Journal of Signal Processing Systems (2019) 91:745–757
https://doi.org/10.1007/s11265-018-1382-7

Heterogeneous Computing Utilizing FPGAs

A New and Flexible Approach Integrating Dedicated Hardware Accelerators into Common Computing Platforms

Marc Reichenbach[1] · Philipp Holzinger[1] · Konrad Häublein[1] · Tobias Lieske[1] · Paul Blinzer[2] · Dietmar Fey[1]

Received: 15 December 2017 / Revised: 21 March 2018 / Accepted: 15 May 2018 / Published online: 31 May 2018

Abstract
Heterogeneous computing plays an ever-increasing role in power-efficient, high-performance embedded systems for various data processing tasks, such as computer vision. One possibility to accelerate this kind of application is the usage of FPGAs as a co-processor for standard CPUs. Although hardware design is becoming easier by utilizing High-Level-Synthesis tools, the question of interfacing FPGAs and CPUs has yet to be completely solved. The Heterogeneous System Architecture (HSA) Foundation defines and publishes architecture neutral standards for heterogeneous systems and programming models. While compatible CPU, GPU and DSP designs exist, FPGA models have not been defined yet. This paper describes the IP library LibHSA, which greatly simplifies integration of domain specific FPGA acceleration into existing HSA compliant systems. It allows FPGA based accelerators to take immediate advantage of high-level language tool chains. Including user space memory access, low-latency task dispatch and other benefits of the HSA programming model. We will demonstrate LibHSA with a programmable image processor implementation on a Xilinx FPGA. The image processor supports low-level algorithms, e.g. Sobel, Median, Laplace, or Gaussian. Our results show that the LibHSA infrastructure greatly simplifies the effort integrating FPGAs and customized hardware into existing accelerator systems, runtimes and application software.

Keywords Heterogeneous system architectures · HSA foundation · Hardware accelerator · Image processing · FPGA

1 Introduction

Advancements in many different fields such as data mining, virtual reality, (medical) image processing or the recent rise in deep learning methods demand fast processing of exponentially larger workloads. This phenomenon, also known as the "curse of dimensionality", challenges traditional systems, requires new ways of handling highly compute intensive tasks [13] and therefore strongly drives the upsurge of heterogeneous compute platforms in research and industry.

Since several years GPU compute has seen the strongest uptake compared to FPGAs due to their easier accessibility and the wider support of high-level languages. However, FPGAs can significantly improve the performance per watt for targeted workloads versus CPUs [17] or other accelerators [20]. Currently the main impediments of FPGAs are the programming model and the integration into the larger software infrastructure, among other things due to proprietary support software. Some academic and commercial solutions to this problem, e.g. from Altera or Xilinx, are available, but do not cover typical, multi-component heterogeneous system configurations needed for optimal results [20]. Even today after years of research, setting up, programming and maintaining a heterogeneous platform is by no means trivial, but requires significant

✉ Marc Reichenbach
marc.reichenbach@fau.de

Philipp Holzinger
philipp.holzinger@fau.de

Konrad Häublein
konrad.haeublein@fau.de

Tobias Lieske
tobias.lieske@fau.de

Paul Blinzer
paul.blinzer@amd.com

Dietmar Fey
dietmar.fey@fau.de

[1] Chair of Computer Architecture, Friedrich Alexander University Erlangen-Nürnberg (FAU), Martensstraße 3, 91058 Erlangen, Germany

[2] Advanced Micro Devices (AMD), Bellevue, WA, USA

cooperation between many different hardware and software companies.

To approach this problem members of industry and academia formed the HSA Foundation[1]. They defined a performant yet standardized system and hardware model definition for data parallel processors and accelerators in high-level language (HLL tool chains). By targeting an abstracted hardware programming model, high-level languages can directly take advantage of the underlying accelerators with much lower dispatch overhead compared to common API based models like CUDA or OpenCL.

The HSA infrastructure provides many software environment tools, like compiler toolchains for C++, Python, Fortran and many more. A compatible accelerator can take advantage of the existing platform, greatly simplifying software and platform integration, but designing the hardware accelerator itself remains the vendors' responsibility. Fortunately, this task can be divided into three subtasks:

(1) the development of the accelerator core itself
(2) the development of the hardware dispatch model to make the accelerator HSA compliant
(3) the implementation of a fast bus infrastructure to connect the host and accelerator

While (1) is application-specific and has to be achieved by the developer, tasks (2) and (3) can be automated because it shares similarities with every new hardware accelerator. Therefore, a library to ease the development process of arbitrary hardware accelerators is needed. This especially improves the developer productivity of FPGAs by supplying blocks for (2) and (3). This paper focuses on the second point above by presenting a novel library called *LibHSA*. Using this enables the developer to integrate new accelerators into a heterogeneous system easily by making it compatible with the HSA environment.

This paper is structured as follows. In the next Section related work is discussed. Section 3 describes the key concepts of HSA and how LibHSA is implemented. Afterwards, Section 4 shows how LibHSA can be used to make an accelerator HSA compliant. This is demonstrated with an image processing core and implemented on a Xilinx FPGA. Section 5 presents the implementation and timing results of our prototype. Finally, Section 6 concludes our work and provides an outlook to future work.

2 Related Work

One of the first works to describe the HSA system architecture is provided in [19]. This publication shows various key concepts of the programming model, the unified address space, queuing, context switching as well as the benefit of HSAIL. A first performance analysis of HSA on real hardware was first provided in 2016 [14]. There, they compare the execution times for different algorithms on GPGPUs utilizing OpenCL and HSA. Furthermore, they developed a framework, called HeteroMark, to allow benchmarking heterogeneous hardware. Overall, the HSA runtime environment provides very promising results for real world applications.

In 2017 AMD released Vega, a new HSA compliant high-performance GPU architecture. It contains the so-called High-Bandwidth Cache Controller (HBCC) which operates as unified memory cache [15]. LibHSA also brings this important feature to FPGA accelerators now.

With [3] a conceptual work was provided, for the use of HSA with DSPs. This work focuses on how the HSA execution model can be mapped to these devices. For application purposes an FIR (Finite Impulse Response) filter was shown. To our knowledge Soft-IP-blocks for connecting self-designed circuits to the HSA universe were not provided in the past. Therefore, to date, no FPGA-based hardware has been available.

Nowadays FPGAs use several different software models in heterogeneous systems. Commercial examples are the developer environments *Altera OpenCL* [11] and *Xilinx SDAccel* [4]. Both attain good results and significantly ease the integration processes. Unfortunately, these often use proprietary, limited and intransparent mechanisms which makes it impossible to extend them beyond their original scope.

Furthermore, academic approaches, as shown in [10], have been developed as well to generate application-specific hardware accelerators from a high-level description (OpenCL). Unfortunately, this generated hardware suffers from the self-integration problems described above.

