Karlsruhe Institute of Technology

Employing a High-Level Language for Porting Numerical Applications to Reconfigurable Hardware

V. Heuveline, W. Karl, F. Nowak, M. Schmidtobreick, F. Wilhelm

No. 2011-13

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.emcl.kit.edu

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
No. 2011-13

Impressum

Karlsruhe Institute of Technology (KIT)
Engineering Mathematics and Computing Lab (EMCL)

Fritz-Erler-Str. 23, building 01.86
76133 Karlsruhe
Germany

KIT – University of the State of Baden Wuerttemberg and
National Laboratory of the Helmholtz Association

Employing a High-Level Language for Porting Numerical Applications to Reconfigurable Hardware

Mareike Schmidtobreick[1], Florian Wilhelm[1], Fabian Nowak[2], Vincent Heuveline[1], Wolfgang Karl[2]

[1] Engineering Mathematics and Computing Lab (EMCL)
at Karlsruhe Institute of Technology (KIT)
[2] Institute of Computer Science and Engineering (ITEC)
at Karlsruhe Institute of Technology (KIT)

firstname.lastname@kit.edu

Kaiserstraße 12
76131 Karlsruhe
Germany

Abstract. The deployment of FPGAs has become more and more common over the last years. Many applications have since then been accelerated by porting advantageous parts onto FPGA hardware. High-level, C-like programming languages and advanced tools such as Impulse CoDeveloper that produce hardware descriptions can potentially help with this task. We showcase the applicability of this new approach to FPGA acceleration in terms of solving the Poisson equation with the conjugate gradient (CG) method and a red-black symmetric successive over-relaxation (SSOR) preconditioner as a model problem. In this case, the CPU executes the CG method while an FPGA takes over the red-black SSOR preconditioning part. We compare a purely CPU-based algorithm to our FPGA-extended approach in order to evaluate the maturity and applicability of high-level language translators with regard to accelerating numerical applications.

Keywords: Reconfigurable Hardware, FPGA, Numerical Methods, Preconditioner, Impulse CoDeveloper, Accelerator

1 Introduction

One cannot emphasize too much on the importance of numerical methods to solve socially relevant problems. Solving such increasingly complex problems requires much computational power in terms of speed and parallelism. Traditionally, these requirements are met with even larger clusters of commodity hardware based on x86 CPU design. However, most scientific software does not scale linearly with the number of processors, resulting in a decrease of efficiency with an increase of CPUs.

Nowadays, scientists reconsider the multi-purpose approach of a CPU, meaning they become aware of the fact that a CPU is falling behind other technologies when it comes to a special niche of applications. To properly exploit modern multi-core processors, special-purpose software that focuses on parallelizing applications by annotations (cmp. OpenMP) can be employed. In contrast, special-purpose hardware can be used, of which the most lately renowned are GPUs and Cell processors. GPUs are built to offer a high degree of parallelism and fixed function units that perform special tasks, often related to 3D calculations, with high efficiency. Today, the usage of GPUs as accelerators for certain parts of numerical programs is an overly accepted method to speed up execution, e.g. as preconditioners [3] or even entire solvers [7].

Another special-purpose hardware approach is to let the programmer build their own parallel function units, e.g. scalar product, according to the special needs of any application from any domain. The technology providing this is termed *Reconfigurable Hardware* and has gained more attention over the last years (although it's much older than GPGPU technology), but has not been considered the same breakthrough as GPU-based accelerators in scientific computing yet. The properties of reconfigurable hardware like Field-Programmable Gate Arrays (FPGAs) are intriguing as they can be configured to adopt any arbitrary circuit design. Hence, FPGAs leave it up to the programmer to decide which and how many special-purpose function units are needed. A well-known example of the possibilities resulting from this is the Smith-Waterman algorithm that performs local sequence alignment of proteins in biotechnology. The speedup of Smith-Waterman on an FPGA [12] is up to 100x compared to a CPU implementation because of its special needs that a generic CPU does not satisfy sufficiently. The main reason for the hesitation of scientists to adapt to this technology is the challenge of implementing an algorithm. Commonly, an algorithm is implemented in a comparatively low-level description language like Verilog and VHDL. Synthesis tools further process this description of the algorithm to finally configure the FPGA. These languages follow a different programming paradigm, namely implicit parallelism and explicit sequentiality. Hence, programming hardware is error-prone, time-consuming and not feasible for most engineers and mathematicians. Recent advancements however allow implementing an algorithm by means of higher-level programming paradigms based on C, C++ or Java, which is converted to hardware descriptions that can be synthesized by proprietary vendor tools afterwards.

