A Framework for PC Applications with Portable and Scalable FPGA Accelerators

Markus Weinhardt, Alexander Krieger, Thomas Kinder
Osnabrück University of Applied Sciences, Osnabrück, Germany
m.weinhardt | alexander.krieger | t.kinder@hs-osnabrueck.de

***Abstract*—This paper presents a novel framework for implementing portable and scalable data-intensive applications on reconfigurable hardware. Instead of using expensive "reconfigurable supercomputers", we focus our work on standard PCs and PCI-Express extension cards featuring Field-Programmable Gate Arrays (FPGAs) and memory. In our framework, we exploit task-level parallelism by manually partitioning applications into several parallel tasks using a communication API for data streams. This also allows pure software implementations on PCs without FPGA cards. If an FPGA accelerator is present, the same API calls transfer data between the PC's CPU and the FPGA. Then, the tasks implemented in hardware can exploit instruction-level and pipelining parallelsims as well. Furthermore, the framework consists of hardware implementation rules which enable portable and scalable designs. Device specific hardware wrappers hide the FPGA's and board's idiosyncrasies from the application developer.**

We also present a new method to automatically select a task's optimal degree of parallelism on an FPGA for a given hardware platform, i. e. to generate a hardware design which uses the available communication bandwidth between the PC and the FPGA optimally. Experimental results show the feasibility of our approach.

I. Introduction

The perfomance of today's standard PCs (based on the x86 architecture) is not sufficient for computation-intensive applications. However, using ASICs or expensive supercomputers is often infeasible. Off-the-shelf FPGA boards are a reasonably priced alternative for upgrading standard PCs. Using them, computation-intensive application kernels can be mapped to fast coprocessors and configured into the FPGAs, thereby almost reaching the performance levels of expensive specialized hardware.

Unfortunately, these FPGA boards have not yet been widely used for accelerating standard PC programs. One reason is the fact that designing digital circuits is much more difficult and time-consuming than developing software [1]. Furthermore, most hardware accelerators are hand-optimized ad-hoc solutions for one specific FPGA board. Using other FPGA boards requires time-consuming porting or complete rewriting. Hence, the high development effort is not worthwhile if, e. g. for cost reasons, FPGAs of different sizes and vendors shall be used in different workplaces of a company.

The first problem, the lack of mature high-level synthesis (HLS) methods, has been researched intensively [2], [3], but no solutions mature enough for industrial applications are available yet. Therefore we do not use HLS systems nor automatic hardware/software partitioning methods (which require HLS) in this work.

Only a few research projects tried to solve the second problem mentioned above, the lack of portable and scalable hardware accelerators, though it is recognized as a major obstacle for widespread use of FPGAs [4]. We aim to tackle this more manageable problem in the context of the *HPVis* project at Osnabrück University of Applied Sciences.[1]

Our approach is based on generic, device-independent hardware components modeled in standard hardware description languages, e. g. register-transfer level VHDL.[2] The goal is to develop a hardware-accelerated multi-process application only once and automatically adjust it to FPGA boards of varying size, performance and cost. The boards communicate with the host PC via PCI-Express (PCIe) [5].

In *HPVis*, the applications must also run on PCs without FPGA cards. This additional flexibility is achieved by a standard API which uses socket communication between software processes and - if an FPGA card is available - PCIe-based host communication including hardware wrappers for the PCIe endpoint on the FPGA boards.

The hardware platforms differ vastly (more than an order of magnitude) in the available PCIe bandwidth due to the used FPGA, the number of PCIe lanes, the PCI chipset and other factors. Furthermore, the FPGAs differ significantly in the available chip area, but not as much in the achievable design frequency. Therefore, if different FPGA boards are to be used for the same application, a big mismatch between the bandwidth of the PCIe interface and the FPGA design can occur and should be compensated. We suggest to adjust the hardware parallelism to match the available bandwidth for suitable algorithms. Our method computes the optimal parallelization degree automatically from the available FPGA resources, the PCIe bandwidth and some application-specific metrics. Using it, the best hardware implementation for the given PC/FPGA system is generated.