Aside from generating hardware through OpenCL further work shows how specialized programmable architectures can be integrated into a computing system. The researchers in [18] demonstrated specialized vector unit integration, and targeted it through C source. However, it does not integrate well with other accelerators in a heterogeneous system. An FPGA cluster system based on GBit Ethernet interconnects and a lightweight processor architecture has been demonstrated in [2]. Here, the system is only accessible by own C language extensions and more powerful custom accelerators cannot be easily integrated. Moreover, the Ethernet infrastructure might become the systems bottleneck in case of heavy data transfer. Both designs would benefit from integration through a common heterogeneous system interface as developed in our work.

[1] http://www.hsafoundation.com/

3 LibHSA Conception

3.1 HSA System Architecture Specification

As mentioned above, LibHSA is based on the three main specifications published by the HSA Foundation:

- The *Platform System Architecture Specification* [7] defines the hardware model software is targeting
- The *Programmers Reference Manual* [9] specifies the HSAIL instruction set and the software model compilers are expected to target.
- The *Runtime Programmers Reference Manual* [8] defines the resource management APIs a compatible language runtime is expected to support.

In the following all necessary steps for successfully launching a kernel from software and processing it in the hardware are summarized. For further details the HSA specification [6] can be consulted.

Each participant of an HSA compatible system is referred to as an *agent*. Devices, capable of executing parallel regions of code, are called *kernel agents*. These are typically GPUs, but DSPs and, through our work also FPGAs, can act as kernel agents. The two core concepts of an HSA system are the queueing system and the *Architected Queuing Language* (AQL) packets. Each packet has a length of 64 bytes and represents a task the device has to perform. There are different kinds of packets which can be distinguished by a common header. First of all it can be a *kernel dispatch packet* which contains meta information associated with a kernel dispatch. The second type is the *agent dispatch packet* used to call special built-in functions. Thirdly *barrier AND/OR packets* contain synchronization primitives to realize complex dependencies. In general AQL packets symbolize the concept of "what should be done" in the system, but not by whom. The distribution of tasks to different accelerator cores is controlled by the AQL queues. Each device can have one or more AQL queues associated with it. The hardware is responsible for executing the tasks in them and maintaining the state of the queue on its own. This job is usually done by a special unit, the *packet processor*. Now processes can directly interact with the hardware by a thin runtime environment.

The general procedure for launching a kernel can be seen in Fig. 1. At first a slot in a queue, belonging to a kernel agent capable of executing the task, must be reserved. This can be done by atomically incrementing the write index of the queue. All meta-information of the kernel is encoded in the kernel dispatch packet. This means the number of dimensions, the kernel length, the workgroup size, the kernel arguments and the kernel executable, are fields in the packet and written to the allocated memory

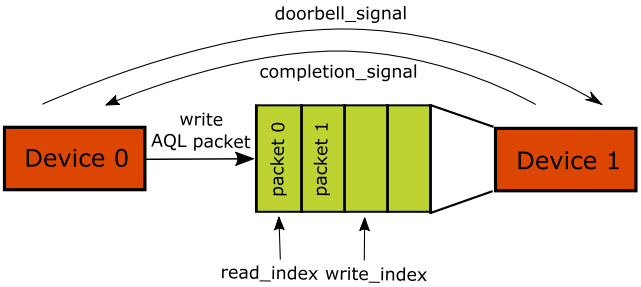

Figure 1 Illustration of the HSA queuing system. Distributing work to a kernel agent can be done in four steps [7]: (1) *Allocating* a packet slot by incrementing the write_index. (2) *Updating* the kernel details like arguments or workgroup size. (3) *Assigning* the packet to the packet processor by writing the packet header. (4) *Notifying* the packet processor by sending the doorbell_signal. As a last step the caller of the kernel is notified by packet processor through the completion_signal that the results are ready.

location of the packet or further referenced memory. Due to the unified address space, required by the HSA specification, host pointers can be used directly and explicit copying is not needed. After submitting the packet, the ownership can be transferred to the device by setting the header and specifically the type field to the correct value via an atomic write operation. From then on the packet can, but does not necessarily have to be processed at any time by the packet processor. In addition the *doorbell_signal* must be sent to notify the accelerator in order to check the queue for new work. Again this is done by an atomic write operation. Eventually the hardware evaluates the packet and takes all necessary steps to execute it. The packet state transitions can be seen in Fig. 2. After reaching the completion phase the packet processor notifies the runtime via a *completion_signal*. Finally the submitting process can work with the results of the kernel invocation.

3.2 Library Overview

As described in the previous Section 3.1 certain components must be present in all HSA compliant hardware. Our LibHSA abstracts these by providing dedicated hardware and software to directly address these parts on the one hand and on the other hand the definition of highly flexible interfaces to support all kinds of accelerator and bus types. An overview of all system components is summarized in Fig. 3. The most important component is the packet processor, because it handles the entire task processing, scheduling and management of memory fences and accelerator cores. It is the core part of the *Packet Processing* section and a *fixed* element in our library and the HSA ecosystem, independent of the

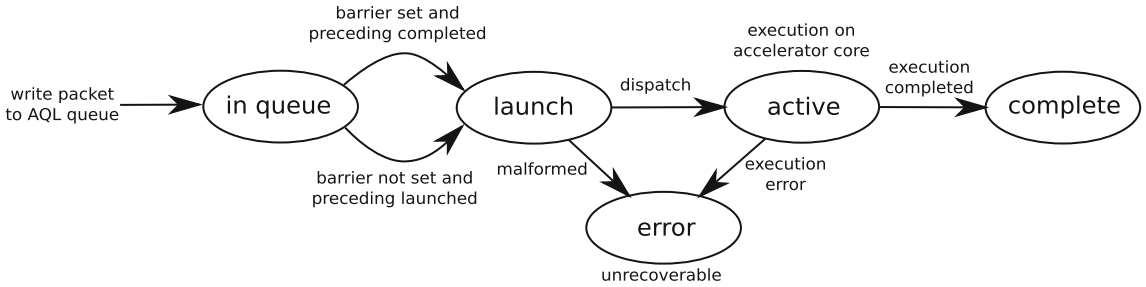

Figure 2 States an AQL packet can be in after enqueuing to a kernel agent as described in [9]. After the assignment of the task to the packet processor it eventually reaches the *launch* phase. Depending on the barrier bit in the packet header this can be delayed until the previous completed execution. In this state the task will be parsed but has not started the actual execution yet. This does not happen until the packet enters the *active* state. Finally, after the data has been completely processed it reaches the *completion* phase where the notification of the host takes place. If an error occurs during launch or execution the packet as well as the AQL queue reaches the *error* state.

accelerator hardware. This processor must support 64-bit addressing and atomic instructions to comply with the HSA specification. Additionally it must be as small as possible to minimize the resource overhead of the system. For that reason we designed our own processor based on the MIPS III ISA [12]. To our knowledge no other light-weight, synthesizeable 64-bit processors with atomic read/write operations are available at the moment. The open design concept of the MIPS core also makes it easily transferable to other manufacturing platforms like *Altera* FPGAs.

