

The Xilinx Vivado suite [4], [18] allows to easily connect different Intellectual Properties (IP) cores to an embedded processor like ARM using the AMBA bus interface. It is possible to integrate IPs from Xilinx, third parties or custom IPs. An IP can be defined at different abstraction levels like C source code, RTL description in Verilog or VHDL or as netlist. The focus of Vivado is to realize Systems on Chips (SoCs) using different IP cores. Their approach suffers from the interconnection bottleneck when several data-hungry, memory-intense IP cores have to communicate over the shared bus with memory. Instead, our focus is to allow easy and efficient programming of accelerators that are included in high performance computing (HPC) systems.

Using a rather small control unit compared to a soft core inside an FPGA, we can integrate more building blocks (BBs) on an FPGA. These BBs communicate efficiently and directly by using a central buffer set so that no bus-sharing or complex bus protocol is required. Therefore, we propose to combine micro-programming, data-driven programming and execution, and preexisting cores, i.e, BBs. Thereby, programmers can exploit potential data-level parallelism inside the BBs, task-level parallelism by executing several BBs concurrently, and foremost pipeline parallelism where available, which helps minimize data exchange with memory. In case special instructions shall be integrated, we favor the aproach suggested by Strozdka [15] of tightly collaborating mathematicians, i.e, domain specialists, and computer scientists, i.e, hardware experts, as is also depicted in Fig. 1.

III. Microprogrammable, Data-driven Architecture

We design the architecture as illustrated in Fig. 1 with portability, easy programmability and adaptivity in mind. Data are exchanged with external memory via DMA units and internally via a central buffer set to leverage data-driven execution. Execution is controlled via microprograms instead of fixed state machines that trigger the BBs, such as DMA or vector adder.

The microprograms are provided by users, e.g., domain specialists. In contrast, the BBs are implemented by hardware specialist. We achieve concurrency by data-driven execution of the BBs. Independent blocks can proceed completely concurrently, and dependent blocks can proceed in a pipelined fashion if output and input rates allow. The building blocks, in return, can exploit data-level parallelism. Internal operations in the BBs are pipelined, if possible. Different BBs can form a pipeline using the central buffer set. So, one BB (source) generates data that will be processed by another BB (drain) in later steps. Synchronization between BBs is achieved by the buffer set and therefore no extra mechanism is needed inside the control unit. The pipeline including different BBs is formed by the microprogramm. Hence, application domain specialists can employ the architecture according to their needs by only writing the microprogram without caring for the details of data transfer, data reuse or hardware synthesis and bitstream generation. This is also key to achieving portability of the architecture and of the developed microprograms.

In the microprogram assembly language, BBs are denoted by representative mnemonics, if possible, for example `vadd` to add a pair of vectors streamed via two input buffers. All instructions take registers or immediate values as arguments, and the 32 registers can be written to explicitly via `regw`. The central micro-programmable control unit depicted in Fig. 2 passes instructions to BBs or to the sequencer modifying the instruction pointer, or to the ALU for operations on the register set. No instruction pipelining is used inside the control unit because most microinstructions are asynchronous BB instructions that will occupy the BB for a long time. So, only litte performance gain can be achieved using instruction pipelining in our case. This aspect also saves resources on the FPGAs and therefore more BBs and buffers are instantiated.

As indicated in Fig. 3, BBs have two interfaces, one for instructions from the control unit, and the other for interaction with the buffer set. Assume the BB is a vector adder that consumes two input vectors from the buffer set and writes one result vector. Then the instruction is to add or subtract the vectors, and the parameter is the length of the vectors. Figure 4 gives the corresponding example microprogram for adding two vectors.

Figure 5 shows in detail the data flow between source and drain blocks via the central buffer set. Any unit is normally both source and drain, e.g, the vector adder consumes as a

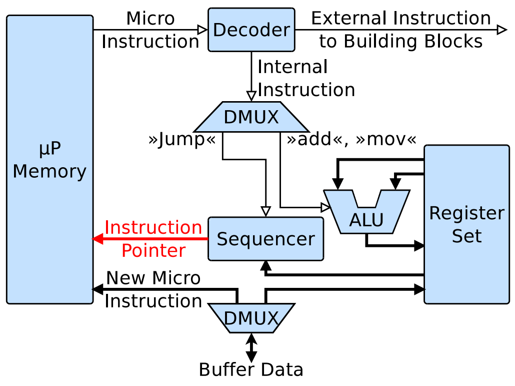

Figure 2. Micro-programmable control unit.