The remainder of this paper is organized as follows: Section II summarizes the related work. Then, our acceleration framework and the automatic system optimization method are presented in sections III and IV, respectively. Finally,

[1] HPVis: High-Performance Processing and Visualization of High-Volume Data, cf. `http://www.ecs.hs-osnabrueck.de/hpvis.html`.

[2] Note that parts of our approach will also be applicable to HLS-based design once the performance of circuits generated by HLS systems is satisfactory.

978-1-4799-2079-2/13/$31.00

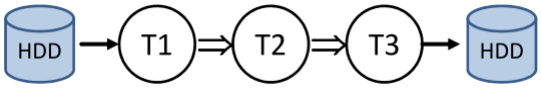

Fig. 1. Communicating parallel tasks T1-T3 accessing data on a hard disk drive (HDD).

section V summarizes our experimental results, and section VI concludes the paper and presents an outlook on future work.

II. Related Work

There is no work described in the literature which is directly comparable to our approach.

On the one hand, there are many publications on parallel high-performance computing, e. g. on applications using the MPI interface [6]. However, these applications cannot easily be ported to reconfigurable computers. On the other hand, high-performance reconfigurable applications are mainly implemented on *reconfigurable supercomputers* [7]. These systems are built from specialized modules consisting of CPUs, FPGAs and memory which are combined by high-speed connections [8], [9]. However, these systems are expensive and use proprietary *core services*, i. e. software APIs and hardware wrappers, to access the accelerator resources [10]. Some of these systems like those built by Maxeler Technologies [8] even use an own, proprietary hardware description language.

For standard PCs with standard FPGA boards as targeted in our project, the authors are not aware of a full *core services* implementation. The support packages provided by the board vendors only provide base functionality on a low level [11], [12], [13]. Only the *Speedy PCIe Core* [14] and the *RIFFA2.0* [15] projects which were published after we started our project follow a similar approach to ours also using PCIe.

[16] presents a method for managing task communication over the software/hardware boundary comparable to our API. The method schedules the communication automatically.

In general, despite a lot of research in the area of reconfigurable computing [1], there are only a few publications on portable and scalable FPGA designs [10], [17], [18]. They also suggest implementations with varying degrees of parallelism, but do not present a method for automatically determining the hardware parameters as in our framework.

III. Accelerator Framework

A. Parallel Tasks

Our applications are based on manually implemented communicating parallel tasks (or processes) as the example system shown in Fig. 1. The base implementation is entirely implemented in software and therefore runs on any PC with a single-core or multi-core CPU. The tasks communicate via data streams (arrows $\Rightarrow$ in Fig. 1) by using the API defined in section III-C. For the software implementation, the API functions are implemented with socket connections.[3]

[3]We chose a socket implementation for its simplicity over other options like MPI [6]. Furthermore, sockets also enable distributed implementations.

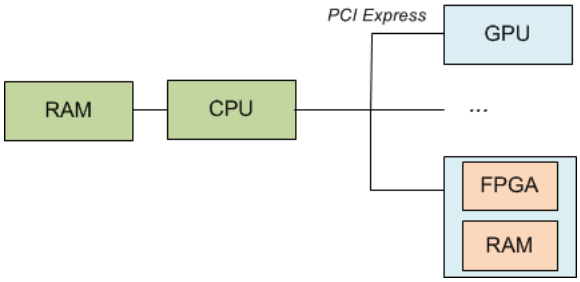

Fig. 2. Accelerator Architecture: Standard PC with PCIe cards.

B. Accelerator Architecture

Fig. 2 shows the system architecture with FPGA-based accelerators for which our framework is intended: a standard Personal Computer (PC) extended by a standard FPGA board. Both components communicate through a PCIe bus. PCIe provides fast and scalable communication. The maximum bandwidth is determined by the capabilities of both the host PC and the FPGA board. We do not impose restrictions on the PCIe lane width (x1 to x8) or protocol generations used.