In the *Accelerator Hardware* section the actual accelerator is instantiated. The example application builds a

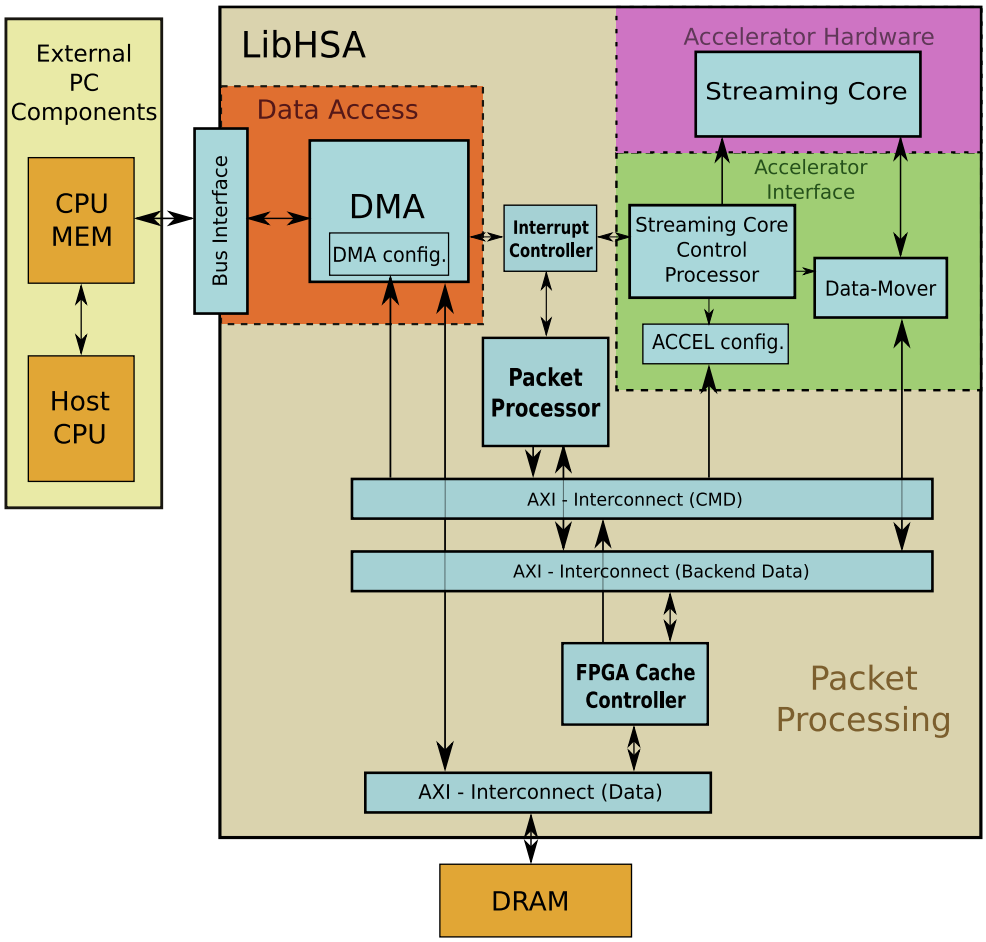

Figure 3 Overview of library segments and components. Segments with dashed borders have flexible or interchangeable components. All subblocks inside the LibHSA box represent IP cores instantiated on reconfigurable hardware. All other areas mark HSA components outside the framework, on the FPGA or off-chip.

streaming core for basic image processing operations, which will be explained in detail in Section 4. The *Accelerator Interface* is partly a flexible component. In our use case it is specialized for streaming operations, but it must be also exchangeable for other processing schemes. For this purpose, we supply additional compatibility components. The *Streaming Core Control Processor* is a MIPS as well, but to be as lightweight as possible it is based on the 32-bit MIPS I ISA from [5]. It controls a data-mover, responsible for transforming the memory-mapped interface to and from a streaming interface. Additionally, the control processor sets the correct configuration for the streaming core. In this way LibHSA provides two different interfaces for streaming and memory-mapped access. In order to make our components even more flexible all soft core components are interconnected by AXI based buses. This is an on-chip bus specification defined by *ARM* and used by *Xilinx* and *Altera* to connect various IP cores. Therefore all accelerator cores can be easily exchanged with other units.

Packet processor and accelerator cores together constitute the actual part where submitted kernels are processed. This is needed for all kinds of tasks, platforms and processing schemes. More complex memory management requirements need the additional interfaces and hardware explained in the following.

First of all the *Data Access* section connects the FPGA to the *External PC Components* and enables DMA transfers to and from the board. For this part PCIe is the established high-performance solution, but other bus systems would also be possible. Therefore, the corresponding section is a *flexible* part, but has a *fixed* interface to our core LibHSA.

The other important component of LibHSA is the *FPGA cache controller* (FCC). It is responsible for realizing the shared virtual memory (SVM) required by the HSA System Architecture specification from the FPGA hardware side. All data transfers to and from the working set are routed through this unit. It checks whether the requested data, represented by address and process address space ID (PASID), is present on the FPGA board. If absent FCC sends a DMA transfer command to the data access section to fetch this memory block. The actual block size can be configured beforehand to be an arbitrary power of two. Thereby the on-board DRAM acts as a large cache to speed up subsequent data accesses to the same region. Moreover this mechanism also hides the PCIe latency and transfer times of the complete system, because other accelerator cores can still work in parallel on data that is already local. To ensure data visibility both on host and accelerator side FCC also incorporates logic to establish acquire and release memory fences. These fences are PASID specific and the packet processor lets FCC apply them if stated in an AQL Packet. This means the host can improve the overall performance by always requesting only the weakest fence applicable in every situation. All in all FCC makes it possible from the hardware side to process pointers to regions which are in general not trivially accessible to accelerator cores. This makes it especially useful for non-uniform or unpredictable memory access patterns. In cases where data is processed in a streaming fashion from contiguous chunks of memory, e.g. for basic image filters, the additional resources for the FPGA cache controller may not be worth it. In these situations the functionality can also be provided by a custom software for the packet processor itself.

3.3 Platform Integration

As required by the HSA specification presented in Section 3.1, we use a kernel dispatch packet to represent an image processing task for our *Image Processing Streaming Core*. As usual the image size is entered in the dedicated grid size fields of the packet and a completion signal is set. But there are several differences in operation between a GPU and FPGA execution model here. First of all the custom FPGA accelerator design is not based on the concept of a workgroup at execution time. Therefore, the corresponding fields in the packet remain unused. However, this is not a limitation, because a similar functionality can already be realized in the hardware core itself. Secondly in contrast to GPUs, application-specific cores are usually not capable of executing arbitrary code. In our case the executable would always contain just a single "instruction" differentiating the local image operation to be used. Therefore, we directly encode this in the *kernel_object* field. This is also not a limitation since it is always possible to perform multiple successive operations with negligible performance penalties by using several kernel dispatch packets for each task and caching the data on the FPGA. Lastly a custom acceleration core on an FPGA typically has a predefined set of kernel arguments. In our case these are the source and destination start addresses, the color model of the image, as well as a value threshold and optionally up to two individual masks and normalization factors. These arguments are encoded in the allocated memory space and pointed to by the *kernarg_address* field.

