



Programmable HSA Accelerators for Zynq UltraScale+ MPSoC Systems

Wolfgang Bauer¹✉, Philipp Holzinger¹, Marc Reichenbach¹, Steffen Vaas¹,
Paul Hartke², and Dietmar Fey¹

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

{WolfgangM.Bauer, Philipp.Holzinger, Marc.Reichenbach,
Steffen.Vaas, Dietmar.Fey}@fau.de

² Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124-3400, USA
phartke@xilinx.com

Abstract. Modern algorithms for virtual reality, machine learning or big data find its way into more and more application fields and result in stricter power per watt requirements. This challenges traditional homogeneous computing concepts and drives the development of new, heterogeneous architectures. One idea to attain a balance of high data throughput and flexibility are GPU-like soft-core processors combined with general purpose CPUs as hosts. However, the approaches proposed in recent years are still not sufficient regarding their integration in a shared hardware environment and unified software stack. The approach of the *HSA Foundation* provides a complete communication definition for heterogeneous systems but lacks FPGA accelerator support. Our work presents a methodology making soft-core processors HSA compliant within MPSoC systems. This enables high level software programming and therefore eases the accessibility of soft-core FPGA accelerators. Furthermore, the integration effort is kept low by fully utilizing the *HSA Foundation* standards and toolchains.

Keywords: Heterogeneous system architecture · FPGA
Programmable accelerator · HSA foundation · Zynq ultrascale+
Nyuzi processor

1 Introduction

Modern computing applications keep growing requirements in terms of execution time and power consumption. This development can be observed for high-performance computing, desktop environments, as well as in embedded systems. However, the upcoming end of Moore's law limits the prospects of traditional CPU centered computing. In the future these requirements can only be satisfied by increasingly heterogeneous systems. Such environments exploit the benefits of

CPUs, GPUs, DSPs and FPGAs by executing each task on the best suited. This way heterogeneous systems can be designed combining different architectures in order to attain the highest energy efficiency. For embedded systems this concept can be further extended by integrating all cores on a single die.

Contrary to all benefits of heterogeneous architectures, software development is getting more and more complex with the rising amount of different parts. Therefore, various programming models and language extensions, like CUDA, OpenCL or OpenMP, have been introduced to reduce the programming complexity by abstracting architecture specific properties. Some of them also hide data communication between different hardware units. However, most of the existing standards are either proprietary or lack exact definitions regarding communication from a hardware point of view. To close this gap the Heterogeneous System Architecture Foundation (HSA Foundation¹) specified a low-level programming model and system software infrastructure to support heterogeneous computing architectures [10]. This facilitates the extension of new HSA compliant acceleration devices to existing systems without any changes to the application source code.

HSA Foundation standards are already established in the desktop computers from AMD and their graphics cards [3, 12]. Recent embedded devices, such as smartphones, also incorporate HSA compliant chips [14]. Unfortunately, FPGAs which are a good choice for heterogeneous systems due to their high peak performance and low power consumption, are currently not fully supported. Due to their highly flexible nature, finding a mapping is significantly more complex and needs to be further investigated. This means up to now, new, emerging SoC architectures e.g. from Xilinx or Altera, which contain processor cores and an FPGA part, could not benefit from HSA Foundation standards. This limits the flexibility of these powerful embedded devices. Therefore, in this paper we show a new methodology to make SoCs, containing a CPU and FPGA part, HSA compliant. Due to the interface's open definition between software runtime and hardware the HSA environment is ideally suitable handling the FPGA's communication to other components in heterogeneous systems.

To demonstrate our concept, we chose a Xilinx SoC containing an ARM application processor deployed as controlling host unit and a FPGA part as hardware accelerator. Traditionally, such FPGA based hardware accelerators are described with custom HDL code, which makes the accessibility of FPGA accelerators rather low. Therefore, to preserve the flexibility and to avoid language restrictions, we use in this paper highly configurable and customizable soft-core accelerators, to unite flexibility and pragmatism of FPGAs. Those cores enable the execution of multiple application tasks without the need of resynthesis and reloading, but can also be adapted for the application, e.g. by adding custom instruction units.

This paper is structured as follows: First, competing concepts and similar approaches are evaluated in Sect. 2. Afterwards in Sect. 3 the used hardware platform, the necessary fundamentals of the HSA Foundation standards and

¹ <http://www.hsafoundation.com>.

the architecture of the selected GPU like soft-core are explained. Then Sect. 4 describes the derived hardware setup and software toolchain. The detailed results are presented in Sect. 5. Lastly Sect. 6 summarizes the paper and briefly covers possibilities for future work.

