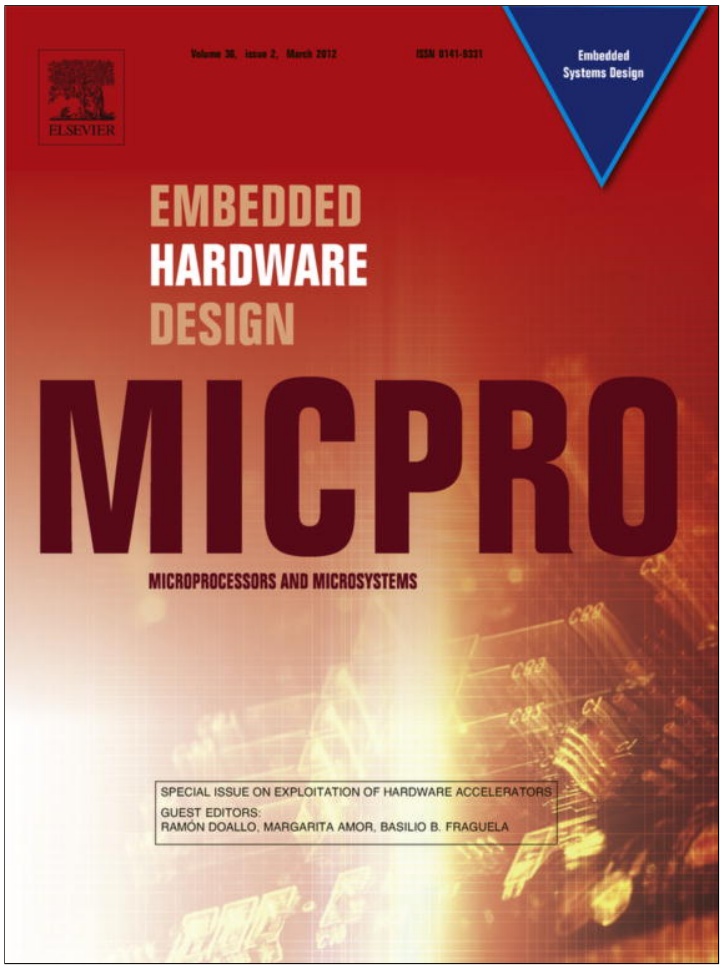

Published in: *Microprocessors and Microsystems*
© Elsevier 2012, DOI: 10.1016/j.micpro.2011.04.002

Microprocessors and Microsystems 36 (2012) 110–126

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

IMORC: An infrastructure and architecture template for implementing high-performance reconfigurable FPGA accelerators

Tobias Schumacher, Christian Plessl *, Marco Platzner

Paderborn Center for Parallel Computing, University of Paderborn, 33098 Paderborn, Germany

ARTICLE INFO

Article history:
Available online 1 May 2011

Keywords:
Reconfigurable computing
*k*th nearest neighbor technique
FPGA

ABSTRACT

The design, implementation and optimization of FPGA accelerators is a challenging task, especially when the accelerator comprises multiple compute cores distributed across CPU and FPGA resources and memories and exhibits data-dependent runtime behavior. In order to simplify the development of FPGA accelerators we propose IMORC, an infrastructure and architecture template that helps raising the level of abstraction. The IMORC development flow bases on a modeling technique for visualizing an application's communication demand and an architecture template that aids the developer in implementing the design. The architectural template consists of a versatile on-chip interconnect with asynchronous FIFOs and bitwidth conversion placed into the communication links, a performance monitoring infrastructure for collecting performance information during runtime and a set of generic infrastructure cores which are frequently needed in accelerator designs. We demonstrate the usefulness of the IMORC development flow by means of the case study of accelerating the *k*th nearest neighbor thinning problem, where IMORC greatly helps us in understanding the communication demand and in implementing the application. With the integrated performance monitoring infrastructure, we gain insights into the data-dependent behavior of the accelerator that helps us in identifying bottlenecks and optimizing the accelerator to achieve a speedup of 10× to 40× over an optimized CPU implementation.

1. Introduction

While research has demonstrated the potential of FPGAs as acceleration technology since decades, the creation of accelerators for realistic workloads has been hindered, at least partially, by the lack of commercially available systems. In the last years, however, computing system vendors began to offer machines that combine microprocessors with FPGAs. More recently, FPGA modules that fit into processor sockets have been introduced, e.g., [1,2], and provide a fairly standardized way of integrating hardware accelerators into mainstream computing systems.

Developing and optimizing accelerators for such machines is a challenge. Even if FPGA cores for important algorithmic kernels become more and more available, combining them into an overall accelerator remains tricky. Generally, the cores will show data-dependent runtimes and compete for shared resources such as external memory or the host interface, which makes it difficult to decide on a proper number of cores, their topology, the degree of core-level parallelism, data partitioning, etc.

This paper extends our previous work on the IMORC infrastructure and architecture template for high-performance FPGA accelerators, which has been introduced first in [3]. IMORC is actually two things, an architectural template for creating core-based FPGA accelerators and an on-chip interconnect. The architectural template assists the designer in combining cores to an overall accelerator and greatly facilitates design space exploration, core reuse and portability. The IMORC interconnect relies on a flexible multi-bus structure with slave-side arbitration and offers FIFOs, bitwidth conversion, and performance monitoring. Especially performance monitoring is indispensable for debugging and optimizing FPGA accelerators. In [4,5] we have demonstrated the potential of IMORC by means of two case studies, an accelerator for the *k*th nearest neighbor thinning problem and an accelerator for a parallel rendering framework.

This paper consolidates our work on IMORC by providing a comprehensive introduction to the IMORC architecture template and a discussion of the platform support for the XD1000 reconfigurable workstation. Further, as a novel contribution of this paper, we present how IMORC fits into an overall accelerator modeling and development process. This modeling approach helps developers in visualizing an application's communication demand and finding an efficient mapping of tasks to hardware. To illustrate the use of our modeling and development process, we apply the approach to a *k*th nearest neighbor thinning application, for which we had introduced a predecessor version in previous work [4].

* Corresponding author. Tel.: +49 5251 605399.
E-mail addresses: tobe@uni-paderborn.de (T. Schumacher), christian.plessl@uni-paderborn.de (C. Plessl), platzner@uni-paderborn.de (M. Platzner).

0141-9331/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2011.04.002

The remainder of this paper is structured as follows. After a discussion of related work in Section 2 we introduce our accelerator modeling and development process in Section 3. In Section 4 we present the IMORC architecture template. Section 5 discusses specific cores that have been developed to support IMORC on the XtremeData XD1000 reconfigurable workstation. In Section 6 we present a case study that demonstrates how the IMORC modeling process is applied to an application and how the application model is translated to an implementation with a sequence of repeated refinement steps into a high-performance FPGA accelerator. In Section 7 we summarize how we have applied the IMORC infrastructure in two other case studies. Finally, in Section 8 we draw conclusions and outline further work.

2. Related work

In this section we discuss related work in areas relevant to IMORC. We first review important on-chip core interconnects, including interconnects available for FPGAs, and then focus on work on performance prediction as well as on in-system performance monitoring and optimization for reconfigurable accelerators.

Common standards for connecting cores on-chip include Whishbone [6] and AMBA [7], which can be used as shared bus but can be also configured to more complex topologies. However, these standards require the cores to communicate with the bus at the same clock frequency. When using cores running at different speeds, designers are forced to create proper wrappers around the cores to resynchronize data. For Xilinx FPGAs, the IBM CoreConnect [8] bus architecture is available which provides several buses: the high-speed processor local bus, the now deprecated on-chip peripheral bus for lower speed communication and the device control register bus. A different communication approach is slave-side arbitration provided, for example, by Altera's Avalon [9] interconnect. Avalon employs one bus per slave core, allows to place cores into different clock domains, and features a performance counter core. In [10], Shannon and Chow introduce the SIMPPL framework for connecting different cores in an FPGA using asynchronous FIFOS, which decouple control and datapath and allow for placing cores in different clock domains. A detailed comparison of several on-chip interconnects is presented by Mitic and Stojcev in [11].

Performance prediction methods for FPGA accelerators usually rely on estimates for the computation times of the different cores and the amount of data transferred and parameters such as clock frequencies, bandwidths, and latencies for accessing memories and the host system, to estimate performance metrics with analytic formulae. An example is given by Steffen [12], who provides a model for determining an applications's suitability for FPGA acceleration by calculating the computational density function of the algorithms. With parameters like the memory size, bandwidth and latency as well as the maximum throughput of the processor the designer can calculate the number of operations performed per second on the abstract processor and compare this value with the measured value on the real processor. In [13,14] Smith and Peterson present an approach that focuses on the expected load imbalance in multi-node systems which is a cluster of workstations, each equipped with one or several reconfigurable accelerators. The model assumes iterative algorithms with several tasks that are executed in parallel on the processors as well as on the reconfigurable hardware. After each iteration, the tasks synchronize. With these properties and parameters like communication bandwidth between nodes, execution time of the tasks in hardware and software and others, the load imbalance between nodes is calculated as well as the complete runtime of the application on the target system. In [15,16] Holland et al. introduce another approach, the Reconfigurable Computing Amenability Test (RAT). The method relies on an analysis of algorithms or existing legacy code and several computations. RAT provides a method for estimating the throughput of an accelerator based on parameters like the interconnect's speed, amount of data to be transferred, the number of operations performed per data element and the clock frequency of the accelerator. Additionally, RAT considers the numerical precision required for an accelerator and its resource usage for analyzing the suitability of an application for acceleration using reconfigurable computers. In [17] the authors present the integration of RAT into the RCML environment for estimation modeling of reconfigurable computing systems [18].

Related work on in-system performance monitoring and optimization includes, for example, [19]. There, Koehler et al. present a framework for performance analysis in high-performance reconfigurable computing. The framework allows for instrumenting HDL code for gathering and analyzing performance values of an application running on reconfigurable hardware during runtime. In [20,21] Curreri et al. expand this work in that they add support for instrumenting high-level synthesis code.

IMORC differs from the related approaches in several ways. Compared to Avalon's slave-side arbitration approach, IMORC shows greater flexibility in forming core topologies and provides built-in support for performance monitoring. Compared to SIMPPL, IMORC again offers more flexibility by supporting data width conversions between cores and providing built-in support for performance counters. In terms of performance prediction, monitoring and optimization, IMORC shares similarities with related approaches in that it also comprises a modeling approach for describing the considered application and support for designing the actual accelerator. However, rather than making any assumptions on the application behavior and on the actual target architecture, IMORC can be applied to all kinds of applications and reconfigurable target architectures. Instead of providing analytic formulae for estimating concrete runtimes, IMORC with its execution and architecture models is also suitable for more complex applications with data-dependent runtime behavior. In addition, IMORC facilitates optimization by integrated instrumentation methods, e.g., performance counters that are automatically inserted into the communication links between cores.

3. Accelerator modeling and development process

An application accelerated with IMORC comprises an architecture model and an execution model. The architecture model captures the structural decomposition of the application into cores and interconnect; the execution model specifies the behavior of the application. Both models can be created at different levels of abstraction, depending on the current stage of the accelerator development process. In this section, we describe the architecture and execution models and provide an overview of typical stages in the IMORC development process.

3.1. The architecture model

The architecture model underlying IMORC is a network of communicating cores. A core represents an execution unit or storage. Cores communicate via a multi-bus on-chip network. For accessing the network, each core provides an arbitrary number of communication ports. Communication ports are either master or slave ports, where links connect master to slave ports. Master ports may connect to multiple slave ports, in which case addressing is required. Conversely, slave ports may connect to multiple master ports through an arbiter.

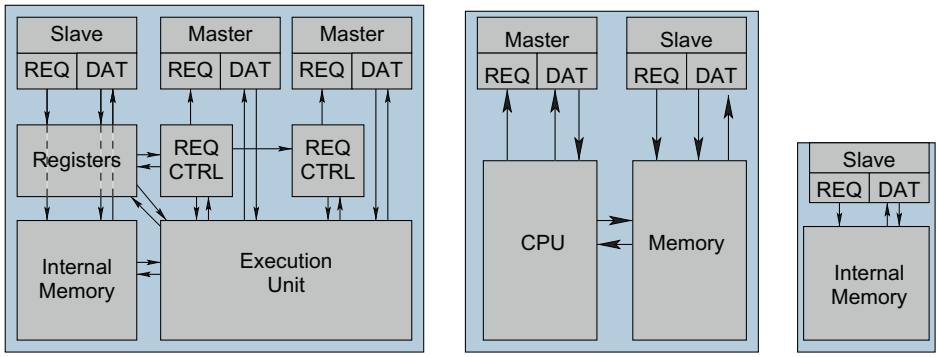

Fig. 1. Block diagram of an example compute core, a host interface core and a memory core.