To increase the overall performance and latency of the system, we decided to implement an asynchronous three-step task messaging system to connect all components. At first the initiator writes the message content to a special memory region of the receiving partner. Secondly the receiver is notified by an interrupt. Lastly the transmitter eventually receives an interrupt, signaling that the other unit executed the task. Note that the interrupts are also individually acknowledged by an ACK signal to enable clock domain crossing between different components. As visualized in Fig. 4 all new packets are written to the AQL queue in the first kilobytes of the on-board DRAM and

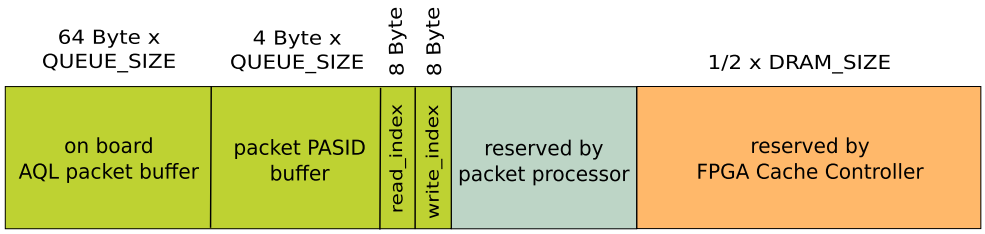

Figure 4 Address space layout of the on-board DRAM. It is embedded in the full 64-bit address space of the whole system and accessed with a flat 64-bit address. The Process Address Space Identifiers (PASID) are located directly after the packets in the same order.

signaled by an interrupt. The packet processor interprets the new packets one by one. Together with the FPGA cache controller they are capable of applying acquire and release memory fences as well as requesting data copies to the on-board DRAM. This is an important ability for the integration into the main memory system of a whole PC. In particular kernel arguments or in our case images can be asynchronously fetched by sending a request as explained before to a DMA unit. For this purpose, we defined the message as a host address, an FPGA address, the payload size in bytes, the transfer direction and a PASID identifying the process which submitted the packet. This makes it possible to increase the overall throughput by processing independent packets simultaneously. Once all data is present, the packet processor issues a processing command by sending a message to an available core. In our case the streaming core control processor then addresses all internal configurations of the PE and data-mover, while the actual processing is performed as explained in Section 4.

After completing the operation, the data is fully updated, but not yet visible to the caller of the kernel function. For this purpose, the packet processor and FPGA cache controller again apply a memory fence and request the DMA interface to copy the results back. Finally, the caller can be notified by sending the *completion_signal*. This time the message contains the signal handle and the PASID. Depending on the bus used this action could involve an atomic operation over PCIe. This high latency again confirms that our asynchronous messaging approach is not only beneficial but necessary in such a system.

4 Use Case: Image Processing

In order to demonstrate our LibHSA, we coupled an FPGA-based accelerator to the accelerator interface. On FPGAs algorithms are very often accelerated by designing an application specific circuit, benefiting from processing schemes like pipelining and/or parallel execution units. However, these kind of accelerators are hardly flexible and fixed to one algorithm with one set of parameters, which would not support processing various dispatch packets. We therefore decided to design a weakly programmable accelerator core for the domain of local image processing operations. These kind of operations require a proper acceleration for various real-time applications, achieved by utilizing custom-designed memory structures such as line buffers to enable streaming. Most common examples are linear operators like the Gaussian Blur, the Sobel or Laplace operator and non-linear operators like the Median operator. The accelerator is designed generically, which makes it customizable in static image parameters (e.g. maximum supported window size) as well as available resource constraints in terms of logic and memory cells. In Fig. 5 the architecture of one processing element (PE) is shown, which can be seen as the core of the entire accelerator unit.

The architecture is fully pipelined and able to produce one pixel within each clock cycle. A PE is divided into two main sections. The Buffer CLB (Configurable Logic Block) holds the *Full Buffering* architecture derived from the generic template from [16] consisting of line buffers and mask registers. Line buffer intersections in Fig. 5 indicate the configurability of the image size within this structure. After instantiating the architecture with a maximum image size (e.g. 1920×1080) all lower values are supported through proper configuration. The mask registers have a similar behavior. In Fig. 5 the maximum mask size is set to 5×5. All mask values are passed in parallel to the Kernel CLB. In the Kernel CLB the image processing operation takes place. Several kernel modules have fully parallel access to the mask registers. The CLB includes two convolution modules. Each is able to perform a MAC operation (multiply and accumulate) within one cycle through a binary tree construct. As a second input each unit is connected to a *Configuration Matrix*, where coefficients for linear operations are stored. The second convolution module is necessary for edge detection operations, which have two coefficient matrices as an input for detecting edges in the x and y-direction. The units *Normalize* and *Threshold Detect* are responsible for the correct pixel output. Normalization is required for the Gaussian Blur, while a threshold enables binary output for edge detection. To support non-linear image processing operations a fully

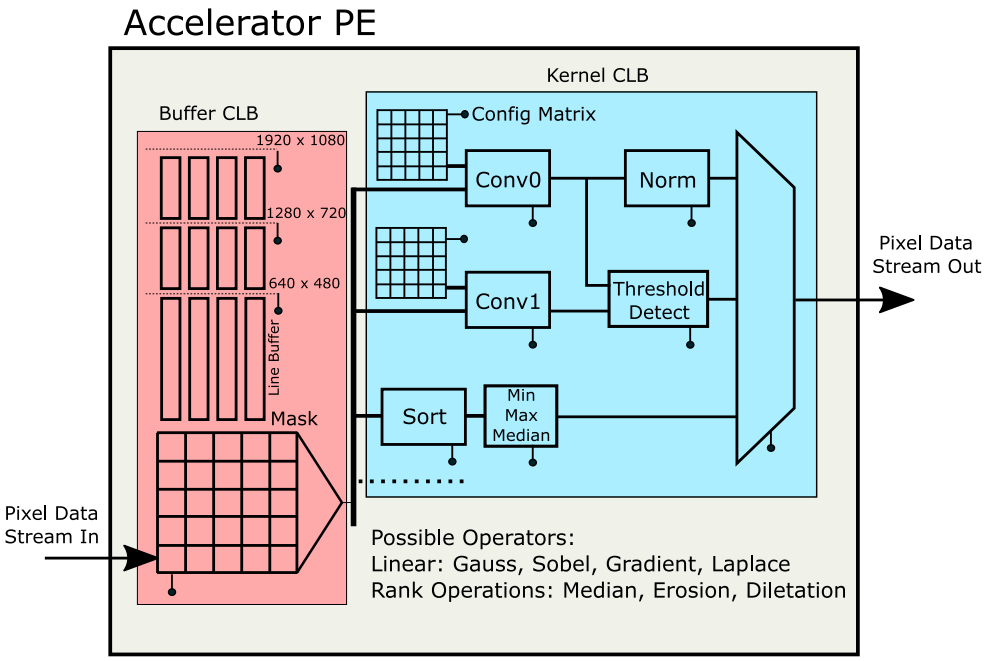

Figure 5 Internal architecture of one PE. Buffer CLB consists of mask and line buffers. The Kernel CLB has parallel access to the mask and is split into several computational units. With the proper configuration various local operations can be applied.

parallel *Bitonic Sort* structure can be accessed. By selecting the proper sorted output value the operations Median, Erosion and Dilation can be executed. Additional custom made kernel modules could be easily added to the design. All units in the design are programmable, which is indicated by black bubbles connected to the dedicated unit. Values for selecting the proper function are stored in a 32-bit-wide configuration register file.

