Shallow Water Waves on a Deep Technology Stack: Accelerating a Finite Volume Tsunami Model Using Reconfigurable Hardware in Invasive Computing

Alexander Pöppl[1(✉)], Marvin Damschen[2], Florian Schmaus[3], Andreas Fried[2], Manuel Mohr[2], Matthias Blankertz[2], Lars Bauer[2], Jörg Henkel[2], Wolfgang Schröder-Preikschat[3], and Michael Bader[1]

[1] Department of Informatics, Technical University of Munich, Boltzmannstraße 3, 85748 Garching bei München, Germany
{poeppl,bader}@in.tum.de
[2] Department of Informatics, Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe, Germany
{damschen,fried,manuel.mohr,lars.bauer,henkel}@kit.edu, matthias.blankertz@student.kit.edu
[3] Department of Computer Science 4, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Martensstr. 1, 91058 Erlangen, Germany
{schmaus,wosch}@cs.fau.de

Abstract. Reconfigurable architectures are commonly used in the embedded systems domain to speed up compute-intensive tasks. They combine a reconfigurable fabric with a general-purpose microprocessor to accelerate compute-intensive tasks on the fabric while the general-purpose CPU is used for the rest of the workload. Through the use of *invasive computing*, we aim to show the feasibility of this technology for HPC scenarios. We demonstrate this by accelerating a proxy application for the simulation of shallow water waves using the *i*-Core, a reconfigurable processor that is part of the invasive computing multiprocessor system-on-chip. Using a floating-point custom instruction, the entire computation of numerical fluxes occurring in the application's finite volume scheme is performed by hardware accelerators.

Keywords: Invasive computing · High Performance Computing
Tsunami simulation · Reconfigurable processor
Resource-aware computing

1 Introduction

General-purpose graphics processing units (GPGPUs) and accelerator cards such as the Intel Xeon Phi have brought heterogeneity to today's High Performance Computing (HPC). While these accelerators focus on general-purpose

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 676–687, 2018.
https://doi.org/10.1007/978-3-319-75178-8_54

computations to provide benefits for a wide range of applications, the emerging application-specific accelerators like Google's Tensor Processing Unit [16] or Microsoft Catapult [20] offer an additional performance increase at a reduced power consumption. In contrast to HPC, application-specific accelerators are used commonly in the domain of embedded systems in the form of application-specific integrated circuits (ASICs), application-specific instruction-set processors (ASIPs) or reconfigurable architectures [27]. The latter combine the performance and power consumption benefits of application-specific accelerators with the applicability of general purpose architectures by employing a reconfigurable fabric (FPGA) that can be flexibly configured to host application-specific accelerators at runtime. Accelerator cards featuring a reconfigurable fabric ("fabric" hereafter) have been used in HPC before. However, such a *loose coupling* of CPU and fabric introduces high latencies between accelerators and computations on the CPU, thus impairing the performance benefits. In the embedded systems domain, *reconfigurable processors* are a well-researched architecture that couples a CPU and a fabric on the same chip. This gives accelerators direct access to the CPU-internal state, a so-called *tight coupling*. Therefore, reconfigurable processors provide acceleration with a low latency (of few CPU cycles) and provide a performance benefit even when accelerating computations of only a few hundred cycles.

Our contribution is an integrated demonstration of a reconfigurable HPC system consisting of custom hardware, operating system, compiler, and application. We employ *invasive computing* [26], which allows us to program our system in a *resource-aware* way: The applications can explore available resources at runtime and allocate them exclusively for the duration of an upcoming computation. We first introduce how invasive computing is supported throughout our technology stack. Then, we present our case study of computing shallow water waves on the heterogeneous InvasIC multi-processor system-on-chip (MPSoC). Finally, we detail how we accelerate the shallow water wave computations using *i*-Core, a processor with reconfigurable accelerators that is part of the invasive computing multiprocessor system-on-chip.

2 The Invasive Computing Stack

The governing thought of invasive computing is to grant applications, running on a massively-parallel computer, temporary exclusive access to resources like processor, communication channels and memory [9,26]. A set of granted resources is called a *claim*. Applications allocate claims by *invading* resources, and then *infect* them with a program to run. Finally, the application *retreats* from its claim, freeing the resources.