Fig. 1a shows the block diagram of a typical execution (compute) core. The core comprises a slave and two master ports, each of which is separated into a request channel (REQ) providing control information and a data channel for the actual data to be transmitted. The slave communication port is connected to a block of registers and to internal memory. A communication request controller exists for each of the master ports, and each of them is connected to the register block for retrieving control information. An execution unit is responsible for performing the actual calculations and is connected to the data channels of the master port. If required, the execution unit may also be connected to the register block, the internal memory block and the communication request controllers.

The host processor (Fig. 1b) is modeled as a simplified core that provides, depending on the actual target platform, one or multiple master ports and optionally one slave port. The core comprises a CPU that is able to execute arbitrary tasks and a local memory. The CPU can access this memory and send messages to the master ports, additionally other cores may access the memory using the optional slave port. Alternatively, if the accelerator is modeled without regarding tasks executed on the host CPU, communication with the host processor of the reconfigurable computing system can be performed through a host interface core. The host interface core provides the same interface as the host processor core, but omits the integrated CPU core.

In IMORC, shared memory is modeled as a special kind of core with no execution unit but a large amount of storage. A memory core provides exactly one slave port (see Fig. 1c).

Fig. 2 presents a sample architecture diagram including three compute cores, a host interface core and a memory core. The host interface core can directly access the slave port of compute core 0, which provides two master ports for accessing the slave ports of compute cores 1 and 2. These two cores' master ports access the memory core, and the master port of compute core 2 is additionally connected to the slave port of the host interface core.

3.2. The execution model

The intention of the execution model is to specify an application's behavior. An application splits into a number of communicating tasks, where each task comprises operations that can be grouped into three distinct but possibly overlapping phases:

- *Incoming communication*: A task receives and processes incoming messages, which typically first specify the parameters of the computing job and then contain the data to be processed.
- Local computing.
- *Outgoing communication*: A tasks sends messages to other tasks or responds to its invoking task.

Fig. 3 shows an example for a task. The tall box in the middle represents local operations performed by the task, the smaller boxes on the left and the right side represent communication points with other tasks. Tasks can communicate with other tasks using messages. Messages can be read (`rd`) or write (`wr`), the first kind is used for requesting data from another task, the other one for sending data to another task. Read requests need to be followed by a response message (`resp`). Tasks can be modeled at different levels of abstraction, depending on the actual requirements. On a rather abstract level, the task computations may be described by pseudo code or a code segment in a high-level language. On a detailed level, the computations may be expressed as a sequence of micro operations or RTL code.

Fig. 4 shows a task graph with two tasks accessing a block of shared memory. Modeling memory as separate task simplifies the runtime analysis, especially when multiple tasks access a shared block of memory and therefore contention may occur. Additionally, due to the request-response nature of the communication model, tasks operating on streams of data can be modeled in natural way. The communication subtasks send read requests to the appropriate memory task, and as soon as the data becomes available the local operation subtasks start processing.

Accelerator development in IMORC requires the designer to eventually map the tasks of the execution model to cores of the architecture model. Since IMORC also models host communication and memory access as activities captured by cores, the mapping step actually includes the partitioning between host CPU, FPGA and memories as well as the breakdown of the application's functionality into communicating cores on the reconfigurable fabric.

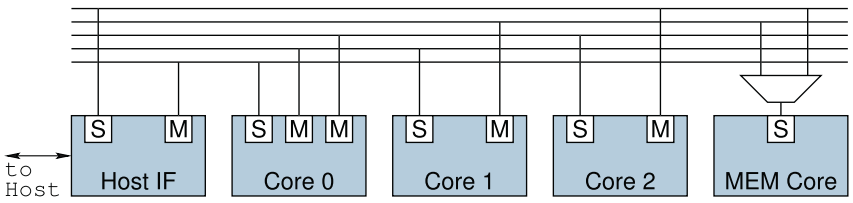

Fig. 2. Sample architecture diagram with three compute cores, one memory core and one host interface core.

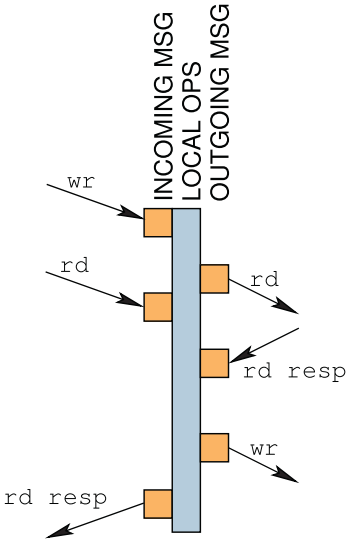

Fig. 3. Diagram of a sample task.

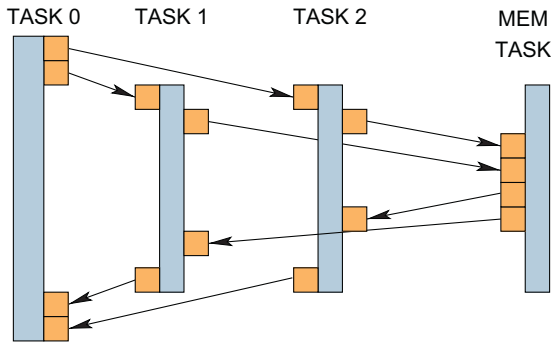

Fig. 4. Task graph with two tasks accessing a memory task.

It has to be noted that the IMORC architecture and execution models are rather general and do not pose any restriction on the behavior of tasks or cores, respectively. For example, there is no need to enforce blocking reads on incoming messages as in the Kahn process network (KPN) model, or to specify rates for the production of outgoing messages as in the synchronous data flow (SDF) model. In contrast to these well-known formal models of computation, IMORC provides more degrees of freedom but misses formally provable characteristics.

3.3. Development flow

Fig. 5 shows the typical IMORC development flow. Accelerator development starts out with partitioning and initially mapping the application to the reconfigurable computing system. In most cases there exists already a software implementation of the original algorithm that can be used as starting point for the partitioning process. To decide which parts of the application should be accelerated, software performance analysis and profiling tools are employed to identify the computationally most intense application parts (kernels). Subsequently, these kernels are mapped to the FPGA. Since storage access times play a major role in many implementations, the application's main data structures are extracted into storage tasks. The storage tasks can then be mapped to the memories available in the system. Fig. 6 demonstrates the partitioning and initial mapping process.

After partitioning and initial mapping a first performance analysis can be performed, for example by estimating the amount of

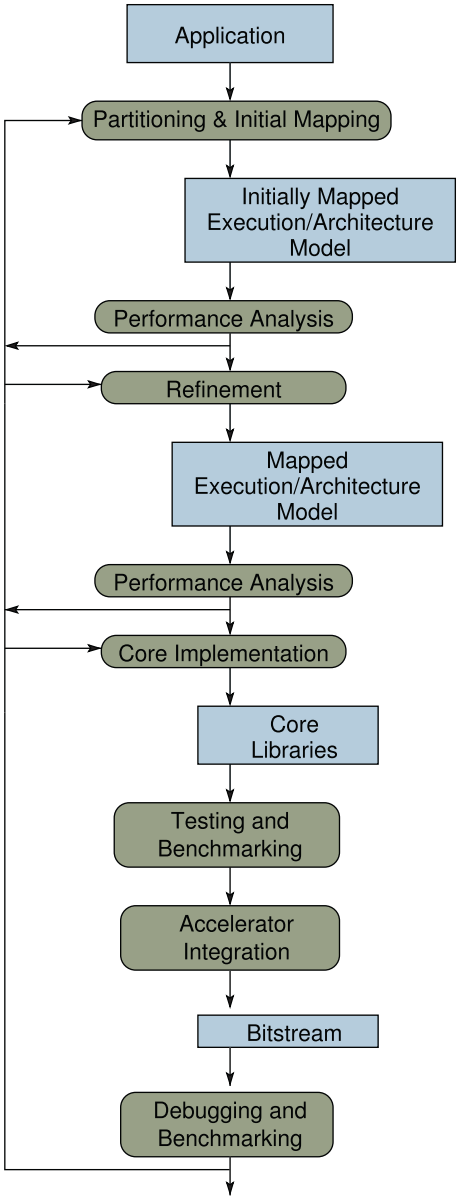

Fig. 5. The IMORC design flow.

data to be transferred to the FPGA and back, or from the FPGA to the memories. We usually perform this step manually and estimate communication times by considering the amount of data to be transferred and the maximum available communication bandwidth. Such basic performance analysis can lead to new mappings which, for example, exclude certain kernels from being mapped to the FPGA or use a different mapping of storage tasks.

In the refinement phase, tasks are broken down to a more detailed level. While for the tasks mapped to the CPU an efficient implementation often exists, tasks mapped to the FPGA have to be split into multiple communicating tasks that actually form specifications for the subsequent circuit design. An essential objective of the refinement step is to extract opportunities for exploiting parallelism. An example for such a refinement is presented in Fig. 7. The original task graph on the left consists of two tasks, a master and a worker. The master on the left sends data to be processed to the worker. When first set of data has been processed, the results are returned and the next set of data to be processed is transferred. In the refined task graph, instead of waiting for the first set of data to be completely processed, data is streamed from

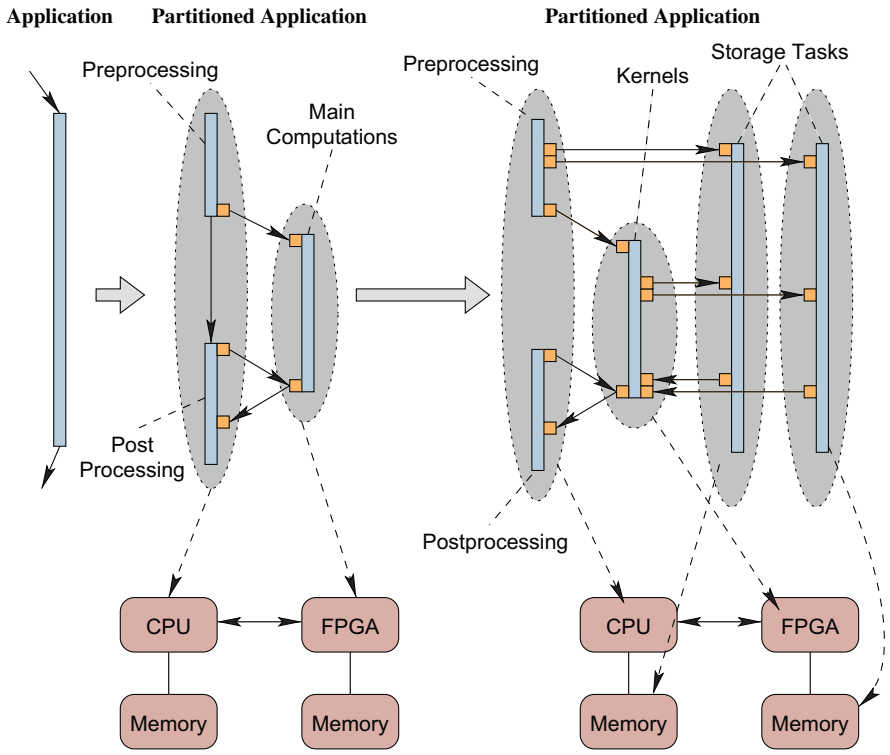

Fig. 6. Partitioning and initial mapping of an application.

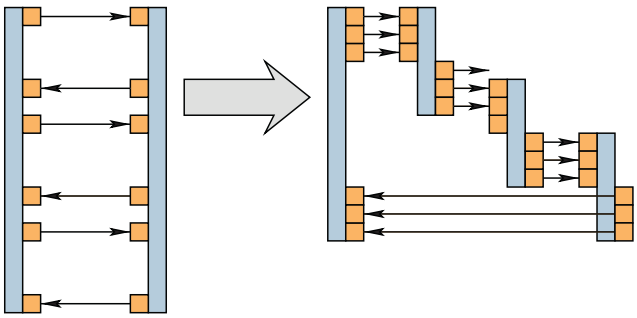

Fig. 7. Example of a refinement by streaming data through a set of tasks.

the master core to the first worker core. This core performs some processing on the data and forwards the results to another worker core and so on. The last worker core then returns the results to the master core.