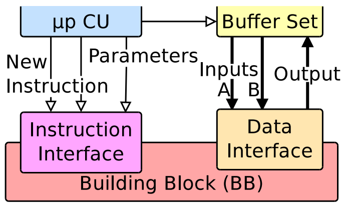

Figure 3. Instruction and data interface for BBs.

drain unit from the two source DMA units, and it is the source unit for the third DMA drain unit. The MUXes are assigned their configurations by `buf_assign`. A source block can write to any destination buffers due to the full crossbar interconnect. These are bound to exactly one drain block that consumes its data. The buffers are then filled by reading via DMA (`dmar`) or by a data-producing BB, and the outputs are consumed by writing to external memory via DMA (`dmaw`) or by another BB. In the end, a pipeline reading data from external memory, concurrently processing it within several BBs and already writing back some results can be established. Computation and communication in a data flow style support this achievement so that no more data dependencies need to be resolved manually. As shown in the program code in Figure 4, the domain specialist, non-hardware expert programmer only has to interconnect the blocks in the most natural data flow way. This can easily be accomplished via graphical user interfaces [13].

To generate machine-readable instructions from the assembly instructions, the toolchain sketched in Fig. 6 parses them and outputs binary code. It can also produce a coefficient file for static initialization of the microprogram memory at synthe-

```
regw R10, 20
buf_assign dma1, buf1
buf_assign dma2, buf2
dma1r 0xadd1, R10
dma2r 0xadd2, R10
buf_assign vadd1, buf3
vadd1 R10
dma3w 0xadd3, R10
j 0
```

Figure 4. Microprogram for adding two vectors.

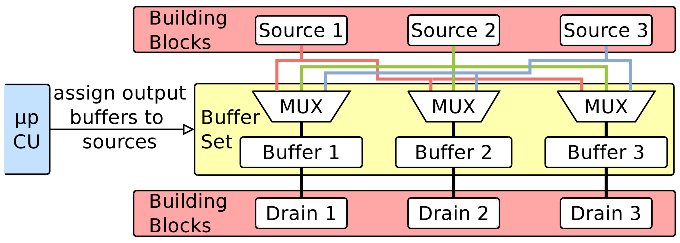

Figure 5. Buffer set and interface between building blocks.

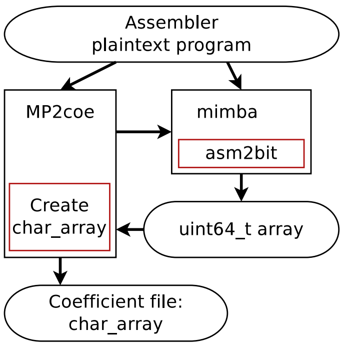

Figure 6. Toolchain parsing the assembly instructions, converting to binary code and generating memory configurations.

sis time. Otherwise, a driver mechanism needs to transfer the compiled microprogram. Depending on the system environment, this driver can even be embedded into the application.

IV. Framework Implementation and System Integration

To demonstrate portability, we implemented the architecture on different heterogeneous systems. The first implementation served as a prototype study and targeted the UoH HTX-Board that connects an FPGA via the HyperTransport Expansion (HTX) Socket to an AMD Opteron-based host system. Due to bad routing results on the chosen system with a Xilinx Virtex-4 FX100 FPGA, we do neither present detailed place&route results nor detailed evaluation results. Using two BBs, one for reading data and one for writing data, we can only process 500 MB data per second. The theoretically available bandwidth is 1600 MB/s per direction. On the one hand, there is overhead due to the protocol, and on the other hand, the host system does only allow 4MB contiguous DMA memory regions. Therefore, we can not fully exploit the bandwidth of the system. Hence, (pipeline-)parallel processing on the FPGA must compensate the low bandwidth usage by raising the computation-to-communication ratio. But integrating more BBs is out of scope for this system setup due to the tight timing results, unless running at a much lower frequency not optimally fitting the HyperTransport system.