Realizing this programming model requires support from the hardware architecture, the operating system, the compiler and the application. Figure 1 shows a high-level overview of the invasive computing technology stack providing that support. Its components will be introduced in the following.

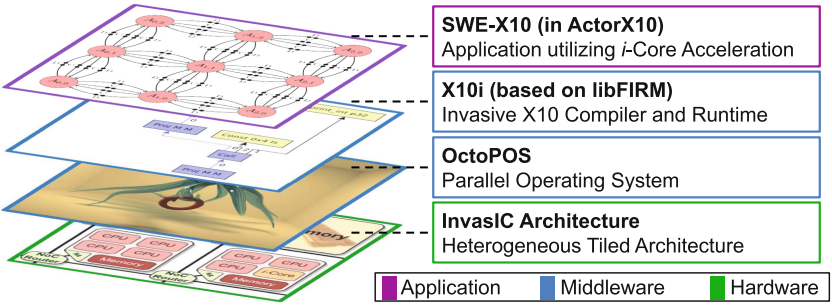

Fig. 1. High-level overview of the invasive computing technology stack. It targets challenges to support invasive computing at the architectural, runtime/compiler and programming level.

2.1 The InvasIC Hardware Architecture

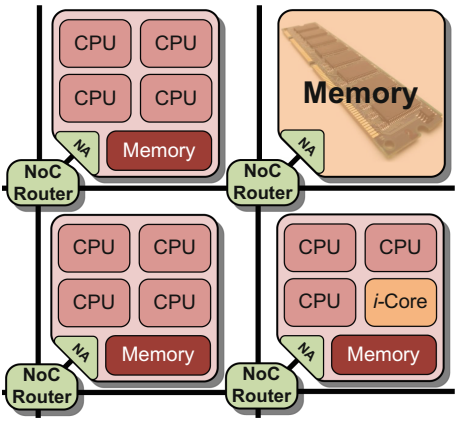

Fig. 2. Overview of the InvasIC hardware architecture used in our work

InvasIC [14] is a heterogeneous Multiprocessor System-on-Chip (MPSoC). It consists of tiles of different types that are interconnected using a network-on-chip (NoC). Within this work, we employ three types of tiles for the Shallow Water Equations (see Sect. 3). (i) RISC tiles contain several RISC cores that communicate over a shared bus, (ii) *i*-Core tiles contain RISC cores and an *i*-Core, a RISC core with reconfigurable hardware accelerators that are accessible through instruction-set extensions (see Sect. 4) and (iii) memory tiles that provide DDR memory. The hardware architecture used in this work is shown in Fig. 2. The RISC cores within these tiles are LEON3 CPU cores (available as part of the Gaisler GRLIB [11]) that implement the SPARC V8 ISA. Each core on a tile has dedicated L1 data and instruction caches. Additionally, the cores on a tile share an L2 cache and a tile local memory (TLM). The TLM is a freely accessible, low-latency and high-throughput scratchpad memory. All tiles are able to access larger amounts of memory (compared to the TLM) provided by memory tiles, and additionally the TLMs of other tiles. This tile-external memory is accessed through a network adapter (NA) providing access to the invasive Network on Chip (*i*NoC) and cached by the L2 cache. Further details on the architecture can be found in [14,26].

While the *i*-Core offers a strict superset of the LEON3's functionality, and may hence be used just like a normal LEON3, special care has to be taken when features unique to the *i*-Core are used: (i) An application can store intermediate state that depends on the *i*-Core so that parts of the further execution need to be scheduled on the *i*-Core (ii) Using accelerators, *i*-Cores can process much more computations than the LEON3 cores in the same amount of time. Simply accessing global memory during these computations leads to memory being the performance bottleneck. We detail challenges (i) and (ii) in Sects. 4.1 and 4.2, respectively.

2.2 The Invasive Operating System – OctoPOS

OctoPOS [19] is a parallel operating system (POS) for the invasive programming paradigm. It was designed and tailored to run on systems with 1000+ cores and therefore implements a non-traditional threading scheme: Instead of long-running threads, parallelized control flows are represented as short snippets of code called *i*-lets. Similar to fibers [25], *i*-lets use cooperative scheduling and mostly run to completion. The exclusive access to resources combined with the mostly-run-to-completion property of *i*-lets relieves us from the requirement of temporal isolation through preemption. This in turn avoids frequent context switches. A run-to-completion *i*-let leaves no state on the stack upon termination, which allows OctoPOS to recycle the used stack for the next *i*-let. Hence, a single stack can be used by multiple *i*-lets. This approach makes them lightweight and inexpensive to create, schedule, and dispatch when compared to traditional threads.