Another possible refinement is the extraction of the request subtask and the datapath of a task. In many other models, data is assumed to be directly accessible in constant time, so operations are directly applied to data. In the IMORC modeling approach, however, requests for data and processing can be split into separate tasks that get mapped to the same core – tasks requesting data are mapped to the communication controller of a core, tasks operating on the data are mapped to the execution engine of a core (cf. Fig. 1).

Based on the refined execution and architecture models a more detailed performance analysis takes place to get more accurate performance estimations. For example, communication estimation now includes the block sizes for data transfers. If the estimated performance is found sufficient, the core implementation phase starts. For each core of the architecture model a corresponding circuit is created or taken from a core library. The individual cores are tested and benchmarked. In the next step, all cores are integrated with the IMORC multi-bus on-chip network infrastructure and synthesized to the accelerator bitstream. The final accelerator is loaded onto the reconfigurable computer, debugged and benchmarked. The performance monitoring infrastructure of IMORC greatly facilitates the analysis of the accelerator behavior and the identification of bottlenecks. Based on such monitoring data, development iterations can be performed in order to optimize the performance. Eventually the IMORC performance monitoring infrastructure can be removed.

4. IMORC architecture template

As outlined in the previous section, IMORC assumes that an application is decomposed into multiple communicating cores which encapsulate computations as well as access to memory and external communication interfaces. A key element of IMORC is its multi-bus on-chip network for connecting such cores within an FPGA. To achieve high-throughput communication with minimal congestion, IMORC relies on a multi-bus architecture with slave-side arbitration. In this section, we present the basic elements of IMORC and show how accelerators architectures with different network topologies are created.

4.1. Cores, links and channels

IMORC cores access the on-chip communication infrastructure via ports. There exist two types of ports, denoted as master and slave ports. A link between two cores can only be formed by connecting a master with a slave port. IMORC uses unidirectional buses and differentiates between commands (requests) and data transfers. Hence, each link splits into three channels, a request channel (REQ), a master-to-slave channel (M2S), and a slave-to-master channel (S2M). The REQ channel is used for transmitting write or read requests from the master to the slave port. Data is transferred from the master to the slave port via an M2S channel,

and from slave to master port via an S2M channel, respectively. A link must comprise at least the REQ channel and can, additionally, comprise an M2S and an S2M channel.

Fig. 8 details the signals for an IMORC link, basically consisting of a data bus and two handshake signals for each channel. The REQ channel uses a data bus `req` to transmit request packets and the two handshake signals `req_wait` and `req_wr` on the master side or `req_rd` on the slave side, respectively. Table 1 lists the fields available in the request packets. A request packet specifies the command, write or read, the destination address, the number of words to write or read, and an optional user-defined field. The data to be written or read is then transferred over the M2S and S2M channels. The destination address is interpreted by the slave port. For example, a memory controller core will use the destination address to access the attached memory, and a compute core could use it to select between a set of internal registers.

IMORC inserts asynchronous FIFOs into each channel which allows for operating each core at its maximal speed in its own clock domain. Particularly, memory cores can operate at their top speed and, possibly, provide sufficient bandwidth to serve several compute cores at once. Moreover, the additional FIFO storage in the network decouples core execution which, depending on the actual application, can help improve performance.

In case the data bitwidth of master and slave ports differ, IMORC inserts a bitwidth conversion module into the link. The bitwidth conversion modules are placed before the FIFOs, which always have the same bitwidth as the slave ports. The conversion from a wide master word to a small slave word is straight-forward and involves several writes to the FIFO for one word sent by the master. Converting from small master to wide slave words is more complex. In such a case the bitwidth conversion module generates a set of sub-word write enable signals to mask the sub-word within the slave word that is to be written. As a consequence of these bitwidth conversion modules, a compute core with a 32 bit interface can be connected to a memory core with a 64 bit interface, and also to a 256 bit interface without any change in the compute core itself. This greatly facilitates reuse of cores and porting applications to different accelerator platforms. The integration of the FIFOs and the bitwidth conversion module into an IMORC link is shown in Fig. 8.

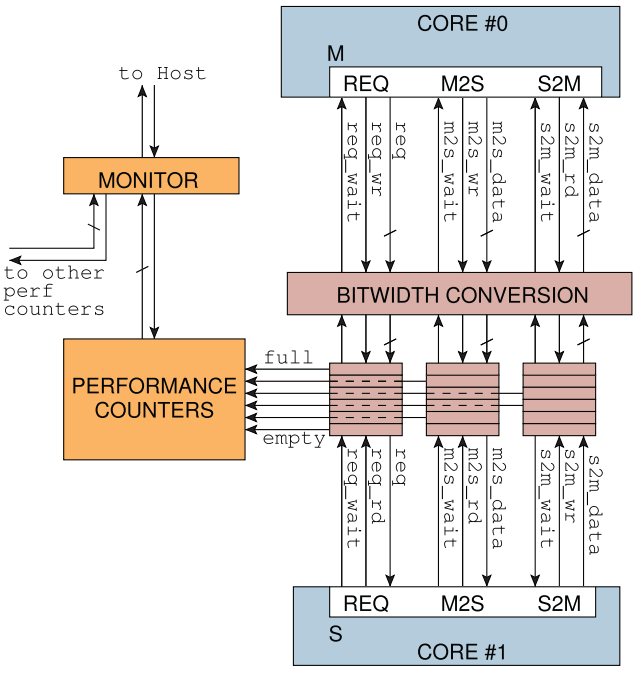

Fig. 8. IMORC link with its channels and signals.

Table 1
Request packet format.

CMD	Command field (RD or WR)
ADDR	Destination address
Size	Number of 32 bit words to transfer
OPT	Optional information Decoding is up to the designer

4.2. Network topology and arbitration

An application typically comprises several IMORC cores connected in a certain topology. Each core can have an arbitrary number of master and slave ports, hence, connecting master and slave ports in a 1:1 fashion is basically sufficient to build arbitrary topologies. However, as this approach requires cores with a rather high number of master and slave ports, IMORC supports 1:*n*, *m*:1, and *m*:*n* connections as well.

A 1:n connection allows a master port to address several slave ports at once. To this end, the signals `req_wr` and `m2s_wr` are turned into vectors. By selecting a subset of these write enable signals, the master may issue multi-cast or even broad-cast request messages and data writes. The wait signals from the individual FIFOs also form a vector and are routed back to the master port. IMORC even supports the S2M channels in a 1:*n* connection with the restriction that a master can address only one FIFO to read from at any time.

An *m*:1 connection allows a slave port to be driven from several master ports. IMORC employs slave-side arbitration and inserts an arbiter module right before the slave port. Fig. 9 displays the IMORC interconnect for a 2:1 connection. In this figure, the optional bitwidth conversion modules are omitted for the sake of readability. The arbiter contains FIFOs for REQ, M2S, and S2M channels from each master port. A selector module SEL in the REQ channel decides which request is served next. By default the selection is done in round-robin manner but can easily be changed as needed, for example to introduce priorities. Once a request is being processed, the corresponding M2S or S2M channel in the arbiter is activated to read or write data, respectively. If required, *m*:1 and 1:*n* patterns can be combined to form an *m*:*n* connection.

4.3. Monitoring infrastructure

Optimizing the performance of an application consisting of multiple cores is not a trivial task and requires to balance computation speed with communication bandwidth, and to minimize contention for shared resources, e.g., external memory. While the IMORC architectural template offers the designer the freedom to address performance problems, e.g., by increasing data widths, replicating compute cores, replacing a compute core with a higher throughput version, selecting the appropriate remedy requires information about the dynamic behavior of the application. To this end, IMORC provides performance counters attached to the FIFOs within the links and arbiter modules. For each FIFO, the number of full and empty events is counted as shown in Fig. 8. In user-defined time intervals a monitoring core triggers a `sync_and_reset` signal that tells the performance counters to sync the current counter value into a register in the monitoring core's clock domain and to reset the counter. The synchronized values can then be sequentially read by the monitoring core using a shared bus.

In user-defined time intervals the performance counters are read and reset, and the event counts are collected by a monitoring core. Since the cores can operate in different clock domains, the

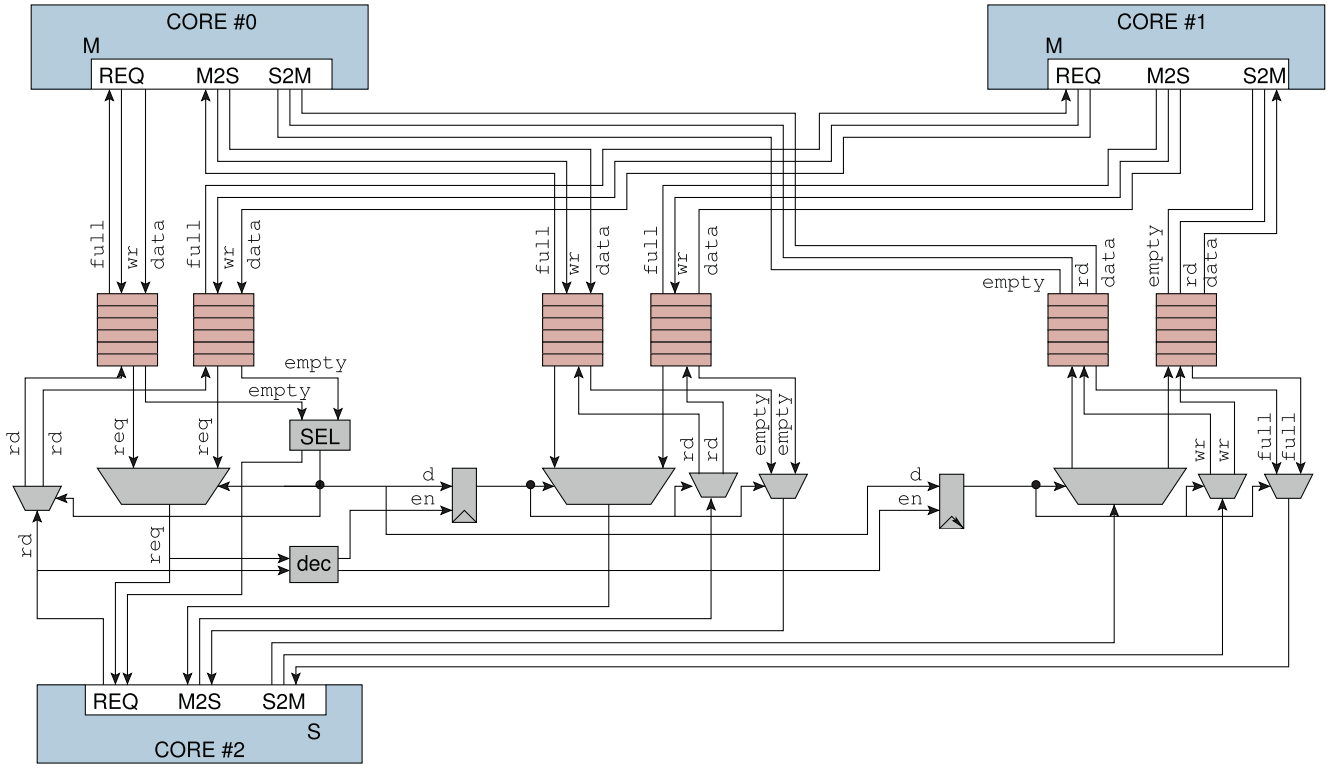

Fig. 9. Arbitration module for a 2:1 connection.

pure number of events is not sufficient to draw conclusions about the overall system. Hence, each performance counter additionally employs a cycle counter that is incremented every clock cycle. Using the performance counter infrastructure, designers can monitor the dynamic behavior of an application and gather information about the cores, e.g., when they start or stop processing and how much bandwidth they produce on the different channels. In our current implementation, the monitoring core is accessed from outside the FPGA via a JTAG port, which is a highly portable approach that has the benefit of not affecting the timing of monitored system.

4.4. Infrastructure cores

Besides the on-chip interconnect, IMORC also provides several infrastructure cores such as memory controllers, a host interface core, IMORC-to-Register interface cores, request generator cores and farming cores. These cores provide functionalities often needed and are available to assist the designer in generating accelerators.