To obtain performance-oriented results, we implemented the concept on the Convey HC-1 depicted in Fig. 7. We target supporting developers of numerical programs. Hence, we need several vector blocks that should execute in an overlapped fashion. Foremost, vectors need to be added and scaled (cmp. axpy operation). Secondly, dot products are required that can also be used for implementing matrix multiplications. Many solvers rely on stencils. Thus, we implemented a data-driven

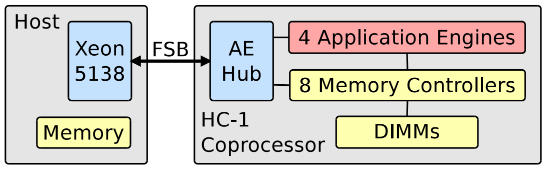

Figure 7. Convey HC-1 comprising 4 user-programmable FPGAs on a coprocessor board attached to CPU socket.

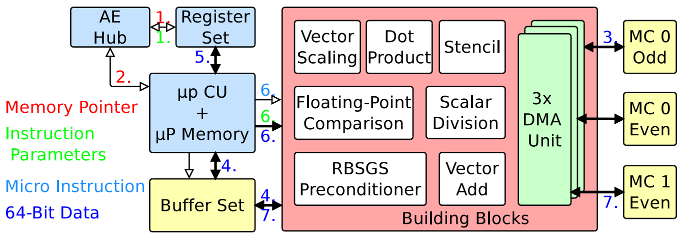

Figure 8. Implementation of the architecture framework on the Convey HC-1.

stencil unit. Upon this basis, a pipelined solver for Red-Black schemes was elaborated [10]. Moreover, a scalar division unit and comparison unit are required to calculate reciprocal values or divisors and to check when the algorithm has converged, respectively. Apart from these arithmetical blocks, data must be read from and written to external memory. Hence, a number of DMA units support several concurrent memory accesses in both read and write direction. As data is reused frequently due to the central buffer set, external memory access does no longer pose the main problem with reconfigurable computing.

The implemented framework, the arithmetical blocks and DMA units were implemented by hardware specialists. It is depicted in Fig. 8 together with the Convey AE interface, memory control logic, sequencer and buffer set. To use it, domain specialists only need to provide microprograms for controlling data flow, which might also be eased by providing a graphical user interface to connect the building blocks in data flow style.

Upon start of usage, the host application passes the memory references for the new, user-supplied microprogram, input data and output locations via the general register set (1.). Then (2.), the coprocessor is invoked, triggering the default microprogram to fetch the new microprogram from memory (3.). Having obtained the microprogram via a DMA unit and the buffer set (4.), the instructions of the new microprogram are decoded, the required registers are read (5.) and passed together with the opcode to the corresponding BB (6.). The blocks execute (7.) in a concurrent and data-driven fashion depending on the availability of data in the buffer sets.

V. Evaluation

We developed the data-driven, microprogrammable architecture for reconfigurable computing with special focus on developers of numerical applications. As case study for our evaluations, we employ the Conjugate-Gradient (CG) method, which is an iterative solver for linear equation systems. We solve the Poisson problem, so that we can employ a 2D stencil operation with problem-related coefficients instead of a matrix A and also a stencil-based preconditioner.

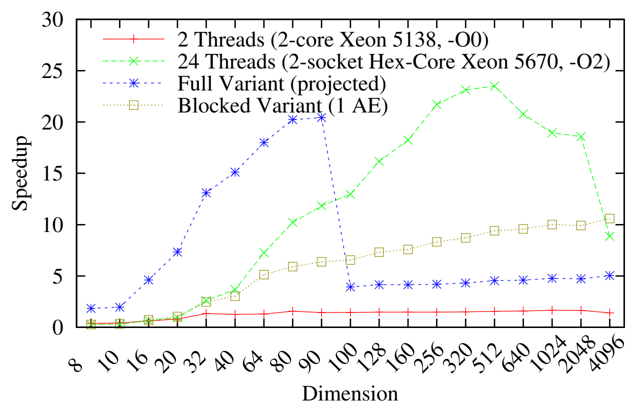

Figure 9. Speedup of CG method on architecture framework on Xilinx Virtex-5 LX330 and on Intel Xeon X5670 against Convey HC-1's Intel Xeon 5138.