The cooperative scheduling is based around a synchronization primitive called *signal* which is a *private semaphore* [13] implemented in a wait-free [15] manner. When an *i*-let performs a blocking operation, its execution context is saved. This is the only case that necessitates a binding of the *i*-let to its stack.

2.3 The Invasive Language

The invasive hardware platform offers a global address space, but caches are not coherent between tiles. The Asynchronous Partitioned Global Address Space (APGAS) model [23] and its implementation in X10 [24] are a good fit for this use-case. Threads within a single address space partition[1] may freely access each others' memory, while accesses between partitions require the programmer to invoke a special operation[2]. We associate each tile with an APGAS address space partition. Thus, APGAS ensures the separation of cache coherence regions.

To transmit data between partitions, the sender flushes its cache to global memory. The receiver then clones the data into its partition. This offers an API similar to shared memory access to the user program and is more efficient than message passing.

[1] An X10 *place*.

[2] `at`-expression for place-shifting.

We have developed a custom X10 compiler based on libFIRM [7] in order to implement X10 on top of the OctoPOS API, mapping X10's activities directly to *i*-lets [18]. Moreover, we have extended X10 to *Dynamic X10* [6] which supports the dynamic resource changes effected by `invade` and `retreat`.

3 Shallow Water Equations in X10

Shallow Water Equations in X10 (SWE-X10) is a proxy application for the computation of shallow water waves, a model that may be used to predict the propagation of a tsunami wave given the initial water displacement. Shallow water waves are governed by a system of hyperbolic partial differential equations. They are a set of conservation laws for water height (h), and horizontal (hu) and vertical (hv) momenta. Enriched with source terms ($S(x, y, t)$) for bathymetry and Coriolis Forces, they are used to capture not just the propagation of tsunami waves, but also the inundation of coastal regions [8,17].

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = S(x, y, t) \tag{1}$$

Equation (1) displays the shallow water equations. For their numeric solution, we use a finite volume scheme on a Cartesian grid with piecewise constant unknown quantities and an explicit Eulerian time step [17]. We use

$$\begin{aligned} Q_{i,j}^{(\mathrm{n}+1)} = Q_{i,j}^{(\mathrm{n})} &- \frac{\Delta t}{\Delta x}\left(\mathcal{A}^{+}\Delta Q_{i-\frac{1}{2},j}^{(\mathrm{n})} + \mathcal{A}^{-}\Delta Q_{i+\frac{1}{2},j}^{(\mathrm{n})}\right) \\ &- \frac{\Delta t}{\Delta y}\left(\mathcal{B}^{+}\Delta Q_{i,j-\frac{1}{2}}^{(\mathrm{n})} + \mathcal{B}^{-}\Delta Q_{i,j+\frac{1}{2}}^{(\mathrm{n})}\right) \end{aligned} \tag{2}$$

to calculate the new values for the unknown quantities h, hu, hv and b in cell (i, j) at time step $n + 1$, $Q_{i,j}^{(n+1)}$ based on the values of the previous time step. To this end, we need to determine the fluxes of unknown values into and out of each cell for each of the cell's borders. In Eq. (2), this is reflected by $\mathcal{A}^{\pm}\Delta Q_{i\pm\frac{1}{2},j}^{(\mathrm{n})}$ and $\mathcal{B}^{\pm}\Delta Q_{i,j\pm\frac{1}{2}}^{(\mathrm{n})}$ for the vertical and horizontal fluxes, respectively. These fluxes can be computed by solving the Riemann problem at the cell boundary. SWE-X10 includes several approximate Riemann solvers. Here, we focus on the fWave solver [3] that we accelerate using the *i*-Core.

SWE-X10 is written in X10 using the ActorX10 framework [21,22]. Figure 3 depicts a high-level overview of the actor graph. Using actors, we are able to parallelize the application while avoiding data races and without having to distinguish between shared and distributed memory. Each actor is assigned a single patch of the overall grid, and data between patches is exchanged using channels. The actor uses a *patch calculator* to compute the updates for the grid points of a patch. By employing resource-aware programming (see Sect. 2.3), we show how specialization of the patch calculator enables support for hardware accelerators so that each instance fully exploits the available resources.