4.4.1. Memory interface cores

IMORC provides memory cores for accessing on-chip, off-chip and host memory. Access to these three kinds of memory is completely transparent to the accelerator cores. From an execution core's perspective, there is no functional difference between on-chip, off-chip and host memory, which greatly eases design space exploration. The IMORC interface to on-chip memory is written completely in synthesizable VHDL and wraps and extends available vendor memory cores. Current vendor implementation tools are only able to infer simple types of on-chip memory from VHDL, such as single port or dual port memory without byte-enable signals. For high speed access wide memories are preferable, making subword writes necessary. Furthermore, IMORC also wraps and extends vendor cores for accessing off-chip memory to allow for an integration of these cores into the IMORC's on-chip network. Many host interfaces, e.g., PCI, PCIe, HyperTransport, support master transfers to host memory. Using an appropriate interface, such memories can be accessed by IMORC. IMORC does not limit the number of memory interface cores that can be used in a system. The maximum number of supported cores basically depends on the number of available off-chip and on-chip memory resources as well as on the number of logic resources available on the target device. Instantiating vendor cores makes the IMORC infrastructure for accessing memory vendor specific.

4.4.2. Request generator cores

Request generator cores are basically pre-defined implementations of the REQ CTRL modules (displayed in Fig. 1a) for certain request sequences. Instantiating and configuring request generators relieves the designer of creating such logic by hand each time. The streaming request generator is a basic module which is configured with a command (read or write), a base address, the amount of data to be transferred and the size of each request. Sending requests can be interrupted, for example when different requests are to be inserted by other request generators. Additionally, the core can be configured with a step parameter to increment the base addresses of the requests by a user-defined value. This is highly useful, for example, for accessing the diagonal elements or a diagonal band of a matrix. A more sophisticated request generator can inject a sequence of read and write requests into a channel. Different base addresses, amounts of data and request sizes can be set for read and write requests. The sequence of reads and writes can be programmed by two further parameters, encoded as bit vectors. The first one represents the initial sequence, that is performed once when the request generator starts. The second sequence is repeatedly executed after the setup phase. Such a setup phase is useful when data is to be read, processed and written back to memory – in this case, the datapath pipeline first gets filled before data is written back. A third example of a request generator posts a

sequence of read and write requests, but does not execute a static sequence. Instead, the request generators monitors the `wr` signal of the M2S channel. When a configurable amount of data is sent to this channel, an appropriate write request is inserted into the sequence of read requests.

4.4.3. IMORC-to-Register/Register-to-IMORC interface cores

Execution cores usually need to be configured with parameters specifying, for example, the start address of data to be processed and the amount of data to be processed. Typically, these parameters are packed into a job message that is to be decoded by the execution core. The IMORC-to-Register (I2R-IF) interface core is an infrastructure core that performs the decoding of job messages. The core provides a configurable amount of internal registers that can be accessed in two ways: First, messages appearing on an IMORC slave interface are decoded and the appropriate read/write operations are performed on the register block. Second, a separate interface provides direct read/write access to the registers. One of the registers can optionally be declared as state register, which blocks the IMORC link when containing a non-zero value. For example, a core can send a job message to such an I2R interface core and set the state register to '1'. Consequently, the I2R interface will not accept any further transfers on the IMORC link. The slave core then starts processing and resets the state register when finished. Now, further job messages can be accepted on the IMORC link. This way, cores sending jobs to other cores are not required to always check if the other core is currently idle – jobs can be sent as long as the corresponding link's FIFOs are not full. Since the state register blocks all requests, the I2R interface will also not respond to any read requests during that time. Since all requests are processed in-order, several job messages following a read will result in a read response on the S2M channel not before all jobs are finished – this way a core can even synchronize with the finalization of the jobs sent to another core. IMORC further inserts a performance counter into the state register of the I2R interface core for monitoring the runtime of the core. Fig. 10 shows a sample core using the IMORC-to-Register interface core. A master core can send a job description to the I2R-IF. The register interface can be directly connected to one or multiple request generator cores, providing a straightforward method for generating the control unit of a core. Additionally, IMORC provides a Register-to-IMORC interface, which also implements a set of directly accessible registers connected to an IMORC link. However, this time the interface is an IMORC master. This interface can for example be used for generating job messages and sending them to a compute core's I2R-interface using an IMORC link.

4.4.4. Farming cores

Besides streaming, farming is another recurring design pattern in parallel programming. Data is partitioned and distributed among multiple processing elements (workers). Each processing element operates autonomously on its own set of data. This design pattern is supported by IMORC by providing a dedicated job scheduler – the farming core. Configuration parameters of the farming core are the format of the job description (number and size of registers) and the number of worker cores. The farming core receives job messages on an IMORC slave port and forwards them to the worker cores. Additional to job messages, the farming core can respond to status requests. Such request messages are useful for probing the execution status of workers and synchronizing with the completion of all worker cores. The request message can be blocking or non-blocking. Non-blocking requests are immediately answered and specify which cores are idle and which are still processing. Blocking requests are answered as soon as all jobs have been finished.

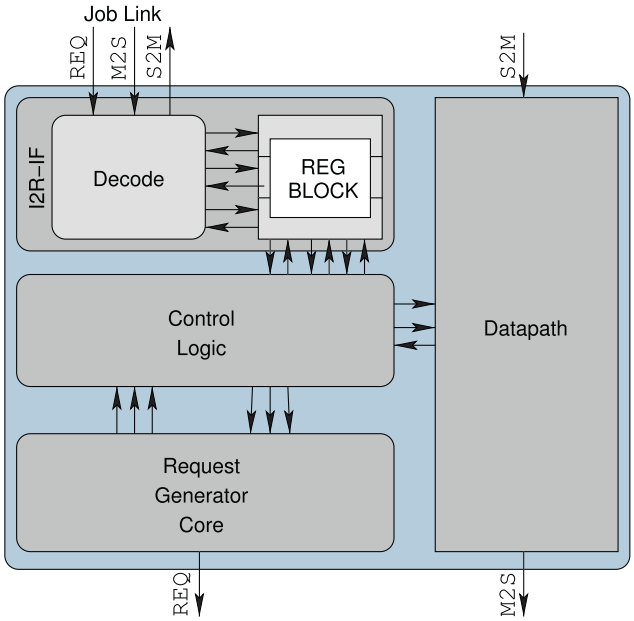

Fig. 10. Sample core using the IMORC-to-Register interface core.

4.5. Accelerator generation and portability

The IMORC architectural template is completely written in synthesizable VHDL. Specifically, the FIFOs are implemented based on the techniques described in [22] without using any architecture specific components, making IMORC a vendor-neutral tool. While currently we have only evaluated IMORC on the XD1000 platform (see Section 5), these properties make porting IMORC applications to new architectures rather simple. One has to implement a host interface core and appropriate memory interface cores for the concrete target architecture and then can remap the application to the new architecture.

Before synthesizing an application to an FPGA target the designer can tailor the IMORC architectural template to match the requirements of the application. Configurable parameters include for example the data widths of the channels for each master and slave port, FIFO depths, the number of masters for each arbiter, etc. To minimize resource overheads, the actually synthesized IMORC infrastructure is application-specific, i.e., only the required channels, arbiters and bitwidth conversion modules are inserted into the design. The decision which design units are required is taken based on the parameters specified by the designer. If, for example, the master's and slave's bitwidth are equal, the bitwidth conversion modules are not required and are automatically removed from the design. The generation of unnecessary channels can be either avoided by configuring the actually required number of ports using a VHDL generic parameter or the developer can leave unused ports open and rely on the synthesis tool to remove any unused logic.

5. XD1000 platform support

This section presents an overview of the XtremeData XD1000 reconfigurable workstation and the IMORC cores that have been developed as platform support. The XD1000 is a dual socket system where one of the sockets is equipped with a 2.2 GHz AMD Opteron processor, and the other one with a module featuring an Altera Stratix II EP2S180-3 FPGA. Physically, the FPGA is connected to the processor using a 16 bit HyperTransport link with 800 MT/s, which results in a rather low latency communication at a theoretical peak bandwidth of about 1.6 GB/s in each direction.

Furthermore, the XD1000 system comes with 4 GB of main memory (DDR SDRAM) for each the Opteron and the FPGA.

5.1. IMORC HyperTransport interface

For communication to the host CPU and memory, IMORC provides an interface to HyperTransport which is based on the HT cave described in [23]. The HT cave and our IMORC interface support communication initiated by the host as well as communication initiated by IMORC cores. HT packets consist of a header containing control information that is optionally followed by a certain amount of data. Fundamental elements of the header are the packet type (byte/dword read/write, broadcast, etc.), a mask/count field, a tag, an address and several other fields. HyperTransport uses virtual channels for communication, "posted" for requests not requiring a response, "non-posted" for requests expecting a response and "response" for responses. Each of these channels exists once per direction. Responses are related to the original non-posted request by copying the tag field.

The HT cave maps three distinct address regions (BARs) of the FPGA into the host processor's address space. HT packets sent by the host are decoded, packet headers are written to separate FIFO interfaces for posted and non-posted requests, data is also written to FIFO interfaces. Our IMORC interface decodes the packet headers. Packets targeting one of the first two address regions are converted into IMORC requests and forwarded to two distinct IMORC links. Data corresponding to the HT packets is also forwarded to the M2S channel of the appropriate link. For reads, the tag and count field and the ID of the IMORC link used is forwarded additionally to a response datapath using a FIFO. As soon as data becomes available on the appropriate IMORC link's S2M channel, an HT header containing the original tag and count field is sent to the HT cave's response FIFO and the data is forwarded to the data FIFO.

The third address region directly accesses an embedded block of memory, used as page mapping table which maps host memory into the address space of a third IMORC link. For this purpose, the page mapping table needs to be filled with physical page addresses of the target host memory by the user application. When cores send requests over this IMORC link, the upper bits of the IMORC request form an index into the page mapping table, the lower bits are the offset into the page. Using this address mapping, the IMORC packet is converted into an equivalent HyperTransport packet and posted to the appropriate virtual channel's FIFO interface of the HyperTransport cave. For read requests, the tag field is incremented with each request posted. Since HyperTransport supports out of order responses, the response datapath possibly has to reorder these responses. For this purpose, it contains a block of embedded memory for each tag, which can store one full-sized HT data packet. Responses arriving on the HT are first written into the appropriate memory block, and the block is marked as valid. The TAG is marked as free, enabling the request block to reuse it for another HT read request. A separate logic waits until the data block for the first tag is valid. Then, data in this block is forwarded to the S2M channel of the IMORC link, the data block is marked as invalid and the logic waits for the next tag's data block to become valid. Additionally, the third IMORC link can be used for sending interrupt messages to the host CPU. Writes to a configurable address on this IMORC link are converted into an HyperTransport interrupt message that is sent to the host.

Table 2 lists the resource requirements of the HT interface core. The number of used block memory resources (M512, M4k and M-Ram) mainly depends on the amount of host memory that is mapped into the FPGA's address space, e.g., the size of the embedded page mapping table. The core's user interface may be operated at up to 200 MHz, enabling accelerators to utilize the maximum bandwidth delivered by the HT interface.

Table 2
Resource requirements for the HyperTransport interface core and the DDR SDRAM controller core on an Altera Stratix II EP2S180 FPGA.

	ALUTs	REGs	M512	M4k	M-Ram
Available	143,520	143,520	930	768	9
HT	8791 (6.13%)	5209 (3.63%)	1 (0.1%)	57 (7.42%)	1 (0.11%)
DDR	1992 (1.39%)	2159 (1.5%)	–	8 (1.04%)	–

5.2. Interface to the DDR SDRAM

For off-chip memory access, IMORC wraps the Altera DDR SDRAM controller core which can access memory in blocks of configurable burst sizes. DDR SDRAM writes always cover a complete burst, i.e., 2, 4 or 8 clock cycles, depending on the controller configuration. The XD1000 system provides a 128-bit-wide memory. Hence, depending on the burst size, 256 bit, 512 bit or 1024 bit are written to memory during a transfer. Since the XD1000 does not provide data mask pins for the memory to mask out bytes not to be written during a burst, IMORC implements a read-modify-write cycle for writes that do not match one of the burst sizes. The read-modify-write cycle incurs overhead but at the same time increases flexibility and potential for core re-use, as the core designer is not necessarily bound to pre-determined burst sizes and link widths. Table 2 lists the resource requirements of the DDR SDRAM interface core that includes the Altera DDR SDRAM controller core and the interface to IMORC. The IMORC interface of the DDR SDRAM interface core is operated at the same clock frequency of 166 MHz as the DDR memories, so the full bandwidth of the memories can be utilized.