The complete accelerator design holds multiple PEs for multicolor support, and uses several AXI interfaces for connecting to the outside world. Through an AXI light interface the configuration registers can be set. AXI stream master and slave components constitute the interconnect for streaming data to and from the accelerator. In our current implementation images with a resolution up to 3840×2160 can be processed. Local operators may range from 2×2 to 5×5 windows for all operators. The number of color streams has been fixed to 3, while each processing element can handle 8 bits per channel. Switching to single gray scale enables 16-bit processing. The instantiated kernel modules allow linear operations like Gaussian Blur, Laplace operator, or Sobel operator. Through the sort module, rank operations like Median, Erosion and Dilation can be applied to the image.

5 Results

5.1 Synthesis Results

We prototyped our design using a *Virtex UltraScale VCU108* board and *Vivado 2017.2*. To test the functionality of our design we used a separate soft core processor and BRAM to simulate a host, distant memory and DMA unit combined. This is by no means an adequate replacement in a full-fledged heterogeneous system and therefore a PCIe DMA unit will be added in the future. This part will not be shown in any of the following results, since it is not a core part of our library. Nevertheless, we could show that LibHSA works on real FPGA hardware.

In a configuration with one accelerator core LibHSA itself consisted of 9761 LUTs, 7464 flip-flops and 18.5 BRAM blocks, as shown in Table 1. This includes the packet processor (PP), the interrupt controller (IC), the FPGA cache controller (FCC) as well as the bus infrastructure. These values are mostly independent from the concrete attached accelerator core, because the interface to them is always the same. However, minor deviations can be caused by cross boundary optimization and critical cell replications. This is the fixed part of our design and with less then 2% of available LUTs and FFs comparatively small. While managing more than one accelerator, the core components do not need to be replicated. This was achieved by directly

Table 1 Implementation Results of fixed Library Components for one Accelerator.

	LUTs	FFs	BRAM	DSPs
PP & IC	6199 (1.2%)	4820 (0.4%)	6 (0.3%)	0 (0.0%)
infrastructure	1850 (0.3%)	1705 (0.2%)	2 (0.1%)	0 (0.0%)
LibHSA basic	8049 (1.5%)	6525 (0.6%)	8 (0.5%)	0 (0.0%)
FCC	1712 (0.3%)	939 (0.1%)	10.5 (0.6%)	0 (0.0%)
LibHSA extended	9761 (1.8%)	7464 (0.7%)	18.5 (1.1%)	0 (0.0%)

Table 2 Scaling Results of LibHSA.

Accel Cores	LUTs (per core)	FFs (per core)	BRAM
1	9761 (9761)	7464 (7464)	18.5
2	10190 (5095)	7537 (3769)	19.5
3	10457 (3486)	7584 (2528)	20.5
4	11126 (2782)	7604 (1901)	21.5

tracking multiple AQL packet states in the packet processor and handling all associated interrupts asynchronously. These are sequentially executed on the main MIPS core. Additionally, AQL interrupts are immediately preprocessed by dedicated hardware components in the packet processor. However, since additional multiplexers and larger AXI interconnects are needed the resource utilization slightly increases, as shown in Table 2. Nevertheless, this led to a negligible amount of additional resources. In particular the resource expenditure of LibHSA per core decreased rapidly. This shows that LibHSA is well suited to function as a dispatch infrastructure and leaves a great deal of room for the actual accelerator cores.

The complete accelerator core (AC) in our image processing use case needed 33296 LUTs, 28474 flip-flops, 47.5 BRAM blocks and 50 DSPs. Table 3 shows, that the image processing streaming core (Image PE) was responsible for roughly 80% of this utilization and nearly the entire BRAM and DSPs. This again means that our interface is slim and leaves the important resources for functional units.

5.2 Performance Results

All major parts (packet processor, image processing accelerator core, etc.) of our test design can be clocked asynchronously to achieve the maximal possible performance of each component. But for simplicity and comparability all clocks have been constrained to 100 MHz in our working demonstrator board.

The latencies introduced by our design depend on the bus utilization, as well as quantity and kind of incoming interrupts. Therefore, it is not possible to determine exact numbers for all cases beforehand. However, the following results show typical latencies for various scenarios which are also representative for other usecases.

Table 3 Implementation Results of Accelerator Core Components.

	LUTs	FFs	BRAM	DSPs
Image PE	26864 (5.0%)	22467 (2.1%)	34 (2.0%)	50 (6.5%)
AC infrastructure	6432 (1.2%)	6007 (0.6%)	13.5 (0.8%)	0 (0.0%)
AC	33296 (6.2%)	28474 (2.6%)	47.5 (2.7%)	50 (6.5%)

5.2.1 Packet Processor Stress Test

First, the overhead of our dispatching scheme needed to be measured. Since high PCIe latencies would distort the results we forgo them at the moment. For this purpose, we restricted the test scenario to a situation where all data needed was already present on the board, so the FPGA cache controller was not needed. This is also a realistic use case, commonly seen in MPSoC systems with an ARM processor as host. To put the maximal load on the packet processor we also limited the eight available accelerators to perform simple vector copy operations with very few work. Here the copy was simply a single read and write access on the bus with minor parameter parsing and setup time. Therefore, very shortly after the dispatch, the packet processor was confronted with an answer form the accelerator core. In this setup we fixed the AQL queue size to 128 packets and configured our MIPS host processor to write 1000 kernel dispatch packets to the queue. We chose to write them one by one and not as a batch submission to further increase the bus traffic. Every time when no more slots were available in the ring buffer, the host had to wait until an older task has finished, in accordance with the HSA standard. In this setup shown in Fig. 6 accelerator core zero received the first job after $14.51\mu s$. In comparison the interval without any additional bus traffic amounted to a mere $14.06\mu s$. This difference was directly caused by the additional time needed for the packet processor to fetch the packet data while the host wrote new tasks concurrently. The completion signal of the first packet arrived after $26.06\mu s$ at the host processor. For all 1000 packets a total of $13.73ms$ was needed. The measurements were also compared to the duration of strict sequential processing. For this purpose, the packet processor software was modified such that the next packet is not processed before the completion signal of the previous one had been sent. In this situation the system needed $20.10ms$ for the same 1000 kernel dispatches. Consequently 32% of execution time could be saved. This proves that the packet processor is capable of efficiently handling multiple tasks in parallel. This means the packet processor can dispatch up to 72833 tasks per second at 100 MHz clock speed. Since real workloads, e.g. for high quality image processing, usually need much more time to process, the presented hardware is well suited for its purpose, even at lower frequencies.