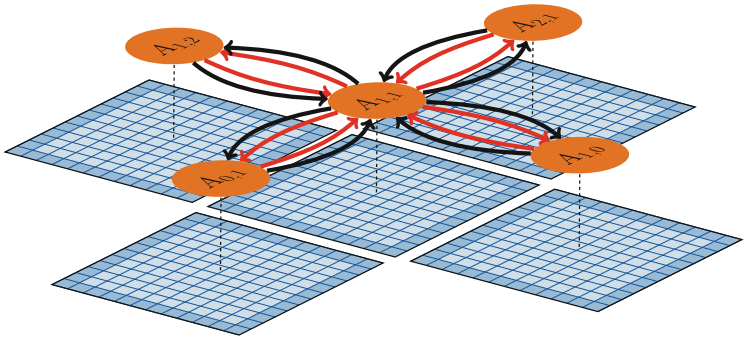

Fig. 3. Grid and actor graph. Five actors (orange) are shown, together with their respective patches (blue). Between each two neighboring actors there are four channels, one pair for simulation data and another for coordination data. (Color figure online)

4 Accelerating SWE-X10 Using *i*-Core

The *i*-Core is a runtime-reconfigurable processor, i.e., it combines a processor core (here: LEON3) with application-specific hardware accelerators. In contrast to application-specific processors (ASIPs), hardware accelerators are not fixed at design time. Instead, they can be reconfigured – even at runtime – to accelerate any given application by the use of a reconfigurable fabric (FPGA). Hardware accelerators are utilized by so-called *custom instructions* (CIs) that extend the ISA of the processor core. A CI invokes execution of a microprogram on the *CI Execution Controller*. Using the microprogram, the CI Execution Controller takes care of data transfers between accelerators and accelerator execution. Thus, a CI can utilize, potentially in parallel, one or more accelerators. The microprogram implementing a specific CI is obtained by scheduling the CI's data flow graph (representing the computations performed by the CI) onto accelerators that are available on the reconfigurable fabric in a specific configuration (see [5] for details). CIs can read inputs from the CPU register file and write results back to it (*tight coupling* of the reconfigurable fabric). A CI can access the whole memory hierarchy through the CPU's cache controller. Additionally, the reconfigurable fabric is directly connected to the TLM using two 128-bit-wide memory ports with a single cycle latency. Therefore, the TLM provides a much higher bandwidth for CIs than accessing the 32-bit-wide system bus. The protocol of invoking CIs from the CPU pipeline is similar to invoking multicycle instructions such as integer division from the standard ISA (Fig. 4).

As the invasive computing paradigm guarantees isolation of resources between applications, each application can adapt the *i*-Core and configure application-specific hardware accelerators that provide maximum benefits for the respective application (in terms of performance but also non-functional properties like worst-case execution time [12]). For accelerating compute-intensive floating-point-based applications like SWE-X10, we introduce a set of pipelined floating-point accelerators that implement generic floating-point operations as

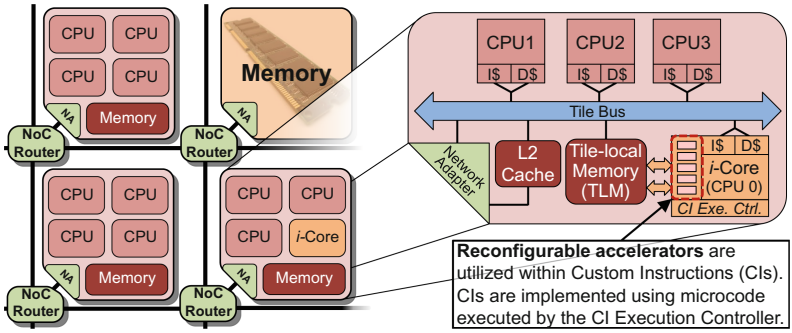

Fig. 4. Overview of the InvasIC architecture with a detailed view of the *i*-Core tile

Table 1. Pipelined floating-point accelerators available for *i*-Core. CIs can utilize multiple accelerators in parallel. Thus, configuring highly-utilized accelerators multiple times can benefit a CI's latency.