6. Case study

We demonstrate the IMORC workflow and benefits on the detailed example of developing a hardware accelerator for the *k*th nearest neighbor thinning problem on the XtremeData XD1000 system [1]. In this section, we present the *k*th nearest neighbor thinning problem and the application-specific IMORC cores that are derived from it.

6.1. kth nearest neighbor thinning

*k*th nearest neighbor (KNN) methods are omnipresent in many areas of science and engineering. In statistics and data analysis, for example, KNN techniques play an important role for the non-parametric estimation of density functions from data samples [24]. Given a set of n data samples, where each sample i is a d-dimensional vector, an Euclidean distance metric σ_i is computed for any pair of samples. For each data sample, the resulting n distance values are sorted in ascending order, i.e., $\sigma_i^1 \leqslant \sigma_i^2, \ldots, \leqslant \sigma_i^n$. A KNN density estimate $\hat{f}(i)$ can then be formulated by setting: $\hat{f}(i) \propto 1/\sigma_i^k$.

The parameter k is typically chosen as $k \approx \sqrt{n}$ [25]. Thus, the local density around each data sample i is estimated by the reciprocal of the distance to the kth nearest neighbor. In other words, a low density means that a d-dimensional sphere with data sample i at its origin has to be rather large in order to contain k data samples.

The KNN approach is also widely applied for solving classifications problems, such as in machine learning, data mining and stochastic optimization [26]. There, a KNN classifier requires a labeled training data set consisting of d-dimensional feature vectors and their class labels. In order to classify a new feature vector, the k nearest training vectors are determined according to some

distance metric. Often, a reduction of the size of data samples is desired to reduce both the classification time and the memory required to store the data set. Many reduction techniques fall into the category of condensing or thinning approaches, [27], that aim at properly selecting a subset of training vectors from the original data set.

Recently, some methods related to KNN-based thinning have been successfully accelerated with FPGAs. For example, Yeh et al. present a KNN classifier [28] that operates in the wavelet domain and uses partial distance search to accelerate the classification process. The resulting architecture is integrated as a core with the Altera NIOS CPU softcore. In [29] Chikhi et al. present an FPGA accelerated KNN classifier for content-based image retrieval that achieves a speedup of 45× over a software implementation. Also the related *k*-means clustering method has been successfully accelerated in reconfigurable hardware. For example, Saegusa and Maruyama have presented an architecture [30] that can perform *k*-means clustering on video data in realtime.

The KNN-based thinning method we are targeting in this work takes a set of input vectors and searches for the subset of vectors that represents the distribution of the original set the most representative. The KNN thinning procedure, shown in Algorithm 1, is called with a set $\mathcal{P}$ of different d-dimensional vectors $p_i = (p_{i1}, p_{i2}, \ldots, p_{id})$ and N, the targeted cardinality of $\mathcal{P}$, and successively eliminates vectors with the shortest Euclidean distance to their neighbors until $\mathcal{P}$ has been reduced to N elements. In each iteration, the algorithm first constructs a distance matrix $\underline{\sigma} = (\sigma_{il})$ from the pair-wise Euclidean distances σ_{il} between all vectors. Sorting $\underline{\sigma}$ row-wise in ascending order defines sorted distance vectors $\sigma'_{\mathbf{i}}$ of length m. While initially equal to $|\mathcal{P}|$, the number of vectors m is reduced by one in each iteration. Now, the algorithm iterates over all columns of $\underline{\sigma}'$. Starting with column $l = 3$, the rows σ'_i with minimum distance values σ'_{il} among all distances in column l are selected and assigned to set $\mathcal{M}$. If the minimum is unique, the respective row $\sigma'_i \in \underline{\sigma}'$ as well as the corresponding vector p_i are deleted which reduces the set of vectors $\mathcal{P}$. If the minimum is not unique, the next column of $\underline{\sigma}'$ is considered which corresponds to checking the distances to the next closest neighbors. If no unique minimal distance is found for all columns of $\underline{\sigma}'$, an arbitrary row having a minimal distance value in the last column is deleted. Obviously, the first two columns never need to be considered since each vector has a distance of 0 to itself (first column) and the distances between pairs of vectors are symmetric (second column).

Algorithm 1. KNN thinning algorithm

```
1: procedure KNN_thinning P, N
2:   while |P| > N do
3:     compute/update σ_il ← sqrt(Σ_{j=1}^{d} (p_ij − p_lj)^2)
4:     ∀ rows of σ : σ'_i ← sort(σ_i)
5:     for l ← 3,...,m do
6:       M ← {σ'_i | ∀σ'_jl : σ'_il ⩽ σ'_jl}
7:       if |M| == 1 then
8:         break
9:       end if
10:    end for
11:    delete arbitrary row σ_i ∈ M from σ'
12:    P ← P \ {p_i}
13:  end while
14: end procedure
```

The operation of the KNN-based thinning algorithm is illustrated in the example shown in Fig. 11. The initial population of six 2-dimensional vectors as well as the three vectors that are discarded by three iterations of the thinning algorithm are shown in Fig. 11a. The distance matrix $\underline{\sigma}$ for the first iteration is presented in Fig. 11b, and the row-wise sorted distance matrix $\underline{\sigma}'$ in Fig. 11c. Note that the matrices actually show the square distances which is sufficient for this algorithm. A unique minimum is found in the third column which leads to the deletion of row b from the matrix and vector b from the population, respectively. In the second iteration, the distance matrix is updated and re-sorted which results in the matrix of Fig. 11d. Again, a unique minimum is identified in the third column and, consequently, row e and vector e are deleted. Finally, the third iteration leads to the deletion of vector c.

6.2. Mapping of KNN to IMORC and refinements

In this section we discuss the mapping of the KNN thinning application to the IMORC architecture and present the refinements we make during this process. As visible in Fig. 5, the design flow usually starts with partitioning and initial mapping of the application. The partitioning phase hereby consists of benchmarking and profiling the application for extracting the most compute intense parts of the application, which are then initially mapped to the FPGA. Different tools can be used for the profiling procedure, such as, GProf [31], Valgrind [32], or OProfile [33]. However, in this case study we do not concentrate on a large application from which only parts are to be accelerated, but the focus lies on the KNN thinning procedure that is to be completely implemented on the FPGA. The KNN procedure will typically not be executed as an isolated application, but as a kernel in a larger application. Since it is already clear that the complete KNN kernel is to be implemented on the FPGA, the partitioning phase is omitted in this case study and the design process starts with the initial mapping phase.

6.2.1. Initial mapping

For mapping the KNN thinning application to the IMORC architecture template, we need to decompose the application into a set of communicating tasks which will be mapped to IMORC cores later on. In addition to the decomposition of the functionality into tasks, we can already determine which tasks could be potentially executed in parallel or in a pipelined fashion at this early stage of the design process. As the algorithmic core of our application is well described in a formal way we can derive much of this information from analyzing Algorithm 1.

We can see, that the algorithm can be broken up into five major tasks:

- *distance calculation:* computing the distance values (line 3 of Algorithm 1),
- *sort:* sorting the distance values (line 4),
- *search:* searching for the vector with minimum distance (lines 5–10),
- *discard:* discarding this vector from the distance matrix (line 12), and
- *ctrl:* controlling the iteration of this process until the number of vectors has been reduced to a target number (line 2).

Further, we can identify two main data structures that are needed by the application, which will also be modeled as IMORC tasks:

- *VM:* a memory for storing all vectors, and
- *DM:* a memory for storing the results of the distance computation.

The interaction of the cores in this initial application mapping is shown as a task graph in Fig. 12a. This figure also shows a data

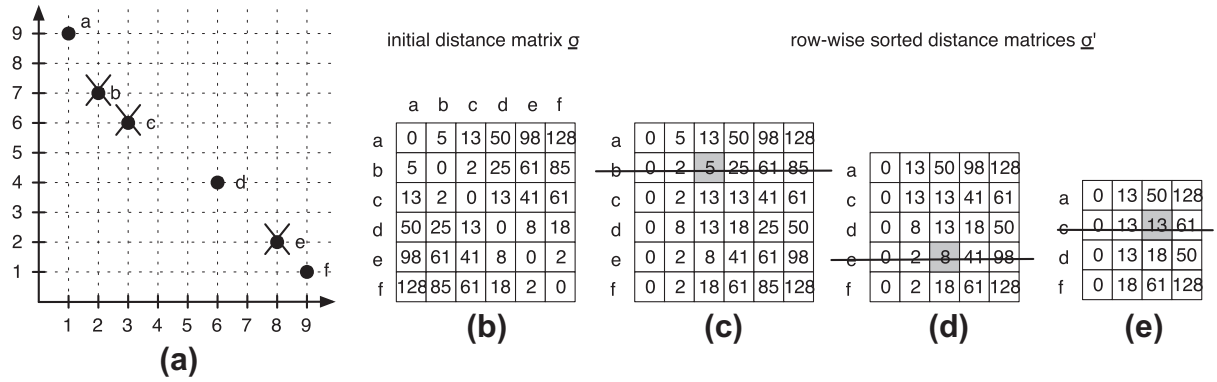

Fig. 11. Example for the KNN-based thinning algorithm in two dimensions: thinning 6 vectors to 3 vectors.

generation task that is executed on the host CPU and is triggering the KNN controller task.

6.2.2. Refinement #1: Compute distances just once

At this stage in the design process, we take two decisions affecting the accelerator architecture. First, to reduce the amount of computations we decide to move the computation and sorting of distance values out of the loop in Algorithm 1 and compute the sorted matrix $\underline{\sigma}$ only once. This is made possible by storing with each distance value σ_{il} the indices of the two vectors for which the distance has been computed. The discard task now not only operates on the original vector table but also on the distance table for removing all references to the vector to be discarded.

This modification increases the size of the distance table and therefore also increases the runtime for the distance calculation, the sort task, and the discard task, which now has to process the complete distance table. However, the distance calculation and the sort tasks are now only executed once, which reduces the overall runtime. In essence, we trade an increased memory requirement for improved performance with this refinement.

The interaction of the cores in the refined mapping is shown as a task graph in Fig. 12b.

6.2.3. Refinement #2: Exploit parallelism with farming cores

The next refinement step is guided by the observation that the distance computation between two vectors and sorting the rows of the distance matrix (lines 3 and 4 in Algorithm 1 are well amenable for parallelization. To exploit the available parallelism we can use IMORC's farming capabilities to distribute the tasks of distance calculation and sorting to several identical subtasks that perform the actual operation.

In this refinement, each distance calculation task is responsible for calculating the distances of exactly one vector to each other, hence calculating one row of the distance matrix $\underline{\sigma}$. The same refinement is done for the sorter task, each one sorts one row of $\underline{\sigma}$. Each distance calculation task starts the corresponding sorter task upon finishing. However, since the search and discard tasks may not be started before all distance calculation and sorter tasks are finished, the sorter tasks are not allowed to directly start the search and discard task. Instead, an additional control task is inserted to synchronize the finishing of the sorter tasks. The interaction of the cores in the refined mapping is shown as a task graph in Fig. 12c.

While it is clear that this refinement may improve the overall performance the impact of increasing the number of distance calculators and sorters used by the farming cores is not evident. Even for a comparatively simple application like KNN thinning, the interaction between cores, the delays introduced by competing for shared resources or the input-specific processing times (the minimum computation in this case), make an analytical performance modeling difficult. Hence, to find a suitable number of distance calculation and sort tasks we evaluate the implementation with test sets whose runtime behavior is analyzed with IMORC's performance counters.

We use the performance counters for measuring the number of full and empty events that happen on the corresponding FIFOs in the IMORC links during each phase of the KNN algorithm. For example, if a FIFO in the S2M channel frequently runs full and the corresponding M2S FIFO runs empty, the respective core is known to be compute-bound. In that case we apply farming and add more compute cores for the corresponding task for maximizing throughput. Conversely, if the S2M FIFO frequently runs empty and the M2S FIFO often runs full, the task is certainly memory-bound. Hence, the number of processing cores used in farming may be reduced without an impact on the overall performance.