We therefore study the applicability of this high-level language approach to FPGA programming and the interplay of CPU and FPGA for numerical applications from a mathematician's point of view. As exemplary application, we implement a preconditioner for the CG method [9], which is not only one of the most time-consuming functions but is also highly suited to automatic parallelization, using a high-level C-based language for gaining an FPGA implementation. Section 2 provides the mathematical background for our model problem and also presents the rationale for a symmetric successive-over-relaxation preconditioner with red-black ordering. After a quick overview of the state of the art in C-based hardware development in Section 3, we present our implementation and benchmarking results of the hardware-assisted preconditioned CG algorithm. Section 4 summarizes our work and points out future perspectives.

2 Numerical Background

The maybe most well-known problem in numerics is to solve the Poisson equation that occurs in electrostatics and mechanical engineering. This makes it a perfect first candidate to be approached with a new technology. Let $\varOmega \subset \mathbb{R}^2$ be an open and bounded domain and let $f : \varOmega \to \mathbb{R}$, $f \in C(\varOmega)$ be a given function. A function $u : \bar{\varOmega} \to \mathbb{R}$, $u \in C^2(\varOmega) \cap C(\bar{\varOmega})$ is to be found that satisfies

$$-\varDelta u = f \quad \text{in } \varOmega, \tag{1}$$

where $\varDelta := \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. We further demand homogeneous Dirichlet boundary conditions $u = 0$ on $\partial\varOmega$ and set $\varOmega = (0,1)^2$ for simplicity. We discretize our domain $\varOmega$ by an equidistant grid with parameter h

$$\varOmega_h = \{(x,y) \in \varOmega \mid x = k \cdot h, y = l \cdot h, (k,l) \in \mathbb{Z}^2\},$$

and approximate $-\varDelta$ by means of finite differences

$$-\varDelta u = \frac{-u_{j+e_1} - u_{j-e_1} + 4u_j - u_{j+e_2} - u_{j-e_2}}{h^2} + O(h^2), \tag{2}$$

where $u_{j+e_i} := u(x_j + he_i)$ and e_i denotes the i^{th} unit vector. We first apply a lexicographical ordering to the grid points, i.e. starting at one corner of the grid and numbering the nodes consecutively. Then we multiply Equation (2) with h^2 and obtain a matrix A_h with block structure

$$A_h = \begin{pmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & T \end{pmatrix}, \quad T = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix}, \quad I = \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix}, \tag{3}$$

and a corresponding right-hand side $b_h(x_j) = h^2 \cdot f(x_j)$. As a result of the sparsity pattern in (3) we can express A_h as the well-known five-point stencil expressing Equation 2, illustrated in Figure 1. Additionally, A_h has the advantage that it is symmetric and positive definite. For solving this kind of linear system, the *conjugate gradient (CG)* method is the best known iterative technique [11]. The number of necessary iterations that the CG method requires for reaching a good approximation to the solution depends on the condition number $\kappa(A)$ through the relation

$$\frac{\|e^{(k)}\|_A}{\|e^{(0)}\|_A} \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k, \tag{4}$$

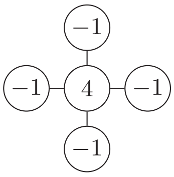

Fig. 1. Five-point stencil of the Poisson problem (1) discretized by finite differences on an equidistant grid.

Algorithm 1 Preconditioned Conjugate Gradient method [2]. Iterations are denoted with lower indices.

```
$r_0 = b - Ax_0$
$z_0 = M^{-1} r_0$
$p_0 = z_0$
for $k = 0, 1, \dots, k_{max}$ do
    $\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$
    $x_{k+1} = x_k + \alpha_k p_k$
    $r_{k+1} = r_k - \alpha_k A p_k$
    $z_{k+1} = M^{-1} r_{k+1}$
    if $r_{k+1}^T z_{k+1} < TOL$ then
        exit loop
    end if
    $\beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}$
    $p_{k+1} = z_{k+1} + \beta_k p_k$
end for
```

where $e^{(k)} = x^{(k)} - x$ is the error in the k^{th} iteration and $\| \cdot \|_A$ the energy norm. This inequality justifies the application of a preconditioner M where $\kappa(M^{-1}A) \ll \kappa(A)$. Conclusively, fewer iterations are necessary to solve the equivalent system $M^{-1}Ax = M^{-1}b$. In our case, M needs to be symmetric and positive definite in order to sustain these properties for the CG method. Algorithm 1 shows pseudo code of the CG method with preconditioning.