A special characteristic of this test scenario was that the execution time of the actual task was lower than the dispatch latency. This means all work was handled by a single accelerator core, even when several more were present in this system. To lessen this effect to a point where all cores are used, we increased the vector length to 256 times more data to copy. The basic task sequence as described in Fig. 6 remained the same. However, this time accelerator zero has not finished execution yet when the

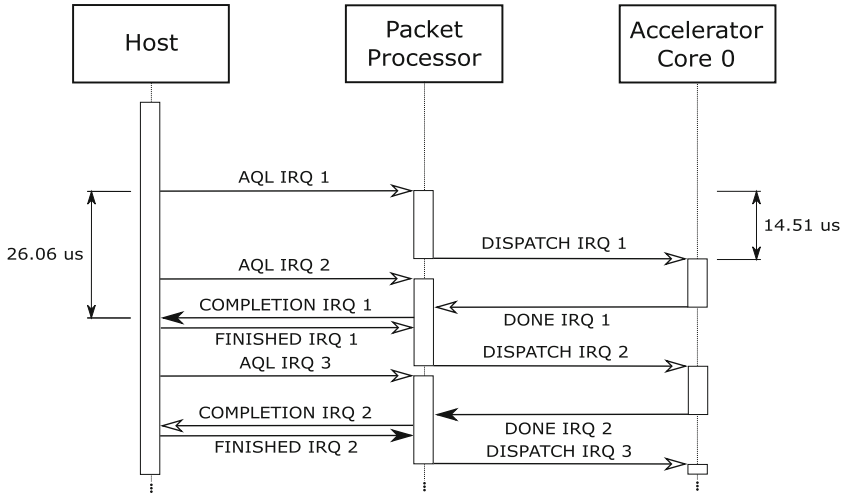

Figure 6 Basic sequence diagram of the first testcase. All AQL packets are scheduled and dispatched to the same core.

packet processor received a new AQL packet. Therefore it had to be dispatched by another core. The same held true for the following packets such that further dispatches had to be stalled until one of the accelerators concluded. In this scenario the host received the completion signal after 133.06μs. With 114.62μs 86.1% of this time was spent in the first accelerator core. The last packet finished after 22.14ms passed since the first one has been submitted to the queue. Therefore, this scenario took 61% longer than the first one. Considering the 256 times as much data to be transferred and the occurring bus contention this is only a minor increase. This still holds even when accounting for the additional accelerator cores used. On the one hand, this phenomenon can be explained by a better bus utilization, up to the point where contention problems arise. On the other hand, the packet processor has more work available at all times and therefore spends less time in idle states. This effectively hides the dispatching time and the setup latencies of the accelerators, making the whole system more efficient.

5.2.2 SVM Capabilites

In setups where the FPGA is a PCIe add-in card, the previous observations need to be adapted to account for the additional copy latencies. As explained in Section 3.2 the copying is fully automated by the FPGA cache controller unit, but acquire and release memory fences may need to be applied by setting the appropriate bits in the packet header. In this test cases the block size of FCC was set to 2 MB to match the size of x86_64 huge pages. Our Virtex UltraScale board provides a PCIe 3.0 x8 interface which can theoretically transfer up to 7.9 GB/s. This bandwidth is unaccounted for the protocol overhead and additional latencies and real rates are most probably lower. That means the following measurements are a worst case scenario for LibHSA. Since we only have a MIPS host processor, these transfer times of about 260μs for 2 MB were approximately simulated by this unit. To maximize the load on the LibHSA components a simple multiplication by two was used again as an "accelerator core". Nevertheless, this also accounts for setup times with realistic kernels (e.g. processing of arguments written to the unit by the packet processor). The amount of data was chosen to be transferable in a single AXI burst at 16 beats length in each direction.

First, a single kernel with an acquire and release fence was tested. This is a common use case for a wide range of kernels when only a single task is to be computed by a GPU. For now kernel arguments and data resided in the same 2 MB page of the host address space. This means only a single transfer was needed to fetch both, but naturally this is not a requirement. Figure 7 shows the corresponding sequence diagram. The packet processor applied the specified acquire fence 3.85μs after the packet has been submitted. Since there was no dirty data in the cache at the beginning, FCC only had to check if all valid signals were unset, which takes 1.50μs. Next, 11.72μs after the packet arrived, the packet processor started fetching the kernel arguments with a simple AXI transfer. This was needed as described in Section 3, because there is only a pointer to the kernel arguments embedded in an AQL packet. This access was done with an untranslated address, so that FCC had to resolve it and ultimately request the copy of the whole 2 MB block. 259.85μs later the transfer finished and all additional data access could be done locally. The packet processor then needed an additional 1.22μs to dispatch the kernel to the accelerator core. After the core had processed the data there were two further steps which needed to be done. First the release fence had to be applied. This happened after 5.04μs and flushed the whole 2 MB block, where the dirty data resided, back to host address space. Again, this took quite some time as shown before when loading the data. Finally the completion signal arrived 551.55μs after the