An application as shown in Fig. 1 is ported to the accelerator architecture by implementing one or several tasks (e. g. T2 in Fig. 1) in VHDL on the FPGA. The remaining software tasks need not be changed. Instead, the functionality of the API changes if an FPGA board is available. Then the functions use PCIe communication. If two communicating tasks are both implemented in hardware, the API calls are replaced by direct handshakes in hardware.

Ideally, data from the PC's main memory is streamed sequentially to and from the FPGA board in parallel to the hardware processing, thereby hiding the communication latency. If required, intermediate data is stored in the FPGA's Block RAM or in the board's SDRAM.

On the hardware side, the FPGA components shown in Fig. 3 are provided by our framework. The *PCIe Wrapper* provides a standardized interface for all supported platforms and must be instantiated in all designs. Depending on the used FPGA, parts of the *PCIe Endpoint Core* are provided as hard cores or implemented as soft cores in the FPGA logic. The wrapper uses a system clock frequency f_{sys} (fixed for the individual board and platform architecture, equivalent to TRN_CLK frequency in [13]). In each system clock cycle, a data packet of a fixed *transaction width* can be generated or consumed.

On the other hand, the user logic uses an independent user clock frequency f_{user}. Specialized small FIFOs are used to cross the clock domains. On the wrapper side, their width is set to the transaction width. But the width on the user logic side can be set to other values, cf. section IV. Differing FIFO widths should be reflected by the clock frequencies to achieve balanced FIFO inputs and outputs. If, e. g., the user side width of the top FIFO in Figure 3 is half the wrapper side width, f_{user} should be twice as fast as f_{sys} in order to consume the data fast enough.

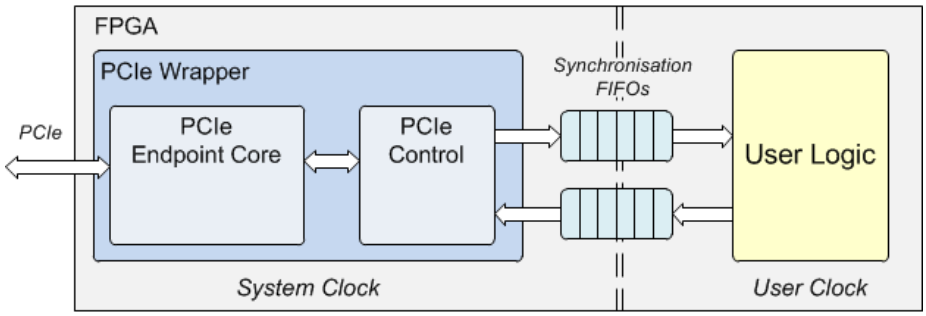

Fig. 3. FPGA Components.

TABLE I
COMMUNICATION API FUNCTIONS. ALL FUNCTIONS ARE BLOCKING AND RETURN ZERO IF THE TRANSFER WAS SUCCESSFUL.

Data Flow (DF) Transfers

`uint8_t WriteDF(uint8_t num, void *pval)`
Writes `*pval` (one 32-bit value) to FPGA user logic's stream input `num` or to corresponding socket connection `num`.

`uint8_t ReadDF(uint8_t num, void *pval)`
Reads `*pval` (one 32-bit value) from FPGA user logic's stream output `num` or from corresponding socket connection `num`.

`uint8_t StreamWriteDF(uint8_t num, void* pdata, uint16_t size)`
Copies a block of `size` bytes, starting at address `pdata`, to FPGA user logic's streaming input port `num` or to corresponding socket connection `num`.