Accelerator	Operations	Min./Max. latency[1]	Initiation interval
`FP_MAC`	Add/subtract, multiply, multiply-accumulate	3/5	2
`FP_DIV`	Divide, reciprocal	6/6	2
`FP_SQRT`	Square root	5/5	2
`FP_UTIL`	Min/max, absolute and compare (<, >)	3/3	2

[1] Clock cycles on the reconfigurable fabric

listed in Table 1 (details on a previous version of `FP_MAC` can be found in [4]). To accelerate SWE-X10, we implemented the fWave solver as a CI (`fwave`) for *i*-Core. The `fwave` instruction performs all 54 floating-point operations of the fWave solver as a single CI using our floating-point accelerators. This results in a data-flow graph that consists of 97 nodes (operations) including memory accesses, address generation, communication between accelerators and accelerator execution. On our current *i*-Core prototype within the InvasIC architecture, we instantiate *i*-Core using five reconfigurable containers. We utilize these containers for SWE-X10 to configure the following accelerators: 2×`FP_MAC`, 1×`FP_DIV`, 1×`FP_SQRT` and 1×`FP_UTIL`. The reconfigurable fabric needs to be configured once at application startup, which takes ca. 5.5 ms at a reconfiguration bandwidth of 100 MB/s. This configuration enables us to schedule the 97 operations of `fwave` onto the accelerators in a microprogram consisting of 41 steps. Pipelining is very beneficial for `fwave`: when disabling it, the number of steps almost doubles (>71 steps). Each step of the microprogram takes 2 clock cycles (at maximum 100 MHz) on the reconfigurable fabric. In total, the 54 floating-point operations of the fWave solver are executed in 82 cycles using `fwave` and our pipelined floating-point accelerators on *i*-Core.

CIs like `fwave` are provided to the X10 programmer using wrapper methods that are inlined by the compiler. Thus, we can directly access the CIs from X10 with minimal overhead.

4.1 Adaptions to the OctoPOS Operating System

To maximize the utilization of the available resources on the InvasIC architecture, the OctoPOS scheduler has to be able to schedule i-lets over CPU cores that feature instruction set extensions. More specifically, the instruction set of the LEON3 is a strict subset of the instructions provided by the i-Core. As a consequence, i-lets that rely on the availability of i-Core CIs have to be scheduled on an i-Core that is configured accordingly, as the invocation of the CI would cause an *illegal instruction* trap otherwise. i-lets that only contain standard SPARC-V8 instructions can be executed on i-Cores as well.

We therefore allow i-lets to be assigned to a team which may have a different *scheduling domain* than non-team members. The scheduler ensures that team members are only executed on cores belonging to the team's scheduling domain. Unlike the original team concept [10], i-lets can be dynamically (re-)assigned to a team. This enables the dynamic pinning of i-lets to a set of cores. An application is thus able to create scheduling domains that only contain its invaded i-Cores. By pinning i-lets containing CIs to such a scheduling domain, it is ensured that those i-lets do not trap, while other i-lets are still scheduled on all available cores.

4.2 Adaptions in SWE-X10

SWE-X10 only required very minor code changes to make it compatible with the APIs exposed by the invasive X10 compiler, and therefore most of the work was spent optimizing the performance on i-Core. In SWE-X10, the computational hotspot is the calculation of fluxes between cell boundaries, $\mathcal{A}^{\pm}\Delta Q^{(\mathrm{n})}_{i\pm\frac{1}{2},j}$ and $\mathcal{B}^{\pm}\Delta Q^{(\mathrm{n})}_{i,j\pm\frac{1}{2}}$, in Eq. (2). As mentioned in Sect. 3, the code utilizes, amongst others, the fWave approximate Riemann solver to compute these net updates. The aforementioned CI of the i-Core may be used as a drop-in replacement for the X10 implementation of the fWave solver. However, this way the i-Core does not benefit from its high-bandwidth connection to the TLM, but accesses data from global memory.