2.1 The Symmetric Successive Over-Relaxation Preconditioner

An often applied preconditioner for CG is a descendant of the Successive-Over-Relaxation (SOR) method. As a member of the class of splitting methods, SOR relies on the matrix splitting $\omega A = (D + \omega L) - ((1 - \omega)D - \omega U)$, where D is the diagonal of A, L its strict lower part, $U = L^T$ its strict upper part and ω a relaxation parameter with $\omega \in (0, 2)$. An iteration scheme is then given by

$$(D + \omega L)x^{(k+1)} = \left((1 - \omega)D - \omega L^T\right) x^{(k)} + \omega b, \tag{5}$$

that solves the given equation system under the same premises as the CG method with the fulfilled additional requirement that all diagonal entries are positive. Given the sparsity pattern of our matrix (5), this can be easily translated to a stencil formulation. The drawback of the scheme is that the left-hand side of

Algorithm 2 Red-black symmetric Gauss-Seidel (SSOR with $\omega = 1$) as applied in line 8 of Algorithm 1.

```
for all $z_i$ in red points do
    $z_i = (r_i + (r_{j+e_1} + r_{j-e_1} + r_{j+e_2} + r_{j-e_2})/4)/4$
end for
for all $z_i$ in black points do
    $z_i = r_i + (z_{j+e_1} + z_{j-e_1} + z_{j+e_2} + z_{j-e_2})/4$
end for
```

(5) enforces the calculation of $x^{(k+1)}$ by a serial forward substitution. Using a red-black ordering of the unknowns remedies this drawback so that unknowns with the same color are decoupled from each other as illustrated in Figure 2. This ordering allows the parallel calculation of unknowns with the same color, thus making it a perfect candidate for execution on a highly parallel system like an FPGA.

An SOR preconditioner is then simply defined as one SOR iteration with the starting vector $x^{(0)}$ chosen to be the null vector and the right-hand side $b = r$. The preconditioner is applied in line 8 of the CG Algorithm 1. To achieve the necessary symmetry, which is violated by the left-hand side in (5), we update consecutively two times with reversed ordering of the unknowns the second time to get a symmetric SOR (SSOR) preconditioner, formally

$$M^{-1} = \omega(2-\omega)(D + \omega L^T)^{-1} D (D + \omega L)^{-1}. \tag{6}$$

In the case of a red-black ordering, the best relaxation parameter ω is known to be 1, which renders the SSOR a symmetric Gauss-Seidel method [1]. Equation (6) can then be simplified to gain Algorithm 2.

3 C-based FPGA Programming

Due to the qualification of FPGAs as accelerators, already a multitude of floating-point algorithms [8] and numerical solvers [10] have been ported onto reconfigurable hardware. Since creating FPGA designs is a very tedious task, intensive research has gone into hiding the technical low-level details of implementation. A suitable approach is to provide a toolbox of elementary operations on an FPGA

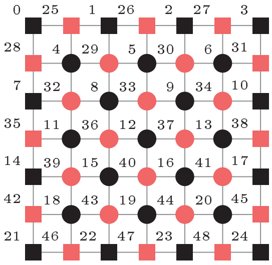

Fig. 2. Example of a red-black ordering. The rectangles denote border values of 0.

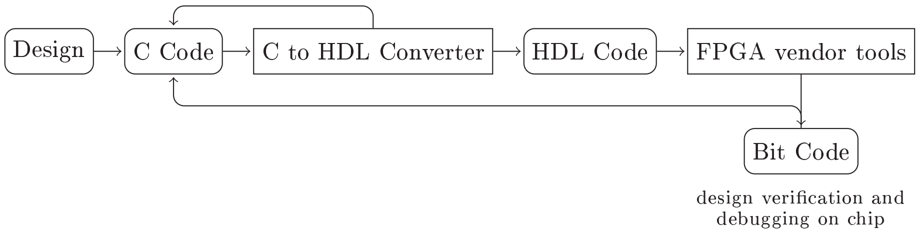

Fig. 3. Hardware design workflow of an application design in C code that is converted to VHDL which is then used to configure an FPGA.