`uint8_t StreamReadDF(uint8_t num, void* pdata, uint16_t size)`
Copies a block of `size` bytes from FPGA user logic's streaming output port `num` to host memory, starting at address `pdata`, or from corresponding socket connection `num`.

```
uint8_t StreamReadWriteDF(uint8_t rnum, void* prdata, uint16_t rsize,
                          uint8_t wnum, void* pwdata, uint16_t wsize)
```
Concurrently executes `StreamWriteDF(wnum, pwdata, wsize)` and `StreamReadDF(rnum, prdata, rsize)`, i. e. enables bidirectional (full-duplex) stream communication over PCIe or sockets.

Coprocessor Memory Accesses

`uint8_t WriteMem(uint8_t bank, uint32_t addr, void* pval)`
Writes `*pval` (one 32-bit value) to address `addr` in memory bank `bank` on board or FPGA.

`uint8_t ReadMem(uint8_t bank, uint32_t addr, void* pval)`
Reads `*pval` (one 32-bit value) from address `addr` in memory bank `bank` on board or FPGA.

`uint8_t BlockWriteMem(uint8_t bank, uint32_t addr, void* pdata, uint16_t size)`
Copies a block of `size` bytes, starting at main memory address `pdata`, to memory bank `bank` on board or FPGA, starting at `addr`.

`uint8_t BlockReadMem(uint8_t bank, uint32_t addr, void* pdata, uint16_t size)`
Copies a block of `size` bytes from memory bank `bank` on board or FPGA, starting at `addr`, to main memory, starting at address `pdata`.

C. Communication API

The API functions defined in Table I (implemented in C) are used to implement flexible applications which run as multiple software tasks on a CPU and optionally on an FPGA coprocessor board. The goal is to use as much identical code as possible on all platforms. All API functions access data by a `void*` pointer regardless of its data type. Note that only one PCIe transfer per direction can be handled at a time. It is the user's responsibility to schedule the PCIe transfers of the concurrent tasks and to guarantee that no data is modified while it is being transferred.

We first implemented the functions in the first section of Table I (Data Flow Transfers). They are used to establish data streams between the processes. The API checks if an FPGA board exists and behaves as follows:

- *FPGA board available:* transfers data to and from streaming ports of the FPGA's user logic via PCIe. Parameter `num` represents the port's number. The ports use a handshake protocol to synchronize the data flow.
- *No FPGA board:* transfers data to and from another CPU task, using socket connections. Parameter `num` is converted to a specific IP port number used for this connection, i. e. for each `num` a separate socket connection is established. For each API call, a corresponding call with the same `num` in another task is required, so that each `Write` is synchronized with a `Read` and vice versa.

For both platforms, single-word transfers (4 bytes) and stream transfers (directly from/to address `pdata`) of arbitrary `size` (in bytes) are supported. DMA transfers are used for stream transfers (and for the block memory transfers, see below). Note that the function `StreamReadWriteDF` first launches a DMA write access and then spawns a separate thread which launches a DMA read access. In this way bidirectional data flow transfers (full-duplex) are possible. If the block size is

too big to be processed by one PCIe transfer or by one socket transfer, the API implementation splits the transfer into several smaller transfers.

Single-word write or read accesses can also be used to synchronize two processes, e. g. to trigger the execution of a coprocessor function (on FPGA or CPU) or to wait for its termination, respectively.

The second section in Table I (Coprocessor Memory Accesses) are only used if an FPGA board is available. For pure software implementations, shared memory accesses should be used instead since copying data does not make sense in this case. These API functions copy data to or from memory on the FPGA board, as single-word or block transfers. The parameter `bank` is mapped to PCIe BARs (Base Address Registers) [5]. It allows to distinguishes separate memory banks (SRAM or DRAM) on the FPGA board.

Memory bank 0 is reserved for on-FPGA storage. Depending on the application at hand, address regions within this bank (or BAR) are mapped to block RAMs, distributed RAMs or even single FPGA registers. These registers can be used for setting function parameters from the host PC or for reading register values (results) from the FPGA. As opposed to the single-word data flow transfers, this allows multiple accesses to the same value. I. e., the accesses are not synchronized with the FPGA process' control flow.