The availability of performance counters allows us to measure such information for a large set of realistic test sets during execution. Much like established profiling tools for software, IMORC provides runtime statistics and allows for identifying bottlenecks in more complex accelerator architectures with data-dependent behavior. It must be emphasized that IMORC monitors the running accelerator in real-time instead of a simulation model. For example, our experiments with VHDL simulations of the KNN accelerator for $|\mathcal{P}| > 128$ and $d = 32$ have shown that we need 8.6×10^6 clock cycles on the simulating machine for one cycle of the simulated accelerator. Simulating VHDL models of the KNN accelerator with realistic workloads thus takes days.

6.3. Implementation of KNN cores

After the application has been decomposed into a number of communicating tasks in the application mapping phase that has been discussed in the previous section, each of the tasks needs to be implemented as IMORC core. In the following we discuss the architecture and implementation of these cores.

6.3.1. Distance calculator core

The computation of the Euclidian distances, line 3 in Algorithm 1, is wrapped into a distance calculator core (cmp. Fig. 13). We omit the square root function as squared values are sufficient for comparing distances. The distance calculator core computes one row of the distance matrix σ_i on each invocation, which allows us to exploit parallelism by instantiating several such cores.

To receive job messages, the distance calculator core contains a slave port, connected to an I2R interface core. A job message comprises the base address of the vectors in vector memory, the

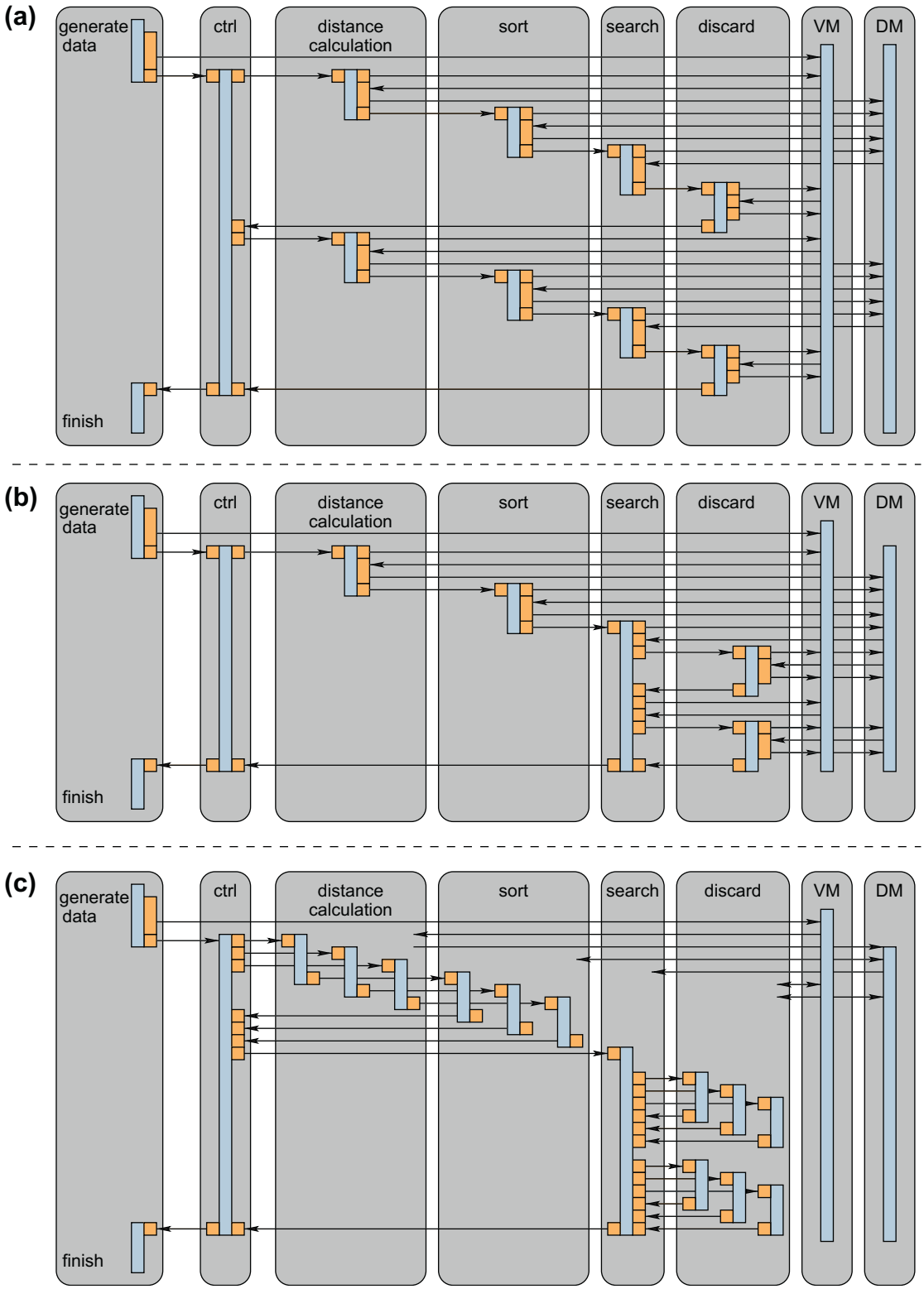

Fig. 12. Development of the KNN execution model.

number of vectors and dimensions, m and d, the index i of the vector for which the distances to all vectors have to be determined, and the address in distance memory where σ_i is to be stored.

Two request generator cores are connected to the I2R interface core, the first one sending read requests to the vector memory, the second one for sending writes to the distance memory. The read request generator is first configured for retrieving the vector i from vector memory. When finished, it is reconfigured for issuing read requests to fetch all vectors. At the same time, the second request generator is configured for sending write requests to the distance memory for storing the distances.

The datapath reads the first i vector's coordinates from the vector memory's S2M-channel and stores them into an internal buffer. Further coordinates are subtracted from the appropriate

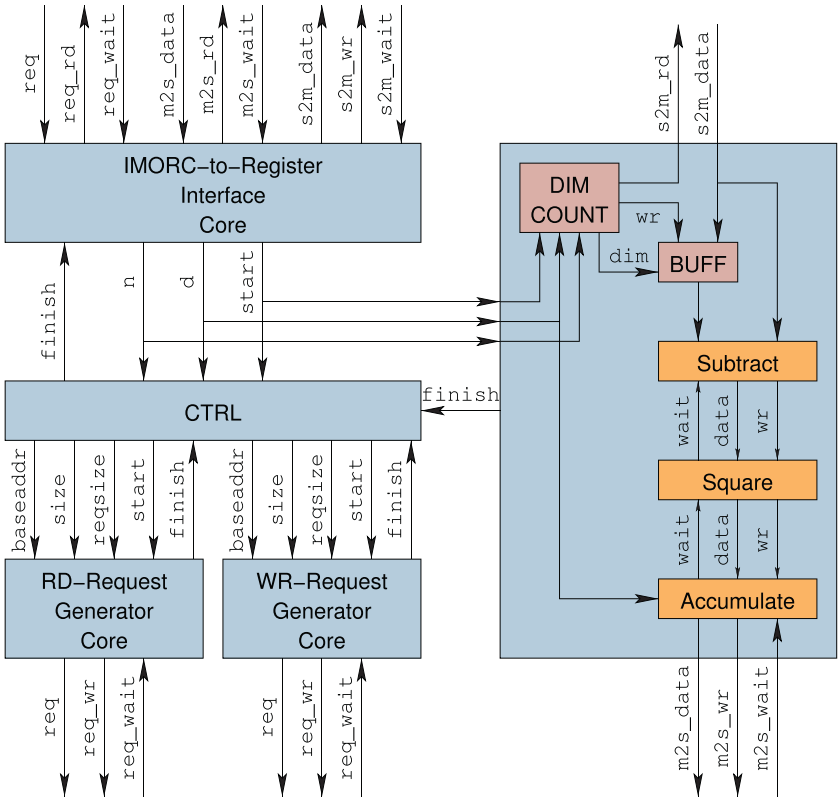

Fig. 13. Block diagram of the distance calculator.

coordinates in the buffer, the coordinates distances are squared and added up. The resulting distances are sent to the distance memory's M2S-channel.

When all distance values are written back, the sorter job generator is started, generating a sort job for the computed distances. The I2R interface core is then set IDLE for accepting further distance calculation jobs or state request messages.

6.3.2. Sorter core

For the sorting step, line 4 in Algorithm 1, we use a variant of bubble sort. On each invocation, the sorter core sorts one row σ_i of the distance matrix, again allowing to exploit parallelism by instantiating several sorter cores. The job messages are received via a slave port and comprise the number of vectors m and the base address of σ_i in distance memory. The job message is decoded in an I2R interface core.

Then, the core proceeds as follows: In the first iteration, a request generator is configured for reading m elements of σ_i, i.e., the complete row from distance memory and for storing the same amount of data back to the same location. The datapath stores the first element received on the S2M channel as the current maximum. If the next element is smaller than or equal to the current maximum, it is directly forwarded to the M2S channel. Otherwise, it becomes the new current maximum and the previous maximum is sent to the M2S channel. At the end of the iteration, the current maximum is written to the M2S channel. Additionally, we remember the position l of the last element in σ_i that has been moved left. Thus, in the next iteration only l elements of σ_i are to be streamed through the sorter core. So, the request generator core is reconfigured for reading and writing l elements. This procedure is repeated until the complete row is sorted.

At this point, the state register of the I2R interface core is reset, so that further sort jobs can be processed or occurring state request messages can be responded.

6.3.3. Search core

The search core implements lines 5–10 of Algorithm 1 and is invoked with a job message specifying the total number of vectors m, the dimension of the vectors and the goal to achieve (e.g. the number of vectors not to discard). The job is decoded in an I2R interface core. A controller transforms these values into input data for an IMORC request generator, which generates read requests to the elements in the third column of the sorted distance matrix. Since data is stored row-wise, m request with a size of 64 bit each have to be generated, with an offset of $m \times 8$ byte.

The datapath receives the values from the distance memory, finds the minimum and the information if this minimum was seen more than once. If no unique minimum was found, the controller instructs the request generator to generate requests for the next column. This procedure repeats, until a unique minimum is found or until all columns were searched.

When finished, discard jobs are generated, one for each row of the distance table. The controller waits until all discard jobs are finished and, if the configured goal is not achieved yet, starts the next search iteration.

6.3.4. Discarder core

The discarder core implements lines 11–12 of Algorithm 1 and is invoked with a job message specifying the total original and current number of vectors m_{orig} and m_{cur}, their dimension d, the

Table 3
Parameters of the discarder's request generator.

	$id_c \neq m_{cur} \wedge id_c \neq id_{disc}$	$id_c = m_{cur} - 1$	$id_c = id_{disc}$
rd_base:	$id_c \times m_{orig}$	$id_c \times m_{orig}$	$(m_{cur} - 1) \times d$
rd_size:	m_{cur}	m_{cur}	d
wr_base:	$id_c \times m_{orig}$	$id_{discard} \times m_{orig}$	$id_{disc} \times d$
wr_size:	$m_{cur} - 1$	$m_{cur} - 1$	d

id id_{disc} of the vector to discard and the id id_c of the current vector (or row in the distance matrix) to process.

A read-write request generator is configured with the main parameters as defined in Table 3. Three different cases exist:

$id_c \neq m_{cur} \wedge id_c \neq id_{disc}$ *the requests target the distance memory, reading* m_{cur} *distances and writing* $m_{cur} - 1$ *distances back into the same row.*

$id_c = m_{cur} - 1$ *the requests also target the distance memory, reading and writing the same amount of data as in the previous case. This time, the writes do not target the same row as the writes, but the row where the distances origin at vector* id_{disc} *reside.*

$id_c = id_{disc}$ *in this case, the requests target the vector memory, reading all coordinates of the last vector* $(m_{cur} - 1)$ *and storing them to the location where the coordinates of vector* id_{disc} *reside.*

With these parameters, the vector with ID id_{disc} is replaced with the last vector in the vector table as well as in the distance table. Since the id of the last vector is now changed, the datapath has to perform this change in the distance table's entries, too. The datapath connected to the distance memory's link reads data from the S2M channel and compares the IDs prepended to the distance value to id_{disc} and to $m_{cur} - 1$. Every entry with an ID equal to id_{disc} is discarded, every ID equal to $m_{cur} - 1$ is replaced by id_{disc}. Entries not discarded are then written to the M2S channel. The datapath connected to the vector memory's link only forwards all values from the S2M channel to the corresponding M2S channel.