as a library that can be accessed by a high-level language [4,5]. Although using such a library is fairly easy and requires no deep knowledge about FPGA programming, the application is limited to the operations provided by the library. Another more flexible approach is to let the programmer design an algorithm in a high-level language and to convert it into a synthesizable Hardware Description Language (HDL) like VHDL or Verilog. Tools provided by FPGA vendors then process this HDL code to configure the hardware according to the design. Figure 3 sketches this process. In this work, we investigate how the latter approach, namely by using the toolchain of Impulse CoDeveloper Version 3.6, performs in solving our model problem with the help of an FPGA accelerator. ROCCC, another C to HDL compilation framework, might also pose a valid choice; however it does not pose the wanted off-the-shelf approach because there is no so-called backend available to attach the generated code to our employed hardware.

On the hardware side we use the Accelium Coprocessor System (AC2030). This is a product of DRC Computer Corporation consisting of a quad-core AMD Opteron processor 2350 and a Reconfigurable Processing Unit (RPU) attached via HyperTransport at 400 MHz. It is placed on a free Opteron socket and contains a Xilinx Virtex-5 LX 330 FPGA. The RPU holds its own Reduced Latency Dynamic RAM (RLDRAM) with a size of up to 512 MB. Detailed specifications can be found in [6].

3.1 Impulse C

As a high-level language, Impulse C uses the syntax of C, but instead of the C-typical procedural paradigm it employs the *communicating sequential processes paradigm* (CSP). This results in concurrently running software and hardware processes talking to each other over streams or shared memory. While this paradigm only requires setting up a stream for input and output data on the host side, on the accelerator side it demands the processing of the data to be in a more or less sequential fashion. Accordingly, random data access is not possible. Streams can transport arbitrarily sized integer values, fixed-point data or floating-point numbers. The FPGA-side memory is used by software processes as well as by the hardware processes on the FPGA to exchange data.

Source code in Impulse C needs to follow a strict scheme. In a source file for

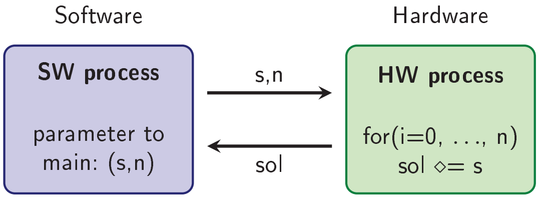

Fig. 4. Repeatedly executing an elementary operation $\diamond \in \{+, *, /\}$.

programming the hardware, the programmer declares functions that are then executed as *hardware processes*. The analogue applies to a software source file where one defines functions that become *software processes*. In a configuration function, inside the hardware source file, all processes are setup to use the formerly defined functions and to communicate by virtue of signals, streams and shared memory. The `main` function resides in the software file and is mostly intended to initialize the architecture with `co_initialize` that calls the configuration function and to start the software and hardware processes with `co_execute`. According to this scheme, the actual algorithm takes place via the interaction between hardware and software processes. To demonstrate the CSP programming paradigm and to find out about the potential of our FPGA, we implemented benchmarks of elementary mathematical operations.

In the first benchmark, a software process sends a single-precision floating-point number s and an amount n over the stream interface to the hardware process, which in return applies n times a given operation $(+, \cdot, /)$ to s. The result is then communicated back to the software process over another stream as illustrated in Figure 4. We use this setup to find out the time a single mathematical operation on the FPGA needs. Since we can only measure the time between sending s and n and receiving the solution, besides the runtime needed for the n operations, the measured runtime includes communication time. Consequently, the hereof calculated time for a single operation includes $1/n$ the runtime of two communications. By increasing n we can asymptotically eliminate the communication time, as shown in Table 1. From our results, we can see that roughly after one million operations the portion of communication time vanishes.

Analyzing the result and considering that the FPGA is running at a clock rate of 100 MHz (10 ns clock period), we can conclude that a floating-point addition or multiplication in a `for` loop takes at least 4 clock cycles (0.05 µs = 50 ns) and in the case of a division at least 28 clock cycles. For the remaining worst-case estimations, we will therefore round to 5 cycles for an addition and to 29 cycles for a division. We performed the same tests for integers and even for an empty loop body. The very satisfying result was that both an integer addition in a `for` loop and an empty `for` loop need 2 clock cycles. The reason is that the configured circuit for this algorithm on the FPGA concurrently executes the addition while also performing the counter increment and the evaluation of the conditional jump in the `for` loop. This kind of instruction level parallelism is uncommon to a standard CPU where the overhead of a loop operation would be clearly visible, and hence the tools and the FPGA itself look promising so far.