Therefore, we created a specialized subclass with an implementation of the iteration optimized for the i-Core that buffers data in the TLM. The size of that memory is limited. Thus, it is impossible to retain the entire patch in the TLM. Instead, we load the data row-wise, using a triple buffering scheme with a *previous*, a *current* and a *next* row. The i-let graph for the scheme is shown in Fig. 5. A task depends on all tasks that are connected to it by an incoming edge. The iteration starts by synchronously loading the first two rows into the TLM ($L_{(0)}$ and $L_{(1)}$), followed by the computation of the horizontal fluxes for row 0 ($H_{(0)}$). Now, we perform the loop for rows 1 to N, N being the number of rows in a patch. In each iteration n, we asynchronously load the next row ($L_{(n+1)}$) into memory and perform the vertical flux computations on the

previous and the current row ($V_{(n-1,n)}$). After the computation is completed, we may asynchronously start the write of the previous row back to the global memory ($S_{(n-1)}$) and perform the horizontal flux computation on the current row ($H_{(n)}$). After clearing the previous row ($C_{(n-1)}$) and, in case of $n = N - 1$, writing back the next row ($S_{(n+1)}$), the loop returns to the beginning.

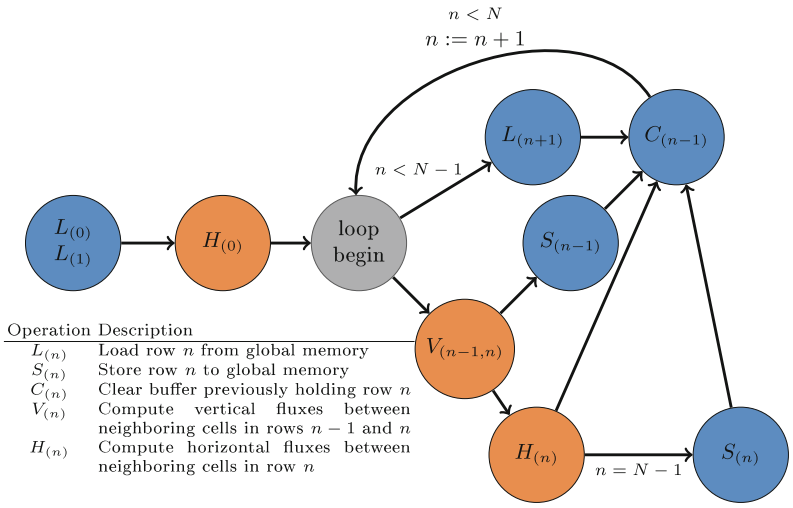

Fig. 5. *i*-let graph for the *i*-Core patch calculator. Nodes that are not (transitively) connected may be executed in parallel. Nodes performing I/O operations are depicted in blue, while nodes performing a computation are depicted in orange. Edges annotated with a condition are only taken if the condition is met. (Color figure online)

5 Results

First, we evaluate the performance benefits and resource utilization on the reconfigurable fabric of executing the fWave solver kernel using the *i*-Core CI compared to execution in software on the LEON3 CPU with different variants of floating-point support. Afterwards, we evaluate the performance of computing one simulation step of a whole patch (see Sect. 3) on the *i*-Core compared to the LEON3 CPU utilizing its high-performance floating-point unit (FPU-HP).

Table 2 shows execution time and resource utilization results for the fWave solver kernel. Compared to a standard LEON3 with FPU-HP (fastest floating-point support variant that also utilizes most resources), *i*-Core is 7.5 times faster and 3.8 times more efficient in the use of lookup tables (LUTs) on the Xilinx Virtex-7 (floating-point operations per second/LUTs).

Table 3 shows the execution time of one iteration of the patch calculators that perform 7140 to 7320 calls to the previously evaluated fWave solver (depending on the patch characteristics). The baseline is execution on the LEON3 with FPU-HP utilizing global memory. Buffering data in the TLM results in a speedup of

Table 2. Execution time and resource utilization results for the fWave solver kernel executed in software (without floating-point unit (FPU), with "lite" FPU and "high-performance" FPU from Gaisler) compared to `fwave` CI on *i*-Core. Results were obtained using GRLIB on a Xilinx VC707 board (Virtex-7 FPGA) at 75 MHz.

	LEON3 – no FPU		**LEON3 + FPU-lite**		**LEON3 + FPU-HP**		***i*-Core**	
Execution Time[1]	[μs]	Speedup	[μs]	Speedup	[μs]	Speedup	[μs]	Speedup
	183.6	1	14.9	12.3	9.8	18.7	1.3	141
Resource Utilization	LUTs	DSPs	LUTs	DSPs	LUTs	DSPs	LUTs	DSPs
	9,103	4	12,756	4	23,949	20	37,658	24
Resource Efficiency	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$	$\frac{\text{FLOPS}}{\text{LUTs}}$	$\frac{\text{FLOPS}}{\text{DSPs}}$
	27.8	73,529	248	906,040	203.8	275,510	780.3	1,730,769

[1] Average over 1024 measurements

1.75×. Execution on the *i*-Core utilizing global memory speeds up the computation by 2×. Both optimizations combined alleviate the memory bottleneck for the *i*-Core and we achieve a speedup of 4.82× in total.