Note that the stream and block functions do not specify the number of bytes transferred at a time. The number depends on the socket implementation or on the transaction width of the PCIe wrapper used.

D. Hardware Implementation Rules

In order to ensure design portability, all board and FPGA specific features are encapsulated in the PCIe wrapper. For the user logic, we impose VHDL implementation rules similar to those described in [10]. I. e., no proprietary components can be directly instantiated and no placement constraints are allowed. Instead, the synthesis system infers FPGA-specific features like multipliers, DSP blocks and on-chip memory automatically. This approach guarantees portability and does not significantly reduce the achieved performance.

Additionally, a minimum user clock frequency must be set for all supported platforms. We currently require all designs to run at f_{user} = 125 MHz. To synchronize the user logic with the PCIe data streams, a handshake protocol [19] must be used at the FIFO interfaces. This is necessary to guarantee correct functionality since the timing of the PCIe data packets are not exactly predictable and pipeline stalls may occur. The FIFOs also buffer values to compensate for varying PCIe speeds due to bursts, thereby increasing the overall throughput.

For best exploitation of differing FPGA sizes and PCIe bandwidths, the designs should be composed from *Processing Elements (PEs)* which can be replicated to process more data in parallel, cf. [17], [18]. Therefore, the user logic has a generic parameter R (replication factor) which causes the instantiation of R parallel PEs with R parallel data stream ports. R must be a power of two since it also affects the FIFO width on the user logic side. Only FIFO widths of powers of two are feasible for exchanging data with the PCIe wrapper.

Hence the performance of a hardware kernel is scaled by factor R – but so are the area and bandwidth requirements. In our framework, the optimal replication factor is automatically selected as detailed in the next section.

IV. Automatic System Optimization

This method automatically selects the optimal degree of parallelism, i. e. the best replication factor R, for a single hardware task implemented according to the rules stated above in Section III-D. Since the performance of a task is proportional to R, we select an implementation with the highest value for R which does not exceed the given PCIe bandwidth in a system and which still fits in the given FPGA area.

For automatic parameter selection, the following values must be available for all supported PC/FPGA combinations:

- $A_{Platform}$ [Slices]: Available FPGA area remaining when the PCIe wrapper is implemented.[4]
- TP_{PCI2F} and TP_{F2PCI} [MB/s]: Maximal sustained PCIe throughputs reached by the PC system. Since PCIe uses independent channels reaching different speeds for transfers to and from the FPGA board, they are handled separately.

Additionally, the designer has to provide the following information for each hardware kernel:

- *A_{PE} (FPGA area used by one PE) and A_{Const} (constant FPGA area for user logic independent of R)* [Slices]
  Hence the entire FPGA area required by the user logic is estimated to be $A \approx R \cdot A_{PE} + A_{Const}$. The values are not exact since optimzations across block boundaries may occur.
- *CW_{1toF} and CW_{1fromF} [Byte]: Channel width for data consumed or generated by the kernel, respectively, for $R = 1$.*
- *TP_{1toF} and TP_{1fromF} [MB/s]: Maximal throughputs supported by user logic for $R = 1$, in direction to and from FPGA, respectively.*
  If the kernel consumes or generates data every cycle, the throughputs are $TP_{1toF} = CW_{1toF} \cdot 119$ MB/s or $TP_{1fromF} = CW_{1fromF} \cdot 119$ MB/s, respectively, for f_{user} = 125 MHz.[5] Less frequent accesses result in lower throughputs.
- *R_{max}: Largest R supported by the implementation.*
  E. g., if the PE cannot be replicated at all for a given task, R_{max} is set to 1.

The channel widths and throughputs for multi-PE implementations are computed as $CW_{RtoF} = R \cdot CW_{1toF}$, $TP_{RtoF} = R \cdot TP_{1toF}$, $CW_{RfromF} = R \cdot CW_{1fromF}$, and $TP_{RfromF} = R \cdot TP_{1fromF}$.