Table 1. Timing results of an n times performed operation on a floating point number in microseconds executed simultaneously by 1, 4 and 8 hardware processes.

	1 HW	**1 HW**	**1 HW**	**4 HW**	**8 HW**
number of	**process**	**process**	**process**	**processes**	**processes**
operations	**time per**	**time per**	**time per**	**time per**	**time per**
	add	*mult*	*div*	*add*	*add*
100	1.543333	1.410000	1.693333	2.056667	2.812500
10,000	0.064400	0.063500	0.300633	0.026533	0.026700
1,000,000	0.049885	0.052089	0.288621	0.012576	0.006407
100,000,000	0.049742	0.049742	0.288497	0.012437	0.006219

As our first benchmark executed sequentially one operation after another due to data dependencies, we are not exploiting the possibility of a nearly arbitrary number of parallel processes in hardware, which is only restricted by the physical size of the FPGA. Hence, we implemented a second benchmark that executes a floating-point addition n times on 1, 4 and 8 hardware processes. Ideally, on k hardware processes the runtime should decrease to $1/k^{\text{th}}$ the time of a single process. Table 1 shows the results of these tests. Looking at the last row of this table, we can see that the asymptotically ideal speedup is achieved for 4 and 8 hardware processes. The parallelization was done manually by copying the function of the hardware process in order to get up to 8 hardware processes. Then we had to consistently wire one software process to many hardware processes via one input stream and one output stream per process.

We encountered that all functions that are called by a hardware process need to be API functions or specially defined primitive functions. These are annotated with an Impulse C pragma and define distinct hardware elements that can be called by hardware processes. Only `void`, `int` and `float` are valid return parameters, which prohibits to return matrices or pointers to such.

3.2 Implementation of Symmetric Gauss-Seidel Preconditioner

We first implemented the preconditioned CG Algorithm 1 on the CPU in plain C. This CG implementation calls either a software preconditioner on the CPU or an FPGA-implemented Symmetric Gauss-Seidel (SGS) preconditioner as in Algorithm 2. We decided to not implement the entire CG method inside an Impulse C software CPU-side process because this covers the use case that an accelerated preconditioner needs to be integrated into existing software. Therefore, the Impulse C software process acts only as a proxy between the CG implementation and the Impulse C hardware process with the SGS preconditioner. Furthermore as a result of the red-black scheme that exposes few data dependencies, the preconditioner is a suitable candidate for automatic parallel execution. Figure 5 illustrates the workflow of our program. Before the first iteration, the software process transfers the values of the five-point stencil to the hardware process where they are stored as coefficients in registers. This allows us to use the same code for other five-point stencil coefficients. In each iteration, CG passes the residual to a preconditioner function that invokes the entire architecture with

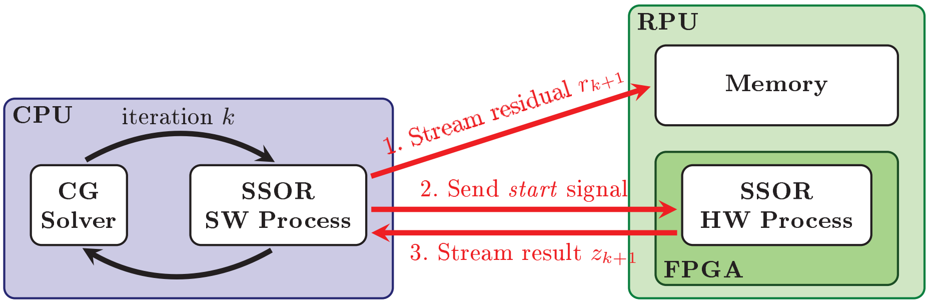

Fig. 5. Flow chart of a CG solver which calls a preconditioner that is implemented on an FPGA. The current iteration number is denoted by k.