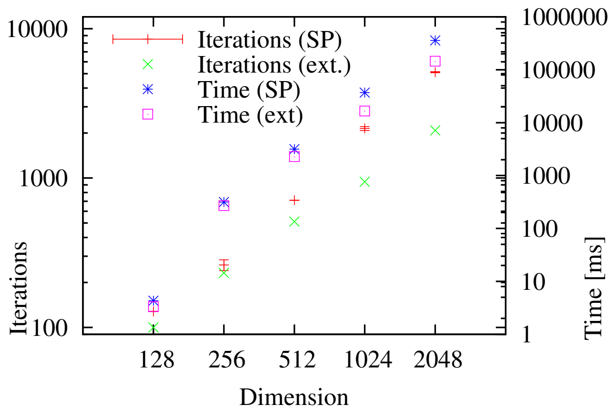

Figure 10. Reducing number of iterations from single-precision (SP) to higher precision (ext) for the dot product.

A. Accelerating the Conjugate-Gradient Method

To fully port the CG method, we would need 13 vector blocks and 9 DMA units. Among the 13 blocks are 3 vector adders, 3 vector-scaling blocks, 2 dividers, 2 dot products, 1 floating-point comparison, 1 stencil unit, and 1 red-black symmetric Gauss-Seidel preconditioner consisting of two differently preconfigured stencil units. Instantiating all these blocks requires a vast amount of BRAMs and DSP units and produces a congested design with densely packed control logic so that 150 MHz timing is not feasible. Hence, we split the main loop of the CG method into 3 distinct parts ("Blocked Variant") so that only one instance of each unit is required, together with only 3 DMA units. The implementation results of the fully functional design are given in Table I. We obtained speedup of 10× over sequential execution time on host processor Intel Xeon 5138 of the the Convey HC-1 as illustrated in Fig. 9 [10]. The untimable design is also displayed, labelled "Full Variant", based on early measurements and a simple timing model. It can provide its massive speedup when its small 90×90 buffers suffice for handling the input data. Our blocked design is even 1.2× as fast as a 24-thread version for large dimensions with optimization turned on (-O2).

B. High-Level Optimization

It might be advantageous to change the order in which the micro instructions are called so that more data would be transferred earlier, thereby potentially enhancing memory bandwidth usage. Knowing the best order is also important for automatically synthesizing the microprograms. Hence, we evaluated three versions of the microprograms for the CG blocks. In the first, original version, DMA operations are slightly optimized toward fetching data while also executing independent operations. We aggressively optimize the second version toward starting all DMA operations as early as possible. And finally, the third version calls DMA operations as late as possible. From the results given in Table II, we can conclude that the order plays only a minor role because a new instruction is fetched, decoded and executed every few cycles and the micro-programmable control unit (μpCU) is not stalled, but only waits for availability of the BB before passing the opcode and parameters. For larger data, it seems slightly advantageous to start DMA operations rather later than sooner, thereby overlapping computation with data transfers as early and much as possible. However, the results do not provide statistical relevance. With this experiment, we have shown that no deep understanding of FPGA development and underlying principles is required from application programmers when using this architecture. Further, optimally exploiting the architecture reduces to changing the microprograms only, neither requiring any synthesis nor reconfiguration. Similarly, software developers only need provide other microprograms for executing other algorithms on the BBs.