Table 3. Patch calculator execution time on the LEON3 (with FPU-HP) compared to execution time on the *i*-Core, with data in global DDR RAM or buffering in the tile-local memory (TLM). Results were obtained using the InvasIC Hardware Prototype on a Synopsis CHIPit system consisting of four Xilinx XC5VLX330 (Virtex-5 FPGA) at 25 MHz.

	LEON3 – global		**LEON3 – TLM**		***i*-Core – global**		***i*-Core – TLM**	
Execution Time[1]	[ms]	Speedup	[ms]	Speedup	[ms]	Speedup	[ms]	Speedup
	2049	1	1169	1.75	1017	2.01	425	4.82

[1] Average over 200 measurements

6 Related Work

SWE-X10 is based on the C++-application SWE [2,8], a code based on the finite volume scheme described by LeVeque [17]. SWE features a modular approach, with one patch per MPI rank. It has been executed on Xeon CPUs [2], Tesla GPUs [8] and the Xeon Phi [2]. In contrast to SWE-X10, SWE uses a global communication approach, and does not have lazy activation.

ElasticX10 [1] also allows for a dynamic and asynchronous change in the amount of places. Compared to this, Dynamic X10 offers more stability: Places change in a predictable fashion, as the application itself drives the change in resources, i.e., it is *resource aware*, enabling it to maximize its performance.

Compared to other reconfigurable processors [27], *i*-Core has the unique feature that its CIs are not implemented as one monolithic accelerator, but using microcode to utilize multiple accelerators. This enables to implement the same functionality with more or less accelerators and enables to opt for a different tradeoff between CI latency and fabric area allocated at runtime.

In contrast to our work, FPGA accelerators such as Microsoft Catapult [20] are coupled loosely to the CPU, and only effectively speed up large computations. Application specific accelerators such as the Tensor Processing Unit [16] are not reconfigurable.

7 Conclusion

In this contribution, we have demonstrated the applicability of techniques from embedded computing, such as application-specific hardware reconfiguration and control over the entire technology stack, to HPC. Using the *i*-Core's tightly-coupled reconfigurable fabric, we were able to implement an fWave approximate Riemann solver in hardware. Thus, we accelerated the computation of fluxes between cell boundaries, the computational hotspot of SWE-X10, by a factor of 4.82 over the baseline solution using the LEON3's high-performance floating point unit, while utilizing resources on a reconfigurable fabric more efficiently (in terms of LUTs and DSPs). This contribution demonstrates the feasibility of accelerating HPC applications using a reconfigurable processor.

Acknowledgments. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

1. Elastic X10. http://x10-lang.org/documentation/practical-x10-programming/elastic-x10.html. Retrieved 9 May 2017
2. Bader, M., Breuer, A., Hölzl, W., Rettenberger, S.: Vectorization of an augmented Riemann solver for the shallow water equations. In: Proceedings of 2014 International Conference on High Performance Computing and Simulation (HPCS 2014), pp. 193–201. IEEE (2014)
3. Bale, D.S., LeVeque, R.J., Mitran, S., Rossmanith, J.A.: A wave propagation method for conservation laws and balance laws with spatially varying flux functions. SIAM J. Sci. Comput. **24**(3), 955–978 (2003)
4. Bauer, L., Grudnitsky, A., Damschen, M., et al.: Floating point acceleration for stream processing applications in dynamically reconfigurable processors. In: IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), October 2015
5. Bauer, L., Shafique, M., Henkel, J.: A computation- and communication-infrastructure for modular special instructions in a dynamically reconfigurable processor. In: International Conference on Field Programmable Logic and Applications, pp. 203–208. IEEE (2008)
6. Braun, M., Buchwald, S., Mohr, M., Zwinkau, A.: Dynamic X10: resource-aware programming for higher efficiency. Technical report 8, Karlsruhe Institute of Technology (2014). (X10 2014)
7. Braun, M., Buchwald, S., Zwinkau, A.: Firm—a graph-based intermediate representation. Technical report 35, Karlsruhe Institute of Technology (2011)