`co_execute`. The software process then copies the residual to the FPGA memory and sends a signal to the hardware process after completion because there is no automatism until now to map the required arbitrary memory access to streams. Hereon, the FPGA performs the SGS operations on the residual which is extended by a boundary halo and therefore avoids unnecessary switch statements to distinguish between inner and boundary points. The treated elements are directly streamed back to the software process. After the last element has been transferred, the software process terminates and CG continues.

Employing the concept of the boundary halo can already be considered a minor aspect of hardware-awareness as it both saves hardware resources and keeps the pipeline structure simple. It should also be mentioned that we are using single-precision floating-point throughout the whole program. Although Impulse C does indeed allow for double precision, we only present a first study on the general applicability of C-to-HDL tools instead of a high-performance implementation.

A-priori performance estimation of the FPGA implementation. We now estimate the theoretical time consumption of a straight-forward implementation. The transport of one floating-point number is achieved at one clock cycle of the 400 MHz HyperTransport interconnect, i.e. $t_{transport} = 2.5ns$. (Additional latency of the HyperTransport is negligible for these amounts of data.) Processing one stencil requires 5 random-access data fetches from RLDRAM, 4 additions, 1 or 2 divisions, and 1 write-back to host memory. This results in a pipeline length of $4\,\text{adds} * 5\,\text{cycles} / \text{add} + 2\,\text{divs} * 29\,\text{cycles} / \text{div} = 78\,\text{cycles}$, with an instruction issue rate of 1 instruction every 29 cycles due to the non-pipelined division. In return, this can hide the memory fetches, i.e. one stencil completing every $29 * 10ns = 290ns$ when running at 100 MHz. The time $t_{writeback} = 2.5ns$ for writing back the results can also be hidden except for the very last datum for which we also have to account clearing the entire pipeline with $78-1$ cycles. For an $n \times n$ matrix, $t_{overall} = n^2 t_{transport} + n^2 * 290ns + 77 * 10ns + t_{writeback} = n^2 * 2.5ns + n^2 * 290ns + 770ns + 2.5ns$ approximates the execution time, which is for our 4000×4000 case $t_{overall} = 4680ms$. Potential for optimization by Impulse C lies in pipelining the division, caching previously used data, and exploiting

Table 2. Runtime in seconds of a software and FPGA-based SGS preconditioner for different refinement levels. The FPGA-supported preconditioner performs 13x below our expectation.

Refinement level	Time of SGS on CPU	Time of SGS on FPGA	Expected time on FPGA	Ratio real to expected time on FPGA
500×500	0.003518	0.974298	0.073	13.32
1000×1000	0.013678	3.880566	0.292	13.26
2000×2000	0.055825	15.60444	1.170	13.34
4000×4000	0.257733	61.91578	4.680	13.23

data-level parallelism by instantiating several pipelines until the implementation becomes memory-bound.

CPU and FPGA performance measurements. To give a fair comparison of the performance of the preconditioner on the FPGA, we implemented the same algorithm as a software function with red-black ordering and sequential processing on a single core. The application was compiled with gcc and -O2. We then solely measured the runtime of the software and hardware preconditioner. Table 2 shows the results of these benchmarks for different refinement levels h. The maximum to-be-expected throughput of a naïve implementation is roughly 20 times less than the processing time on the CPU with a 20 times higher clock rate; though still without exploiting any additional parallelism apart from pipelining and without any caching.

FPGA performance analysis. The poor real performance of the FPGA in comparison to the a-priori estimation and to the CPU has several causes. First, we transfer the residual r_{k+1} to the RPU's memory before the actual calculation starts. However, from the above formula we can see that this transfer only accounts for $2.5ns/290ns = 0.0086$ per element. Secondly, the FPGA is performing with 1/20 of the CPU's clock rate. Thirdly, it needs to separately load each stencil operand from RPU memory into FPGA registers without help of a deep memory hierarchy in contrast to a CPU that employs caches. Fourth, the number of states in the state machine of the stencil is in the quite high range of 40 to 80, with each state lasting between 1 (loop increment) and 29 cycles (division). This high number of executed states potentially indicates that all the operations are only executed one after the other, wasting much room for optimization. So even without caching, the resulting hardware design is rather compute-bound than memory-bound because 29 cycles for the division would leave enough room to read 5 stencil data and write the result. Fifth, computations are not arranged in a tree-based, pipeline-suitable order. Our efforts to allow easy parallelization with the help of the red-black ordering scheme did not automatically yield any notable parallelization because only one pipeline was created automatically as the resource consumption report of the place&route steps of the FPGA vendor

Table 3. Resource consumption of the SGS method on Virtex-5 LX330.