[4] In the future, a more detailed area metric which separately considers logic, flipflops, arithmetic and RAM will be used. This will give better estimates for circuits heavily using specialized resources.

[5] Note the different definition of MHz and MB: A frequency of 1.048576 MHz is required to achieve 1 MB/s if one byte is transferred every cycle.

TABLE II
PLATFORM PARAMETERS.

Platform	$A_{Platform}$	f_{sys} [MHz]	TP_{PCI2F} (DMA)	TP_{F2PCI} (DMA)	TP_{PCI2F}	TP_{F2PCI}
SP605	6,701	62.5	137 MB/s	172 MB/s	31.6 MB/s	1.7 MB/s
ML605	37,074	125.0	1099 MB/s	1373 MB/s	30.6 MB/s	4.2 MB/s

TABLE III
APPLICATION KERNEL PARAMETERS.

Kernel ($f_{user} = 125MHz$)	A_{PE}	A_{Const}	CW_{1toF}	TP_{1toF}	CW_{1fromF}	TP_{1fromF}	R_{max}
Compression (for L=20)	20	27	2	238 MB/s	4	24 MB/s	4
FIR (for N=6)	7	9	2	238 MB/s	2	238 MB/s	8
Blur	18	79	1	119 MB/s	1	119 MB/s	16
Sobel	453	0	1	119 MB/s	2	238 MB/s	1

Using these numbers, the user logic's generic parameter R is automatically computed according to the following equations:

$$
\begin{aligned}
R &= min(R_1, R_2, R_3, R_{max}) \\
\text{with} \quad R_1 &= \max_{n \in \mathcal{N}} \left\{ 2^n | 2^n \leq \frac{A_{Platform} - A_{Const}}{A_{PE}} \right\} \\
R_2 &= \max_{n \in \mathcal{N}} \left\{ 2^{n+1} | 2^n < \frac{TP_{PCI2F}}{TP_{1toF}} \right\} \\
R_3 &= \max_{n \in \mathcal{N}} \left\{ 2^{n+1} | 2^n < \frac{TP_{F2PCI}}{TP_{1fromF}} \right\}
\end{aligned}
$$

Here, R_1 represents the largest circuit still fitting in the given FPGA area. R_2 and R_3 represent the smallest circuits which require at least the available PCIe bandwidths to and from the FPGA board, respectively. Implementing more PEs does not make sense since the circuit is already bandwidth-limited if R_2 or R_3 PEs are implemented. Hence, the minimum of R_1, R_2, R_3 and R_{max} represents the fastest feasible circuit not wasting FPGA resources.

R also determines the widths CW_{RtoF} and CW_{RfromF} of the FIFO ports on the user logic side. The FIFOs are instantiated with these generic parameters to automatically adjust the data stream width between the clock domains.

V. RESULTS

We performed experiments for FPGA acceleration on a Linux PC (Intel Core i5 CPU). The PC was equipped with two different FPGA boards, the Xilinx SP605 x1 lane GEN1 card (featuring a XC6SLX45T Spartan-6 FPGA) [11] and the ML605 x8 lane GEN1 card (featuring a XC6VLX240T Virtex-6 FPGA) [12]. The transaction width of the PCIe wrapper is set to 32 bits. We plan to extend it to 128 for the ML605 board in order to exploit the higher bandwidth achievable with this board when using DMA.