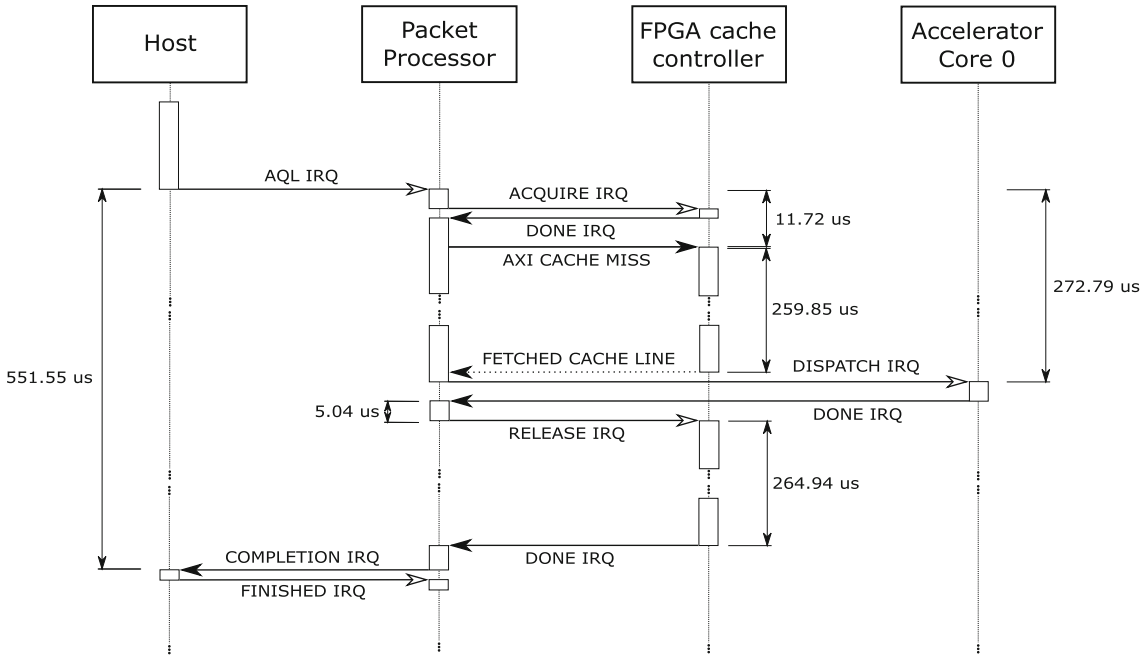

Figure 7 Sequence diagram of one kernel dispatch. Data and kernel arguments were not present in local memory and thus had to be copied by FCC.

AQL packet had been submitted. Therefore 95.1% of the time had been spent copying data between different memory regions.

Compared to the scenario where all data was already local on the board, a vast amount of additional time was needed. This can not be avoided, but handling these cases is automatically done by hardware. As with GPUs it is only reasonable to use this processing scheme if the benefit is great enough to justify the transfer costs. One way to reduce unnecessary flushes is the fine-grained use of memory fences. When multiple successive AQL packets operate on the same working set, fences with system scope are usually only needed in the first and last packet. Once the data is fetched by the FPGA cache controller the work is processed similar to the previous local test cases.

To demonstrate this capability we ran another example directly after the previous test case. This time 1000 packets were launched and each one multiplies the data in a newly malloced buffer by two. Thereby the host was not interested in intermediate results, so a release fence was only needed in the last packet. However, to make sure the newly allocated data buffer was visible to the accelerator the first one also needed an acquire fence. Thus in total two copy operations to and from the host were performed. To reduce bus contention we set a completion signal for every packet, so that we did not need to perform active waiting for an excessive amount of time on a variable in shared local memory. The sequence diagram is similar to Fig. 7, but this time the host wrote multiple AQL packets as visualized in Fig. 6. However, the packet processor dispatches all intermediate tasks without involving the FCC, because no memory fences were specified in the packet header. The only acquire fence was applied $3.24\mu s$ after the first packet had been submitted. This needed $1.62\mu s$ to invalidate all cache lines. The respective data transfer to fetch the new data was performed after $11.54\mu s$ while the first completion signal arrived at the host after $284.18\mu s$. Subsequent completion signals arrived every $12.84\mu s$. This was possible due to the absence of intermediate copies. Finally the only release fence was applied $13.11ms$ after the first packet had been submitted. The whole batch of work needed $13.38ms$ to process. In this usage pattern only 3.9% of the time was spent transferring data between the disjoint memory regions of host and agent. The amount of work to set up this behavior of the hardware is negligible because only two header entries per AQL packet need to be set. This proves that the FPGA cache controller eases the work of application developers by enabling the "pointer is a pointer" equivalence from the hardware side.

5.2.3 Image Processing Performance

The last scenario demonstrates the real-world use case of image processing explained in Section 4. Since images are usually stored in a contiguous block of memory we can also implement the SVM without the FPGA cache controller, but with a custom packet processor software as briefly discussed in Section 3.2. We measured a total number of

5805 clock cycles (58.1μs) for processing an AQL packet and handling all associated dispatch steps as described in Section 3.3. From that number 901 clock cycles (9.0μs) are spent in the initial phase after receiving the interrupt, stating that new AQL packets arrived. This includes reading and analyzing the packet, writing the DMA configuration and sending the first DMA interrupt. This cost is nearly constant for all kernel dispatches. Our accelerator needs at least $1920 \times 1080 = 2073600$ clock cycles to perform a filter operation on a full HD image, regardless of additional bus traffic. This is a common workload and more than 350 times slower than the dispatch and shows that processing an AQL packet is comparatively cheap. Even while managing multiple accelerators in parallel the additional latency introduced by DRAM bus contention is small, since the packet processor transfers only approximately 100 bytes per AQL packet over this bus. This is also negligible compared to real data set sizes and other high latencies. In particular data copy to the FPGA is generally very expensive due to bus limitations and also far slower than our dispatch cost.

We also observed on our demonstrator board that the DMA data copy cost needs the most time. As mentioned in Section 5.1 this is due to our non-optimal DMA unit replacement. However, if we again assume a PCIe 3.0 x8 interface, the theoretical maximum transfer rate unaccounted for the bus protocol overhead would be 7.9 GB/s. The authors in [1] proved that a bandwidth close to the optimum is attainable. To match this rate we can increase the AXI data bus width to 1024 bits for 100 MHz or 512 bits for 200 MHz. If we consider the 2.4 GB/s DDR4 data rate of our Virtex UltraScale board we can still choose a 128-bit AXI bus at 200 MHz. If we account for this full bus utilization in our measurements we need 31709.31μs (31.53 FPS) from packet arrival to completion signal for full HD images. This shows that LibHSA does not limit the attainable transfer rate and does not impact the performance more than a non-HSA solution, but eases the integration noticeably. The time spent copying data onto the FPGA board and back via DMA cannot be considered a cost to our library, since this has to be done in a real system regardless. Moreover, the custom packet processor can also cache the data in the FPGA DRAM so that subsequent kernels on the same data can operate directly on the DRAM data without further DMA access. Therefore, when our accelerator works on the same image multiple times the associated packets only need 20856.14μs (47.95 FPS) or 34% less time than the first operation. However, the time for the accelerator core to process the kernel function is, strictly speaking, also not a cost to our library and depends on the type and implementation of the accelerator core actually connected. All in all the throughput and latency penalty introduced by LibHSA is negligible for workload sizes like Full HD images which are commonly used nowadays.

6 Conclusion and Outlook

With LibHSA we provide a library with several components that enable developers to make their own hardware accelerator HSA compliant. Most components, like the packet processor or FPGA cache controller, can be reused for any custom accelerator, while we provide additional hardware for accelerators with the same processing scheme. By using AXI as a standard interface between components, interchanging devices can be managed easily. LibHSA was demonstrated by building a prototype with a weakly programmable image streaming core as the back end. We were able to synthesize our design on a *Xilinx Virtex Ultra Scale* FPGA. Moreover we showed that several packets can be processed at a time and executed on multiple accelerator cores in parallel. Duration of packet processing, data copying and data processing have been evaluated and compared to each other. With this approach we provide the first step to use FPGAs transparent in modern heterogeneous computing architectures.