C. Low-level Hardware Adaptation

When executing until the convergence criterion is met, our FPGA-supported CG method requires many more iterations than its dimension, and moreover, the amount of iterations varies, similarly to calculating dot products with OpenMP and dynamic scheduling. To find the root cause of this behaviour, we wrote the intermediate results back to coprocessor memory and checked the values. We could track the cause down to the dot product implementation, which in order to yield pipelining stores partial products until they can be fully accumulated, thus not being deterministic. Ideally, one would integrate an exact dot product implementation [12], [11] because with the data-transfer problem being solved now, the required tight integration of the special unit is achieved. Though, to save area on the FPGA, we only extended the instantiated floating-point cores by 23 and 32 additional mantissa bits, respectively. Both the multiplication and the internal accumulator do now suffer less from rounding and the typical windowing problem now [5]. As Fig. 10 indicates, we can thereby already halve the number of iterations and reduce application time up to 2.47 times.

VI. Conclusions and Outlook

Today, there exist several approaches to port scientific, floating-point-based applications onto reconfigurable logic. In this article, we propose a microprogrammable architecture together with data-driven execution of the building blocks (BBs) to conveniently and efficiently employ FPGA-based accelerators in scientific computing. The BBs on the FPGA work internally data-parallel and thereby optimally exploit the available bandwidth, especially with regard to the memory controllers on the Convey HC-1 coprocessor. Task-level

Table I. RESOURCE USAGE OF THE SINGLE-INSTANCE CONVEY HC-1 DESIGN ON A XILINX VIRTEX-5 LX330 ("BLOCKED VARIANT").

Resource	Slices	LUTs	Registers	DSPs	BRAMS	Max. Freq.
Number	34520	89447	86379	110	166	6.643 ns
Usage	66 %	43 %	41 %	57 %	57 %	150.5 MHz

Table II. AVERAGE EXECUTION TIMES OVER AT LEAST 5 RUNS AND SPEEDUP FOR DIFFERENT SCHEDULING OF THE DMA OPERATIONS WITH VARIYNG PROBLEM SIZES.

	320	512	640	1024	2048	4096
Time (ms)						
Original	517.7	2269.8	4425.9	16558.8	145459.4	1171274.3
DMA early	518.7	2269.3	4425.6	16558.9	145478.9	1171063.9
DMA late	517.9	2270.9	4424.8	16557.3	145457.7	1171091.2
Speedup						
DMA early	0.99805	1.00021	1.00007	0.99999	0.99987	1.00018
DMA late	0.99967	0.99953	1.00025	1.00009	1.00001	1.00016

parallelism is achieved by executing several units concurrently, called subsequently from a control unit. It enables fully leveraging the massively parallel FPGA hardware. A central buffer set allows streaming data from DMA units through potentially several BBs, termed pipeline parallelism, and finally writing back to coprocessor memory so that data reuse is maximized and transfer from/to memory minimized. For a given application domain such as solving equations, a set of BBs needs to be developed once. Domain specialists are then freed from creating reconfigurable designs because they only need to develop the microprograms for other algorithms to be executed on the framework.

The architecture was implemented and tested on two different systems. Evaluation of a microprogram for a preconditioned Conjugate-Gradient method on only a single FPGA against an OpenMP implementation running with 24 hardware threads already yielded slight speedup of 1.2× [10]. We showed its ability to quickly try optimizing execution time by changing the microprogram only. No additional time-consuming synthesis runs were required. Further, we extended the dot product unit by additional bits for more accurate results to model the case that special-purpose units are integrated into the domain-specific BB set. Thereby, we could decrease the number of required iterations of our test case. The benefit of embedding special arithmetic units, e.g. for very exact arithmetics, should be evaluated in future.

We have thereby also developed a novel methodology. From an application programmer's perspective, development starts with connecting function blocks, e.g, in a regular C program, from a software library that is formulated in a data-driven way and whose functions can execute concurrently, i.e, in a task-parallel fashion. This program can already serve as a highly-performant program version because first investigations showed that data-driven, task-parallel execution can provide competitive performance to data-parallel execution due to enhanced data locality. Having mapped the algorithm onto function blocks for the software version, the microprogram must be formulated to connect the building blocks via the central buffer set for execution in reconfigurable accelerator hardware. The microprogram only needs to be compiled, then the algorithm implementation can already be tested in our architecture framework. With both a software and a hardware implementation, the programmer can easily and stepwise run distinct parts of the application on the accelerator hardware. The data transfer problem when using acclerators is solved by exploiting data flow principles, and therefore the programmer can efficiently and conveniently benefit from accelerator hardware. We envision that automatic design tools will create such microprograms instead of generating huge state machines in low-level hardware languages.