The following four application kernels were implemented and tested as benchmarks:

- A compression component (used in the *HPVis* project) which computes an output stream consisting of the maximum and minimum of each set of L consecutive 16-bit values in the input stream. For replication factor $R > 1$, R input values are processed at a time. (L is set at runtime.)
- A N-tap FIR filter which consumes R 16-bit input values and produces R output values in parallel. (N is fixed at synthesis time.)
- A 3×3 image blurring filter on 8-bit greyscale images. It uses a very flexible Data Window similar to the method described in [17]. The replication factor is not restricted. Since we tested the filter up to $R = 16$, R_{max} is set to 16 in Table III.
- A 3×3 Sobel filter (x and y direction) on 8-bit greyscale images which converts the gradient to polar coordinates and returns two values, magnitude and angle. The implementation is restricted to one PE, i. e. $R_{max} = 1$, but could be extended to use a flexible Data Window as *Blur* above.

Table II presents the parameters of the used hardware platforms. Note that the values for DMA transfers and the DMA performance results in Table IV are estimated since – while our PCIe wrapper implementation is running – the drivers do not yet allow performance measurements. The transfer rates were estimated by the method suggested in [20].Nevertheless we tested the benchmarks for functional correctness with a preliminary API implementation using single-word accesses for stream transfers. The measured throughputs are shown in the two rightmost columns of Table II. Obviously the protocol overhead incurred for every single word significantly reduces the transfer rates.

The application kernels' parameters for determining the optimal replication factor R are summarized in Table III.

Finally, Table IV shows the achieved performance (i. e. replication factors, channel widths and transfer rates), the speedup over a single-PE kernel (determined by throuhput comparison), and the required FPGA area for DMA stream transfers on the ML605 board. Note that for this board, $f_{sys} = f_{user}$ = 125 MHz.

For *FIR* and *Blur*, the FIFOs are 128 bit (16 byte) wide on both sides. Nevertheless the user logic cannot achieve its full speed since the PCIe wrapper cannot produce and consume 128-bit packets every cycle. But since $TP_{RtoF} > TP_{PCI2F}$ and $TP_{RfromF} > TP_{F2PCI}$, the user logic can process data from the PCIe interface as soon as it is available, i. e. the

TABLE IV
RESULTS ON ML605 BOARD USING DMA TRANSFERS.

Kernel	R	CW_{RtoF}	TP_{RtoF}	CW_{RfromF}	TP_{RfromF}	$Speedup$	A
Compression	4	8	954 MB/s	16	95 MB/s	4.0	105
FIR	8	16	1907 MB/s	16	1907 MB/s	4.6	37
Blur	16	16	1907 MB/s	16	1907 MB/s	9.2	321
Sobel	1	1	119 MB/s	2	238 MB/s	1.0	453

given PCIe bandwidth is optimally used.

The other kernels would benefit from implementations with higher R_{max} since they cannot consume or produce data as fast as the PCIe wrapper does. But note that, for *Compression*, the FIFO inputs and outputs to the FPGA are nearly balanced: $TP_{RtoF} \approx TP_{PCI2F}$. The FIFO port on the user logic side is only 64 bit (8 byte) wide, but consumes data every cycle, while the PCIe wrapper produces 128-bit packets only about every other cycle.

The results show that the computed parameters optimally exploit the PCIe bandwidth. By parallelization, the PCIe bandwidth and the streaming bandwidth of the application kernels are aligned approximately. The parallel kernels achieve a speedup of up to 9.2 over the single-PE versions, and the area of the circuits is well within the available FPGA area. Faster PCIe interfaces will allow even more parallelism and higher speedups.

VI. CONCLUSIONS AND FUTURE WORK

We presented a novel framework for implementing portable and scalable applications on standard PCs with and without PCIe-based FPGA boards. It consists of hardware implementation rules, a communication API along with corresponding hardware wrappers, and a method to automatically select an application's optimal replication factor R for a given hardware platform. Experimental results show the feasibility of our approach.

After finishing the PCIe driver supporting DMA, we will implement the memory access functions. In the future, the automatic optimization method will be extended as follows: (1) consider memory bandwidth as well, and (2) allow more hardware parameters, e. g. two replication factors to represent two-dimensional PE arrays or a pipelining factor to vary the number of pipeline stages in a design. Combined with a parameter representing the speed of an FPGA, the latter extension would allow us to better exploit the maximum performance of a given FPGA type. Furthermore, we plan to automatically extract some of the parameters from the VHDL designs and to support more hardware platforms, possibly including other FPGA vendors. Another important issue is to explore possibilities for direct communication between an FPGA board and other PCIe devices, e. g. GPUs, thus further reducing the communication overhead for visualization applications.