8. Breuer, A., Bader, M.: Teaching parallel programming models on a shallow-water code. In: Proceedings of 2012 11th International Symposium on Parallel and Distributed Computing, ISPDC 2012, pp. 301–308. IEEE Computer Society (2012)
9. Bungartz, H.J., Riesinger, C., Schreiber, M., et al.: Invasive computing in HPC with X10. In: Proceedings of 3rd ACM SIGPLAN X10 Workshop, X10 2013, pp. 12–19. ACM, New York (2013)
10. Cheriton, D.R., Malcolm, M.A., Melen, L.S., Sager, G.R.: Thoth, a portable real-time operating system. Commun. ACM **22**(2), 105–115 (1979)
11. Cobham Gaisler AB: GRLIB IP library user's manual. Technical report, Göteborg, Sweden, January 2016. Version 1.5.0: http://www.gaisler.com/products/grlib/grlib.pdf. Retrieved 2 May 2017
12. Damschen, M., Bauer, L., Henkel, J.: Extending the WCET problem to optimize for runtime-reconfigurable processors. ACM Trans. Archit. Code Optim. **13**(4), 45:1–45:24 (2016)
13. Dijkstra, E.W.: The structure of the "THE"-multiprogramming system. Commun. ACM **11**(5), 341–346 (1968)
14. Henkel, J., Herkersdorf, A., Bauer, L., et al.: Invasive manycore architectures. In: Proceedings of 17th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 193–200, January 2012
15. Herlihy, M.: Wait-free synchronization. ACM Trans. Prog. Lang. Syst. (TOPLAS) **13**(1), 124–149 (1991)
16. Jouppi, N.P., Young, C., Patil, N., et al.: In-datacenter performance analysis of a tensor processing unit. arXiv preprint arXiv:1704.04760 (2017)
17. LeVeque, R.J., George, D.L., Berger, M.J.: Tsunami modelling with adaptively refined finite volume methods. Acta Numerica **20**, 211–289 (2011)
18. Mohr, M., Buchwald, S., Zwinkau, A., et al.: Cutting out the middleman: OS-level support for X10 activities. In: Proceedings of 5th ACM SIGPLAN X10 Workshop, X10 2015, pp. 13–18. ACM, New York (2015)
19. Oechslein, B., Schedel, J., Kleinöder, J., et al.: OctoPOS: a parallel operating system for invasive computing. In: Proceedings of International Workshop on Systems for Future Multi-core Architectures (SFMA), pp. 9–14. EuroSys (2011)
20. Ovtcharov, K., Ruwase, O., Kim, J.Y., et al.: Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper, vol. 2, no. 11 (2015)
21. Pöppl, A., Bader, M., Schwarzer, T., Glaß, M.: SWE-X10: simulating shallow water waves with lazy activation of patches using ActorX10. In: Proceedings of 2nd International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), pp. 32–39. IEEE, November 2016
22. Roloff, S., Pöppl, A., Schwarzer, T., et al.: ActorX10: an actor library for X10. In: Proceedings of 6th ACM SIGPLAN X10 Workshop (X10). ACM (2016)
23. Saraswat, V., Almasi, G., Bikshandi, G., et al.: The asynchronous partitioned global address space model. Technical report, Toronto, Canada, June 2010
24. Saraswat, V., Bloom, B., Peshansky, I., et al.: X10 language specification, December 2015. Version 2.5: http://x10-lang.org. Retrieved 5 May 2017
25. Tanenbaum, A.S.: Modern Operating Systems, pp. 859–860. Prentice Hall, Upper Saddle River (2009)
26. Teich, J., Henkel, J., Herkersdorf, A., Schmitt-Landsiedel, D., Schröder-Preikschat, W., Snelting, G.: Invasive computing: an overview. In: Hübner, M., Becker, J. (eds.) Multiprocessor System-on-Chip, pp. 241–268. Springer, New York (2011). https://doi.org/10.1007/978-1-4419-6460-1_11
27. Tessier, R., Pocek, K., DeHon, A.: Reconfigurable computing architectures. Proc. IEEE **103**(3), 332–354 (2015)