REFERENCES

[1] B. Bomar, "Implementation of microprogrammed control in FPGAs," *IEEE Trans. on Industrial Electronics*, vol. 49, no. 2, pp. 415–422, April 2002.

[2] F. de Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *19th International Conference on Field Programmable Logic and Applications*. IEEE, August 2009.

[3] D. DuBois, A. DuBois, T. Boorman, C. Connor, and S. Poole, "An Implementation of the Conjugate Gradient Algorithm on FPGAs," in *16th Int. Symp. on Field-Programmable Custom Computing Machines*. IEEE, April 2008, pp. 296–297.

[4] T. Feist, "Vivado Design Suite," Xilinx, White Paper Version 1.1, June 2012. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp416-Vivado-Design-Suite.pdf

[5] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, March 1991.

[6] Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient hardware code generation for FPGAs," *ACM Trans. on Architecture and Code Optimization*, vol. 5, no. 1, pp. 1–26, 2008.

[7] S. Kestur, D. Dantara, and V. Narayanan, "SHARC: A streaming model for FPGA accelerators and its application to Saliency," in *Design, Automation Test in Europe*, March 2011, pp. 1–6.

[8] J. S. Kim, L. Deng, P. Mangalagiri, K. Irick, K. Sobti, M. Kandemir, V. Narayanan, C. Chakrabarti, N. Pitsianis, and X. Sun, "An automated framework for accelerating numerical algorithms on reconfigurable platforms using algorithmic/architectural optimization," *IEEE Trans. on Computers*, vol. 58, no. 12, pp. 1654–1667, 2009.

[9] G. Morris, "Floating-Point Computations on Reconfigurable Computers," in *DoD High Performance Computing Modernization Program Users Group Conference*. IEEE, 2007, pp. 339–344.

[10] F. Nowak, I. Besenfelder, W. Karl, M. Schmidtobreick, and V. Heuveline, "A data-driven approach for executing the cg method on reconfigurable high-performance systems," in *Architecture of Computing Systems – ARCS 2013*, ser. Lecture Notes in Computer Science, vol. 7767. Springer Berlin Heidelberg, 2013, pp. 171–182.

[11] F. Nowak and R. Buchty, "A tightly coupled accelerator infrastructure for exact arithmetics," in *Architecture of Computing Systems – ARCS '10*, ser. LNCS, vol. 5974. Springer, February 2010, pp. 222–233.

[12] F. Nowak, R. Buchty, D. Kramer, and W. Karl, "Exploiting the htx-board as a coprocessor for exact arithmetics," in *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA2009)*, February 2009, pp. 20–29.

[13] F. Philipp and M. Glesner, "(GECO)^2: A graphical tool for the generation of configuration bitstreams for a smart sensor interface based on a Coarse-Grained Dynamically Reconfigurable Architecture," in *22nd International Conference on Field Programmable Logic and Applications*, ser. FPL '12, 2012, pp. 679–682.

[14] D. Singh, "ImplementingFPGA Design with the OpenCL Standard," Altera Corporation, White Paper, November 2013. [Online]. Available: http://www.altera.com/literature/wp/wp-01173-opencl.pdf

[15] R. Strzodka, "Hardware Efficient PDE Solvers in Quantized Image Processing," Dissertation, University Duisburg, 2004.

[16] W. Vanderbauwhede, S. Chalamalasetti, S. Purohit, and M. Margala, "A Few Lines of Code, Thousands of Cores: High-level FPGA Programming using Vector Processor Networks," in *Int. Conf. on High Performance Computing and Simulation*. IEEE, July 2011, pp. 561–567.

[17] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: an infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[18] Xilinx, "9 Reasons Why The Vivado Design Suite Accelerates Design Productivity." [Online]. Available: www.xilinx.com/publications/prod_mktg/vivado/Vivado_9_Reasons_Backgrounder.pdf