Resource	Consumption	Ratio
Number of DSP48Es	47 out of 192	24,00%
Number of RAMB36SDP_EXPs	54 out of 288	18,00%
Number of Slice Registers	41574 out of 207360	20,00%
Number of Slice LUTS	43767 out of 207360	21,00%
Number of Slice LUT-Flip Flop pairs	56843 out of 207360	27,00%

toolchain indicates in Table 3 (a simple test application revealed that the management infrastructure of the RPU system already accounts for more than 35K of the slice registers and more than 45K slice LUT-flip flop pairs). Seemingly, the Impulse Compiler does not exploit this implicitly, and explicit loop-unrolling via a pragma proved not to be possible with dynamic loop boundaries.

As manual parallelization like in Subsection 3.1 proved too error-prone, we did not further investigate in splitting the preconditioner into several hardware processes that concurrently work on distinct data sets. Adding to this is the fact that our target platform does only provide two RLDRAM interfaces, thus only allowing concurrent memory access of two hardware processes.

Table 2 also shows remarkably well, that our implementation on the FGPA scaled better with increasing refinement levels than the CPU implementation. This seems to be due to the streaming model and is of special interest with regard to the ongoing increase in FPGA bandwidth and increasing FPGA frequencies.

4 Conclusion and Perspectives

We showed that with the help of Impulse CoDeveloper it is possible to implement a preconditioner on an FPGA as part of a CG solver on a CPU for a Laplace model problem. This did not require any deeper knowledge of reconfigurable hardware and an HDL. This task was fulfilled in a reasonable amount of time, incomparably shorter than it would have taken using an HDL. The actual performance results (approximately 13.4 times below what could be expected) are not good enough to consider it a valid approach to design an accelerator for numerical applications yet. Until now, hardware-awareness is crucial when targeting FPGA technology, such as exploiting bandwidth and the available FPGA resources while also creating efficient pipeline structures. Hence, only hardware designers rather than high-level programmers can access the full potential of FPGAs. Nevertheless, high-level language to HDL converter technology shows great potential because providing hardware designers with C-like descriptions and the generated hardware description can significantly reduce time to market for high-performance FPGA designs. However, it should be kept in mind that it is still an ongoing field of research, with numerous investigations towards streaming and memory access optimizations currently being undertaken and already today it allows non-hardware developers to easily develop (low-performance) applications for FPGAs. Moreover, the development of the FPGA technology itself is gathering pace and we are looking forward to higher clock rates, faster interconnects and other improvements yet to come, especially since frequency in general-purpose processors has stopped to rise for the sake of more cores on a single die. We are convinced that reconfigurable computing, made accessible to scientists in the field of HPC by high-level languages, has a bright future as an accelerator technology or even processing technology.

5 Acknowledgment

This work arose among others as part of the *Scalable-Earth-System-Models for high productivity climate simulations* (ScalES) project funded by the German Bundesministerium für Bildung und Forschung (BMBF No. 01IH08004E).

References

1. Alefeld, G.: On the convergence of the symmetric SOR method for matrices with red-black ordering. Numerische Mathematik 39, 113–117 (1982)
2. Alefeld, G., Lenhardt, I., Obermaier, H.: Parallele numerische Verfahren. Springer, Berlin (2002)
3. Ament, M., Knittel, G., Weiskopf, D., Strasser, W.: A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In: PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. pp. 583–592. IEEE Computer Society, Washington, DC, USA (2010)
4. Cardoso, J.M.P., Diniz, P.C., Weinhardt, M.: Compiling for Reconfigurable Computing: A Survey. ACM Comput. Surv. 42(4), 1–65 (2010)
5. Curreri, J., Koehler, S., Holland, B., George, A.D.: Performance Analysis with High-Level Languages for High-Performance Reconfigurable Computing. In: 16th International Symposium on Field-Programmable Custom Computing Machines. pp. 23–30 (2008)
6. DRC Computer Corporation: DRC Coprocessor System User's Guide (April 2009), v3.1
7. Göddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P., Turek, S.: Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU. Int. J. Comput. Sci. Eng. 4(4), 254–269 (2009)
8. Herbordt, M., Sukhwani, B., Chiu, M., Khan, M.A.: Production Floating Point Applications on FPGAs. online (July 2009), http://saahpc.ncsa.illinois.edu/papers/Herbordt_paper.pdf, symposium on Application Accelerators in High Performance Computing (SAAHPC'09)
9. Hestenes, M., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards 49(6), 409–436 (December 1952)
10. Lopes, A.R., Constantinides, G.A., Kerrigan, E.C.: A floating-point solver for band structured linear equations. In: ICECE Technology, 2008. FPT 2008. International Conference on. pp. 353–356 (October 2008)
11. Saad, Y.: Iterative methods for sparse linear systems. SIAM, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2. ed. edn. (2003), includes bibliographical references and index
12. Storaasli, O., Yu, W., Strenski, D., Maltby, J.: Performance Evaluation of FPGA-Based Biological Applications. Online (May 2007), proc Cray Users Group'07