6.3.5. Controller core

The controller core is responsible for sending job requests to the appropriate cores. A IMORC2Register interface core is used for decoding job messages received from the host. When such a message arrives, the distcalc job generator calculates parameters for n distance calculation jobs and sends them to the distance calculators using a REG2IMORC interface core. When all jobs are sent, a read request is sent to the distance calculators for being informed when all jobs are finished. Then, the sort waiter is started. Since sort jobs are directly generated and sent to the sorters in the distance calculators, this core only sends a read request to the sorter cores for getting their finishing time. Then, the search job generator generates parameters for the search module, sends them to the search core using the REG2IMORC interface core and again waits for completion. Last, the interrupt generator core is started, sending an interrupt message to the host link, informing the host that the current job is finished.

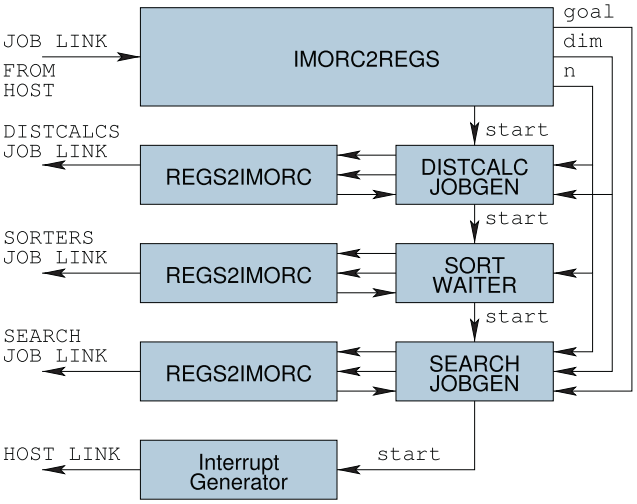

Fig. 14. Block diagram of the controller core.

Fig. 14 shows a block diagram of the controller core. Most of the controller core could be implemented using the existing IMORC supporting cores, only a few lines of additional HDL code had to be written for calculating the correct job parameters.

6.4. Numeric evaluation

Using the cores presented in the previous section, we generate a configurable accelerator for the XtremeData XD1000 system. The number of distance calculators, sorters and discarder cores in the accelerator is configurable, job distribution is performed using the IMORC farming cores. All slave arbiters except that for the sorter job messages use the default round robin port scheduler. The sort job slave arbiter uses a modified port scheduler, which tries to select one of the links from the discarder cores in round robin manner first, and only selects the link from the controller core if all other links are empty. This way, the final status request from the controller is ensured to be posted to the farming core last, when all job messages are already forwarded to the sort compute cores. Fig. 15 shows the IMORC diagram of the complete accelerator with each two distance calculators, sorters and discarder modules.

Using this mapping, we generate accelerators with different configurations (indicated as $(n_{distcalc} \times n_{sort} \times n_{discard})$. Table 4 shows the resource usage of some basic elements of the accelerators. The last two rows represent the resources used by the complete accelerator and the resources used for the IMORC communication infrastructure of the complete accelerator in the $(6 \times 6 \times 6)$ configuration. The communication infrastructure hereby takes about 14% of the overall logic resources.

Fig. 16 and Table 5 show the speedups these accelerators generate over an optimized software solution that utilize the host CPU of the XD1000 workstation only. The reported speedups are application-level speedups, that is, all overheads for data transfers are included. The test sets were executed with test vectors in 32 and 128 dimensions. The initial data set consisted of 32–1024 vectors that were thinned to 1/4 of the original set of vectors (i.e., 3/4 of the original data set was discarded). Vector coordinates and distance values were stored in 32 bit fixed point representation, the vector ids prepended to the distance values were stored in 16 bit unsigned integer. Hence, each element in the distance matrix was 64 bit wide (32 bit distance +2 × 16 bit vector ids). The HyperTransport interface core was running at 200 MHz, the DDR SDRAM at 166 MHz. The accelerator clocks were configured to 200 MHz for the CTRL core, 100 MHz for the distance calculator core, 120 MHz for the sorter core and 150 MHz for the search and the discard core.

We can see, that increasing the number of compute cores increases the speedups in all cases. The best configuration presented in the figure is the $(6 \times 6 \times 6)$-configuration, which provides speedups of up to 44 × over the optimized software solution. From the $(6 \times 1 \times 1)$, $(1 \times 6 \times 1)$ and $(1 \times 1 \times 6)$ configuration we can also see that the number of distance calculators used only has only a minor impact on the actual computation time. The impact of the number of sorters and discarder modules is much larger.

Table 6 exemplary shows the execution times of the software implementation and the different accelerator configurations with $m = 1024$ initial vectors. While the software implementation needs about 33,466 ms for $d = 32$ and about 47,957 ms for $d = 128$, the $(6 \times 6 \times 6)$ configuration of the accelerator only takes about 771 ms and 1080 ms for the same problems, respectively. These runtimes include the thinning procedure and, since the accelerator directly accesses the vector table in host memory, the time needed for data transfer.

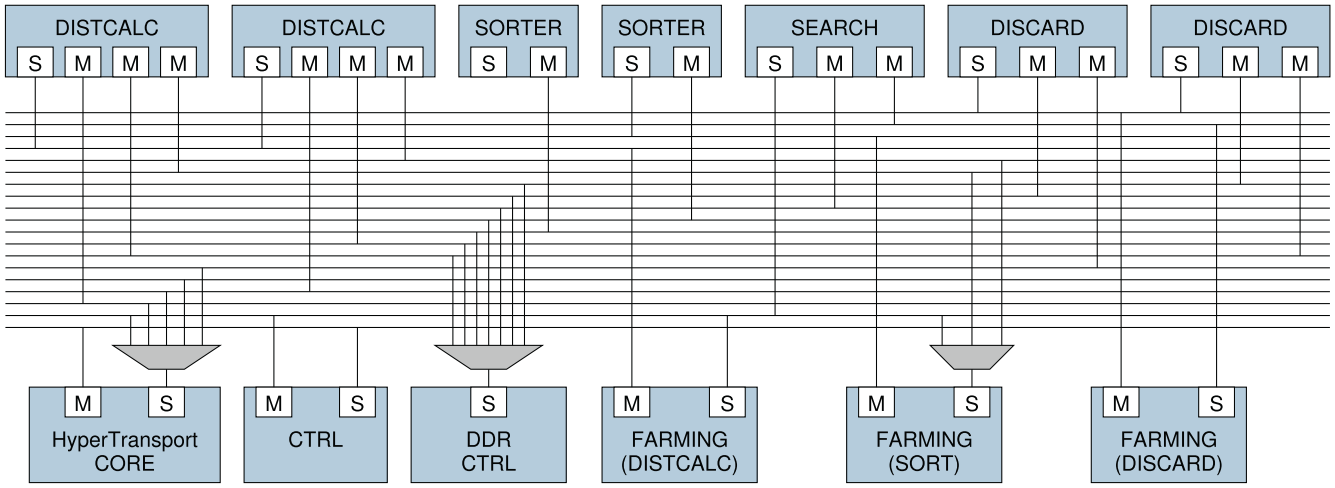

Fig. 15. Architecture diagram of the accelerator with two distance calculators/sorters/discarders mapped to the XD1000.

Table 4
Resource requirements for the accelerator.

	ALUTs	REGs	DSP	M512	M4k	M-Ram
EP2S180-3	143520	143520	768	930	768	9
Distcalc core	1462	1192	6	–	–	1
Sorter core	967	1176	–	–	–	–
Search core	1758	1035	4	–	–	–
Discarder core	1507	1504	–	–	–	–
CTRL core	782	325	–	–	–	–
IMORC link	99	226	–	–	4	–
Bitwidth conv.	68	126	–	2	–	–
Farming core	76	29	–	–	–	–
DDR CTRL	1992	2159	–	–	8	–
HT core	8791	5209	–	1	57	1
Complete acc.	42,084	35,492	40	7	383	8
IMORC	5896	4766	–	28	344	–

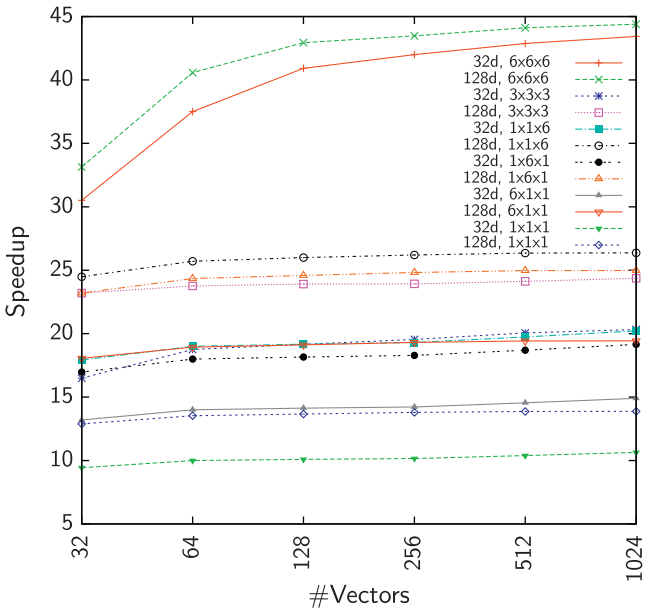

Fig. 16. Resulting speedups for different accelerator configurations.

For a systematic optimization of the architecture we have used the data collected by the performance counters. While KNN thinning is deterministic in the sense that no randomness is used in any part of the algorithm, the application exhibits data-dependent processing times. For example, the time for sorting the distance vectors depends on the contents of the vectors and the subsequent search for a unique minimum distance is also data-dependent. Hence, the number of sorter cores is a design optimization parameter that can be adjusted to obtain a desirable utilization vs. performance ratio. Monitoring the utilization of sorting cores when executing representative test sets helped us determining an optimal number of sorter cores to be included in the design and to chose optimal sizes for the FIFOs.

7. Further applications of IMORC

While the case study presented in this paper discusses the IMORC development flow for an accelerator for the kth nearest neighbor thinning problem, we have also evaluated IMORC in further case studies presented in different publications. One of these case studies focuses on the cube-cut problem, which we introduced in [34]. The cube-cut problem deals with the question, how many hyperplanes are necessary for cutting all edges of the d-dimensional hypercube. While for $d \leqslant 5$ it is known that at minimum d hyperplanes are required, it was shown that for $d = 6$ actually only five hyperplanes are required. A formal proof for arbitrary numbers for d seems to be hard to obtain, which led to the development of a computational evaluation of the problem. The underlying algorithm first calculates all possible cuts that are possible with one hyperplane, the hypercube edges sliced by the cuts are stored as bitstrings with one bit for each edge. A value of '0' means, that the corresponding edge was not sliced, while a value of '1' means that the edge was sliced. Given a list of bitstrings representing all possible cuts, the goal now is to find the minimum number of cuts that slice all edges of the hypercube. In other words, we have to find a minimum set of bitstrings where each bit has a value of '1' in at least one of the bitstrings. To reduce the number of bitstrings to be searched for such a minimum set, in an intermediate step the bitstrings are searched for domination. A bitstring b is said to dominate another bitstring a when all bits of a with a value of '1' also have a value of '1' in b. In [34] we presented a detailed introduction into the cube-cut problem and implemented an accelerator for the dominance check phase on a cluster of four compute nodes equipped with an AlphaData ADM-XP FPGA board featuring a Xilinx Virtex II Pro 2VP70 FPGA, each. Using the IMORC development flow and architecture template, we ported this accelerator to the XtremeData XD1000 [35]. The IMORC development flow hereby helped in analyzing the application, generating the accelerator

Table 5
Speedups for the different accelerator configurations.

m	d	$1 \times 1 \times 1$	$6 \times 1 \times 1$	$1 \times 6 \times 1$	$1 \times 1 \times 6$	$3 \times 3 \times 3$	$6 \times 6 \times 6$
32	32	9.43	13.20	16.97	17.92	16.50	30.51
32	128	12.89	18.05	23.20	24.49	23.20	33.14
64	32	10.02	13.97	18.06	18.93	18.74	37.50
64	128	13.53	18.94	24.35	25.71	23.76	40.58
128	32	10.09	14.13	18.16	19.17	19.16	40.92
128	128	13.66	19.12	24.59	26.00	23.91	42.94
256	32	10.16	14.22	18.29	19.30	19.55	42.00
256	128	13.79	19.32	24.82	26.21	23.93	43.48
512	32	10.39	14.55	18.70	19.74	20.06	42.87
512	128	13.87	19.42	24.97	26.35	24.12	44.11
1024	32	10.64	14.90	19.25	20.22	20.33	43.43
1024	128	13.88	19.43	24.98	26.37	24.37	44.39

Table 6
Runtime comparison of the software implementation and the different accelerator configurations ($m = 1024$, runtimes in ms).

d	SW	$1 \times 1 \times 1$	$6 \times 1 \times 1$	$1 \times 6 \times 1$	$1 \times 1 \times 6$	$3 \times 3 \times 3$	$6 \times 6 \times 6$
32	33,466	3145	2246	1738	1655	1646	771
128	47,957	3455	2468	1919	1819	1968	1080

design and specifying parameters for this design. IMORC request generator cores were used for reading and writing the data. Since the amount of data to be written back is not deterministic but depends on the number of elements discarded, we used the request generator variant that injects requests after a certain amount of elements was written to the M2S channel. The resulting accelerator provided a speedup of 365× over a CPU implementation executed on the Opteron processor of the XD1000.