Nevertheless, as for the complexity of such a system, various optimizations could speed up the entire processing. While the accelerator core is capable of high frame rates, data transfer on our on-chip prototype remains to be our system's bottleneck. However, this is not an inherent design flaw of LibHSA, but rather a problem of a missing high-performance DMA unit. In the future we will port LibHSA to an off-the-shelf PC system by adding such a PCIe DMA unit, which is compliant with the HSA standard and our interface. This unit could perform the copy operations between the main and FPGA on-board memory when requested by the FPGA cache controller on runtime. Combined with our work this would be a further step in establishing FPGAs in heterogeneous systems for accelerated data processing similar to GPUs nowadays.

References

1. de la Chevallerie, D., Korinth, J., Koch, A. (2016). Fflink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators. *SIGARCH Computer Architecture News*, *43*(4), 34–39.
2. Georgakoudis, G., Gillan, C., Hassan, A., Minhas, U.I., Spence, I.T.A., Tzenakis, G., Vandierendonck, H., Woods, R.F., Nikolopoulos, D.S., Shyamsundar, M., Barber, P., Russell, M., Bilas, A., Kaloutsakis, S., Giefers, H., Staar, P.W.J., Bekas, C., Horlock, N., Faloon, R., Pattison, C. (2016). Nanostreams Codesigned microservers for edge analytics in real time. SAMOS, pp. 180–187.
3. Glossner, J., Blinzer, P., Takala, J. (2015). HSA-Enabled DSPs and accelerators. In *IEEE Global Conference on Signal and Information Processing (GlobalSIP)* (pp. 1407–1411).
4. Guidi, G., Reggiani, E., Tucci, L.D., Durelli, G., Blott, M., Santambrogio, M. (2016). On How to Improve FPGA-Based Systems Design Productivity via SDAccel. In *2016 IEEE*

International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 247–252).
5. Hennessy, J.L., & Patterson, D.A. (2011). *Computer architecture fifth edition: a quantitative approach*, 5th ed. San Francisco: Morgan Kaufmann Publishers Inc.
6. HSA Foundation (2016). HSA Foundation Specification Version 1.1.
7. HSA Foundation (2016). HSA Platform System Architecture Specification 1.1.
8. HSA Foundation (2016). HSA Programmer Reference Manual Specification 1.1.
9. HSA Foundation (2016). HSA Runtime Specification 1.1.1.
10. Jaaskelainen, P., de La Lama, C.S., Huerta, P., Takala, J.H. (2010). OpenCL-based Design Methodology for Application-Specific Processors. In *International Conference on Embedded Computer Systems* (pp. 223–230): IEEE.
11. Janik, I., Tang, Q., Khalid, M. (2015). An overview of Altera SDK for openCL A user perspective. In *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)* (pp. 559–564).
12. Heinrich, J. (1994). MIPS R4000 Microprocessor User's Manual.
13. Kim, N.S., Chen, D., Xiong, J., Wen-mei, W.H. (2017). Heterogeneous computing meets Near-Memory acceleration and High-Level synthesis in the Post-Moore era. *IEEE Micro*, *37*(4), 10–18.
14. Mukherjee, S., Sun, Y., Blinzer, P., Ziabari, A.K., Kaeli, D. (2016). A comprehensive performance analysis of HSA and openCL 2.0. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (pp. 183–193).
15. Radeon Technologies Group (2017). Radeon's next-generation Vega architecture. Tech. rep., Advanced Micro Devices (AMD).
16. Schmidt, M., Reichenbach, M., Fey, D. (2012). A Generic VHDL Template for 2D Stencil Code Applications on FPGAs. In *15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops* (pp. 180–187).
17. Segal, O., Nasiri, N., Margala, M., Vanderbauwhede, W. (2014). High level programming of FPGAs for HPC and data centric applications. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1–3): IEEE.
18. Severance, A., & Lemieux, G.G.F. Embedded Supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. CODES+ISSS '13, IEEE Press, pp. 6:1–6:10.
19. Su, L.T. (2013). Architecting the future through heterogeneous computing. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (pp. 8–11).
20. Wu, Q., Ha, Y., Kumar, A., Luo, S., Li, A., Mohamed, S. A heterogeneous platform with GPU and FPGA for power efficient high performance computing. In *2014 14th International Symposium on Integrated Circuits (ISIC) (2014)* (pp. 220–223): IEEE.

Marc Reichenbach received his Diploma degree in Computer Science at Friedrich-Schiller University Jena in 2010 and his PhD in 2017 at Friedrich Alexander University Erlangen-Nürnberg (FAU). He now works as a postdoctoral researcher at FAU at the chair of Computer Architecture. His research interests are embedded systems, especially smart sensor architectures for varying application fields.

Philipp Holzinger received his Master's degree in Computer Science at the Friedrich Alexander University Erlangen-Nürnberg (FAU) in 2017. Currently he is a research fellow at FAU at the chair of Computer Architecture. His research interests are heterogeneous system architecture, especially with a focus on HSA Foundation standards, FPGA accelerator design, high-level synthesis and deep learning.

Konrad Häublein studied at the University of Applied Science in Jena. He holds a Diploma degree in Mechatronics and a Master's degree in Electrical Engineering. In 2012 he started as a Ph.D. student at the chair of Computer Architecture at Friedrich Alexander University Erlangen-Nürnberg (FAU). His research interests are smart cameras, generic hardware architectures and embedded hardware accelerators.

Tobias Lieske received his Master's degree in Information and Communication Technology at Friedrich Alexander University (FAU) Erlangen- Nürnberg in 2015. Currently he is a research fellow at FAU at the chair of Computer Architecture. There he focuses on low-power system design, architecture generation and accelerator design.

Paul Blinzer works on a wide variety of Platform System Software architecture projects at Advanced Micro Devices, Inc. (AMD) as a Fellow for heterogeneous system software in the Radeon Technology Group. Living in the Seattle, WA area, during his career he has worked in various roles on system level driver development, system software development, graphics architecture, graphics & compute acceleration since the early '90s. Paul is the chairperson of the "System Architecture Workgroup" of the HSA Foundation. He has a degree in Electrical Engineering (Dipl.-Ing) from TU Braunschweig, Germany. https://www.linkedin.com/in/paul-blinzer-4523602

Dietmar Fey holds a Diploma degree in Computer Science from Friedrich Alexander University (FAU) Erlangen-Nürnberg, Germany. In 1992 he received a Ph.D. from FAU with his work on "Using Optics in Computer Architectures". From 1994 to 1999 he researched at Friedrich-Schiller-University Jena where he completed his "Habilitation". From 1999 to 2001 he worked as a lecturer at Siegen University before he became a professor of Computer Engineering at Jena University. Since 2009 he has led the Chair for Computer Architecture at Friedrich Alexander University (FAU) Erlangen-Nürnberg. His research interests are parallel computer architectures, programming environments, embedded systems, and memristive computing.