Preprint Series of the Engineering Mathematics and Computing Lab

recent issues

No. 2011-12 Vincent Heuveline, Gudrun Thäter: Proceedings of the 4th EMCL-Workshop Numerical Simulation, Optimization and High Performance Computing

No. 2011-11 Thomas Gengenbach, Vincent Heuveline, Mathias J. Krause: Numerical Simulation of the Human Lung: A Two–scale Approach

No. 2011-10 Vincent Heuveline, Dimitar Lukarski, Fabian Oboril, Mehdi B. Tahoori, Jan-Philipp Weiss: Numerical Defect Correction as an Algorithm-Based Fault Tolerance Technique for Iterative Solvers

No. 2011-09 Vincent Heuveline, Dimitar Lukarski, Nico Trost, Jan-Philipp Weiss: Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs

No. 2011-08 Vincent Heuveline, Dimitar Lukarski, Jan-Philipp Weiss: Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method

No. 2011-07 Thomas Gengenbach, Vincent Heuveline, Rolf Mayer, Mathias J. Krause, Simon Zimny: A Preprocessing Approach for Innovative Patient-specific Intranasal Flow Simulations

No. 2011-06 Hartwig Anzt, Maribel Castillo, Juan C. Fernández, Vincent Heuveline, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ortí: Optimization of Power Consumption in the Iterative Solution of Sparse Linear Systems on Graphics Processors

No. 2011-05 Hartwig Anzt, Maribel Castillo, José I. Aliaga, Juan C. Fernández, Vincent Heuveline, Rafael Mayo, Enrique S. Quintana-Ortí: Analysis and Optimization of Power Consumption in the Iterative Solution of Sparse Linear Systems on Multi-core and Many-core Platforms

No. 2011-04 Vincent Heuveline, Michael Schick: A local time–dependent Generalized Polynomial Chaos method for Stochastic Dynamical Systems

No. 2011-03 Vincent Heuveline, Michael Schick: Towards a hybrid numerical method using Generalized Polynomial Chaos for Stochastic Differential Equations

No. 2011-02 Panagiotis Adamidis, Vincent Heuveline, Florian Wilhelm: A High-Efficient Scalable Solver for the Global Ocean/Sea-Ice Model MPIOM

No. 2011-01 Hartwig Anzt, Maribel Castillo, Juan C. Fernández, Vincent Heuveline, Rafael Mayo, Enrique S. Quintana-Ortí, Björn Rocker: Power Consumption of Mixed Precision in the Iterative Solution of Sparse Linear Systems

No. 2010-07 Werner Augustin, Vincent Heuveline, Jan-Philipp Weiss: Convey HC-1 Hybrid Core Computer – The Potential of FPGAs in Numerical Simulation

No. 2010-06 Hartwig Anzt, Werner Augustin, Martin Baumann, Hendryk Bockelmann, Thomas Gengenbach, Tobias Hahn, Vincent Heuveline, Eva Ketelaer, Dimitar Lukarski, Andrea Otzen, Sebastian Ritterbusch, Björn Rocker, Staffan Ronnås, Michael Schick, Chandramowli Subramanian, Jan-Philipp Weiss, Florian Wilhelm: HiFlow^3 – A Flexible and Hardware-Aware Parallel Finite Element Package

The responsibility for the contents of the working papers rests with the authors, not the Institute. Since working papers are of a preliminary nature, it may be useful to contact the authors of a particular working paper about results or caveats before referring to, or quoting, a paper. Any comments on working papers should be sent directly to the authors.