Another case-study we have presented is an accelerator for a parallel rendering and 3D compositing application [5]. The application divides the scene to be rendered across the compute nodes of the cluster. A master node collects the rendered framebuffers and the corresponding z-buffers. Then, it composes the framebuffers pixelwise based on the depth information of the z-buffers. The partitioning phase of the development process showed that the compositing kernel is the most time critical component of the complete application, since the master node has to gather the data from all compute node. Hence, our accelerator focused on the compositing kernel. Compositing is a very simple operation, the performance achievable greatly depends on the time needed for data accesses. The IMORC development flow along with the architecture characterization greatly helped in optimizing the throughput of the accelerator. Data accesses could be efficiently implemented using the IMORC request generator cores, only for the compositing kernel's datapath a small amount of custom VHDL code had to be implemented. In the end, we achieved an increase of the frame rate of 1.25× to 2.1× which resulted in a much smoother visualization than the original CPU implementation.

8. Conclusions

In this paper we have made the case for raising the level of abstraction in the design process for FPGA accelerators for high-performance computing. To this end, we propose the use of the IMORC infrastructure and architecture template that have been presented in this paper. IMORC provides a model for designing FPGA accelerators as an ensemble of communicating tasks. The interaction between tasks is not restricted to a specific formal model, but supports many programming techniques that are commonly used in high-performance parallel computing, such as, parallel execution of independent tasks, pipelining, and farming. Further the IMORC communication model allows concurrent communication over a multi-bus structure and overlapping communication and computation, which are key techniques for creating highly effective FPGA accelerators. Thus, IMORC provides an important step in generating useful and familiar abstractions for designers from the high-performance computing domain.

The IMORC architecture template allows for mapping the task model to an FPGA accelerator by providing a customizable communication infrastructure for connecting application-specific cores which implement the tasks. Further, IMORC provides a number of infrastructure cores that implement common functions, that can be reused in many accelerators, such as memory, request generator, register interface, and farming cores. The creation of FPGA implementations of the application-specific compute tasks is out of the scope of IMORC. However, IMORC provides an easy to use FIFO interface, which significantly simplifies the design of a custom core. The FIFO interface allows the designer to focus on the actual data processing task without considering the synchronization and communication with other cores. The well defined FIFO interface also allows for using high-level C-to-gates synthesis tools that are suited for streaming applications, such as, Impulse C, Catapult C or similar tools. Finally, IMORC supports the process of optimizing an FPGA accelerator by providing built-in performance counters that allow for monitoring the execution in real-time. This allows the designer to use real workloads for analyzing bottlenecks in the architecture and for guiding the design optimization process.

By means of an in-depth case study, we have shown how the IMORC design method is applied to a KNN thinning algorithm. Starting from the algorithm, we have decomposed the application into a set of communicating tasks. We have shown how this task model has been refined several times for optimizing the performance while remaining at the high level of abstraction of a task model. The final refinement has been implemented as an FPGA accelerator by leveraging the IMORC architecture template for communication and the infrastructure cores and by translating the computing cores to custom IMORC cores. This process has resulted in a high-performance FPGA accelerator for KNN thinning implemented on an XD1000 reconfigurable workstation. Depending on the problem size, this accelerator is able to outperform the host CPU by 10× to 40×.

In future we plan to extend our work in two main directions. First, we plan to work towards an integrated performance analysis and optimization approach that uses data gathered by the performance counters to calibrate analytical or statistical performance models. Second, we will look into automatically mapping application models to IMORC architectures. While in our case study we

have manually analyzed the KNN thinning algorithm and decided which cores to implement and how to schedule them, more formal application models, e.g., process networks, might allow us to derive core topologies and schedules automatically.

Acknowledgement

We kindly acknowledge the support of this work through the Altera-AMD-Sun-XtremeData university program.

References

[1] XtremeData, Inc., Schaumburg, IL, USA, XD1000 Development System, 2008. <www.xtremedatainc.com>.
[2] A. Cantle, R. Bruce, An Introduction to the Nallatech Slipstream FSB-FPGA Accelerator Module for Intel Platforms and the Nallatech High-Level Toolset (NT 405-0342), Nallatech Ltd., Glasgow, UK, September 2007.
[3] T. Schumacher, C. Plessl, M. Platzner, IMORC: application mapping, monitoring and optimization for high-performance reconfigurable computing, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE Computer Society, 2009, pp. 275–278, doi:10.1109/FCCM.2009.25.
[4] T. Schumacher, C. Plessl, M. Platzner, An accelerator for *k*-th nearest neighbor thinning based on the IMORC infrastructure, in: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2009, pp. 338–344.
[5] T. Schumacher, T. Süß, C. Plessl, M. Platzner, Communication performance characterization for reconfigurable accelerator design on the XD1000, in: Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig), IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 119–124. doi:10.1109/ReConFig.2009.32.
[6] OpenCores.org, WHISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision: B.3, September 2002.
[7] Advanced Risc Machines (ARM), AMBA specification.
[8] IBM, The CoreConnect Bus Architecture White Paper, September 1999.
[9] Altera, Avalon Memory-Mapped Interface Specification, 2007.
[10] L. Shannon, P. Chow, Simplifying the integration of processing elements in computing systems using a programmable controller, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2005, pp. 63–72.
[11] M. Mitic, M. Stojcev, An overview of on-chip buses, Facta Universitatis, Series Electronics and Energetics 19 (3) (2006) 405–428.
[12] C. Steffen, Parametrization of algorithms and FPGA accelerators to predict performance, in: Proceedings of the Reconfigurable System Summer Institute (RSSI), 2007, pp. 17–20.
[13] M. Smith, G. Peterson, Analytical Modeling for High Performance Reconfigurable Computers, in: Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS).
[14] M.C. Smith, G.D. Peterson, Parallel application performance on shared high performance reconfigurable computing resources, Performance Modelling and Evaluation of High-performance Parallel and Distributed System 60 (1–4) (2005) 107–125, doi:10.1016/j.peva.2004.10.004.
[15] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, A.D. George, RAT: A methodology for predicting performance in application design migration to FPGAs, in: Proceedings of the High-Performance Reconfigurable Computing Technologies and Applications Workshop (HPRTCA), HPRCTA '07, ACM, New York, NY, USA, 2007, pp. 1–10. doi:10.1145/1328554.1328560.
[16] B. Holland, K. Nagarajan, A.D. George, Rat: RC amenability test for rapid performance prediction, ACM Transactions on Reconfigurable Technology and Systems 1 (4) (2009) 1–31, doi:10.1145/1462586.1462591.
[17] B. Holland, A.D. George, H. Lam, Integrating application specification and performance prediction for strategic design-space exploration, in: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), 2010, pp. 201–210.
[18] C. Reardon, B. Holland, A.D. George, G. Stitt, H. Lam, Rcml: an environment for estimation modeling of reconfigurable computing systems, ACM Transactions on Embedded Computing Systems (TECS), in press.
[19] S. Koehler, J. Curreri, A.D. George, Performance analysis challenges and framework for high-performance reconfigurable computing, Parallel Computing 34 (4–5) (2008) 217–230.
[20] J. Curreri, S. Koehler, B. Holland, A.D. George, Performance analysis with high-level languages for high-performance reconfigurable computing, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE Computer Society, 2008, pp. 23–30.
[21] J. Curreri, S. Koehler, A.D. George, B. Holland, R. Garcia, Performance analysis framework for high-level language applications in reconfigurable computing, ACM Transactions on Reconfigurable Technology and Systems 3 (1) (2010) 1–23, doi:10.1145/1661438.1661443.
[22] C.E. Cummings, P. Alfke, Simulation and synthesis techniques for asynchronous FIFO design with asynchronous pointer comparisons, in: Proceedings of the Synopsys Users Group (SNUG), San Jose, CA, 2002.
[23] D. Slogsnat, A. Giese, U. Brüning, A versatile, low latency HyperTransport core, in: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), ACM, New York, NY, USA, 2007, pp. 45–52.
[24] D. Loftsgarden, C. Quesenberry, A nonparametric estimate of a multivariate density function, The Annals of Mathematical Statistics 31 (1965) 1049–1051.
[25] B.W. Silverman, Density Estimation for Statistics and Data Analysis, Chapman & Hall/CRC, 1986.
[26] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Transactions on Information Theory 13 (1967) 21–27.
[27] J. Sanchez, J. Sotoca, F. Pla, Efficient nearest neighbor classification with data reduction and fast search algorithms, in: Proceedings of the IEEE International Conference on Systems, Man and Cybernetis, IEEE, 204, pp. 4757–4762.
[28] Y.-J. Yeh, H.-Y. Li, W.-J. Hwang, C.-Y. Fang, FPGA implementation of kNN classifier based on wavelet transform and partial distance search, in: Proceedings of the Scandinavian Conference on Image Analysis (SCIA), LNCS, No. 4522, Springer-Verlag, 2007, pp. 512–521.
[29] R. Chikhi, S. Derrien, A. Noumsi, P. Quinton, Combining flash memory and FPGAs to efficiently implement a massively parallel algorithm for content-based image retrival, in: Proceedings of the International Workshop on Reconfigurable Computing: Architectures, Tools and Applications, LNCS, No. 4419, 2007, pp. 247–258.
[30] T. Saegusa, T. Maruyama, An FPGA implementation of k-means clustering for color images based on Kd-tree, in: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2006, pp. 1–6.
[31] GNU Binutils. <http://www.gnu.org/software/binutils>.
[32] Valgrind. <http://valgrind.org>.
[33] OProfile. <http://oprofile.sourceforge.net>.
[34] T. Schumacher, E. Lübbers, P. Kaufmann, M. Platzner, Accelerating the cube cut problem with an FPGA-augmented compute cluster, in: Proceedings of the ParaFPGA Symposium, International Conference on Parallel Computing (ParCo), NICS, vol. 38, John von Neumann Institute for Computing, Jülich, 2007, pp. 749–756. <http://www.fz-juelich.de/nic-series/volume38>.
[35] T. Schumacher, Performance modeling and analysis in high-performance reconfigurable computing, Ph.D. thesis, University of Paderborn, Germany, 2011.

Tobias Schumacher received a Dipl-Inform. degree in Computer Science from University of Paderborn, Germany, in 2005. He is currently a Ph.D. candidate in Computer Science at University of Paderborn. His research focus is high-performance reconfigurable computing.

Dr. Christian Plessl earned a Ph.D. degree in Computer Engineering from ETH Zurich in 2006, and a Dipl.-Ing. degree in Electrical Engineering in 2001, also from ETH Zurich. Since 2007, he has been a postdoctoral research associate and lecturer at University of Paderborn, Germany. His current research interests include high-performance custom computing, design-methods for dynamically reconfigurable architectures, and embedded systems.

Prof. Marco Platzner earned a Ph.D. degree in Telematics from Graz University of Technology in 1996, and a Dipl.-Ing. degree in Telematics from the same university in 1991. Since 2004, he has been Professor for Computer Engineering at University of Paderborn, Germany. His current research interests include reconfigurable computing, hardware-software codesign and parallel architectures.