ACKNOWLEDGEMENT

This work is supported by the European Regional Development Fund and the Lower Saxony State Government/Germany in the research project *HPVis: High-Performance Processing and Visualization of High-Volume Data (Hochperformante Verarbeitung und Visualisierung von Massendaten)*.

REFERENCES

[1] C. Bobda, *Introduction to Reconfigurable Computing*. Springer, 2010.
[2] D. D. Gajski, N. D. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer, 1992.
[3] J. M. P. Cardoso, P. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey," *ACM Computing Surveys*, vol. 42, no. 4, June 2010.
[4] T. El-Ghazawi, "The challenges of computing with FPGAs," May 2011, Keynote speech at Reconfigurable Architecture Workshop (RAW 2011, Anchorage/USA) and personal communication.
[5] R. Budruk, D. Anderson, and T. Shanley, *PCI Express System Architecture*. Addison Wesley, 2004.
[6] J. Dongarra and M. Walker, "MPI: A Standard Message Passing Interface," *Supercomputing*, vol. 12, no. 1, Jan. 1996.
[7] T. El-Ghazawi, E. El-Araby, and M. Huang, "The promise of high-performance reconfigurable computing," *IEEE Computer*, February 2008.
[8] O. Lindtjorn, R. Clapp, O. Pell, O. Mencer, M. J. Flynn, and H. Fu, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, March/April 2011.
[9] SRC Computers LLC, "Reconfigurable MAP processors," www.src computers.com.
[10] P. Saha, E. El-Araby, M. Huang, M. Taher, S. Lopez-Buedo, T. El-Ghazawi, C. Shu, K. Gaj, A. Michalski, and D. Buell, "Portable library development for reconfigurable computing systems: A case study," *Parallel Computing - Systems & Applications*, May 2008.
[11] "Spartan-6 FPGA SP605 Evaluation Kit," www.xilinx.com.
[12] "Virtex-6 FPGA ML605 Evaluation Kit," www.xilinx.com.
[13] "XAPP1052 - Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions," www.xilinx.com.
[14] R. Bittner, "Speedy bus mastering PCI Express," in *Proc. 22nd Int. Conf. on Field Programmable Logic and Applications (FPL 2012)*, Aug. 2012.
[15] M. Jacobson and R. Kastner, "RIFFA 2.0: a reusable integration framework for FPGA accelerators," in *Proc. 23rd Int. Conf. on Field Programmable Logic and Applications (FPL 2013)*, September 2013.
[16] M. King, A. Khan, A. Agarwal, and O. Arcas, "Generating infrastructure for FPGA-accelerated applications," in *Proc. 23rd Int. Conf. on Field Programmable Logic and Applications (FPL 2013)*, September 2013.
[17] M. Huang, O. Serres, S. Lopez-Buedo, T. El-Ghazawi, and G. Newby, "An image processing architecture to exploit I/O bandwidth on reconfigurable computers," in *Proc. 4th Southern Conf. on Programmable Logic*, 2008.
[18] M. Huang, O. Serres, T. El-Ghazawi, and G. Newby, "Parameterized hardware design on reconfigurable computers: An image processing case study," *International Journal on Reconfigurable Computing*, 2010.
[19] B. Lang, "Self arbitrating elements for modelling systolic dataflow in field programmable gate arrays," in *GI/ITG-Workshop Anwenderprogrammierbare Schaltungen*, Karlsruhe, Germany, July 1994.
[20] A. Goldhammer and J. A. Jr., "WP350 - Understanding Performance of PCI Express Systems," 2008, www.xilinx.com.