2 Related Work

The most promising alternative to programmable soft-core GPUs for using an FPGA from a common language is high-level synthesis (HLS). However, instead of compiling to an accelerator's instruction set architecture (ISA), the functionality of the kernel itself is mapped to an application-specific hardware circuit. While in its simplest form only the kernel itself is translated to HDL [17], there exist more sophisticated solutions generating the hardware connection as well as software interfacing automatically. The most prominent ones are the commercial Xilinx SDAccel [16] and the Intel FPGA SDK for OpenCL [2]. Both are based on the vendor neutral and well known OpenCL standard to ease the usage for developers. An academic approach to HLS was for example done by LegUp [8]. It provides FPGA accessibility via *pthread*s or OpenMP [9], but also tries to get a step further by automatically determining and offloading frequently used code sections. While this process usually leads to better results than code execution on a soft-core processor, it has major drawbacks. Since each kernel can only execute a specific program, the time-consuming synthesis and FPGA reconfiguration has to be done anew for each kernel in the application. Therefore, a more flexible approach based on soft-core accelerators is advantageous.

Further approaches rely on application-specific instruction set processors (ASIP). The ISA of those cores is optimized for a certain application-field. For example in [15] an ASIP for power quality monitoring was developed. In comparison to hard-wired solutions ASIPs require a slight resource overhead and provide slightly less performance, but offer programming flexibility without resynthesis as big benefit. This strategy works well for processing intense tasks, but in times of Internet-of-Things even more flexibility is necessary to realize communication protocols like for example OPC UA or TCP/IP. Thus, a combined system-on-a-chip architecture consisting of multiple general-purpose embedded CPU- and accelerator cores (MPSoC) is required, to obtain a low-power solution at sufficient performance and flexibility. Nevertheless, the performance enhancement of ASIPs can only be exploited when using their rather complex instructions, like FFT- or mean value calculations. Current compilers are incapable to map standard source code to those specific instructions, so developers have to use architecture-specific functions to benefit from such architectures. That practice cums the flexible source-code portability from and to other architectures.

While many open source processors like LEON3², OpenRISC³, Amber ARM-compatible core⁴ and various RISC-V⁵ implementations are available for scalar

² <http://www.gaisler.com>.

³ <https://openrisc.io>.

⁴ <https://opencores.org/project/amber>.

⁵ <https://riscv.org>.

data processing, the variety for soft-core GPU architectures is more reduced. Al-Dujaili et al. extended the mentioned LEON3 processor by adding parallelization and synchronization features to support the CUDA execution model with their Guppy GPU-like soft-core processor [1]. An other approach is pursued by the FlexGrip project [4] and the MIAOW project [5]. The developed architectures are based on proprietary Nvidia or AMD GPUs and modeled according to available information. Due to ISA compatibility the existing vendor toolchains can be used for code generation.

Al Kadi et al. proposed the FGPU [11] with a MIPS-based ISA, which is extended by further vector-processing instructions to execute OpenCL kernels. It provides hardware support for scheduling work items to multiple computing units conveniently to the SIMT programming model and includes an LLVM-backend. In contrast to this the Nyami [6], or later Nyuzi [7], presented by Jeff Bush et al. uses a more general purpose architecture utilizing wide vector registers with predicated execution vicarious to the Intel Xeon Phi architecture. It also uses its own ISA similar to MIPS-ISA and, besides integer arithmetic, floating point operations are supported as well. Due to the LLVM-backend many languages providing LLVM-frontends can be translated to the processor's ISA.

The selection of available source languages is currently quite limited for all these approaches and there is little freedom of choice. A common low-level standard like the HSA specifications can reduce the time to develop language frontends and diversify the existing solutions.

3 Fundamentals

3.1 MPSoC Platform

The Xilinx Zynq UltraScale+ MPSoC integrates a quad-core ARM Cortex-A53 MPCore based processing system (PS) and Xilinx programmable logic (PL) in a single device [18]. The 16 nm FinFET+ PL communicates with the PS through 6,000 interconnects that are organized into twelve 128-bit high-performance ARM AMBA AXI4 ports each providing different capabilities. The high-performance AXI4 ports provide access from the PL to DDR and high-speed interconnect in the PS. The PL can be tightly or loosely coupled to the A53 APUs via two-way coherent, I/O (one-way) coherent, or non-coherent transactions. Address translation is provided by the system memory management unit (SMMU) on select AXI4 interfaces.

3.2 HSA Specifications

In this paper we leverage the HSA Foundation standards [10] and its existing ecosystem to improve the integrability of heterogeneous SoCs. This provides a new level of flexibility for developers of embedded systems. The specifications consist of three main parts:

- The *Programmer's Reference Manual* defines the HSA intermediate language (HSAIL) which abstracts the target ISA.

- The *Runtime Programmer’s Reference Manual* defines the vendor neutral hardware communication API a language runtime is expected to target.
- The *Platform System Architecture Specification* defines the underlying hardware model which the software toolchain is targeting.

To utilize all features the HSA ecosystem provides an application developer does not need to use any HSA specific constructs. Instead the programming language can freely be chosen among the available HSA compiler frontends. When the application software is compiled the compiler is expected to separate kernel from host code and generate all HSA runtime API calls needed. To provide an enhanced flexibility with regards to the actual accelerator hardware a special virtual language is used as an intermediate representation of accelerator kernel code.

This language is called HSAIL and has a textual form which resembles Nvidia’s PTX. Its binary representation is BRIG. For all purposes of this paper HSAIL and BRIG are equivalent and can be converted into each other. Similar to CUDA and OpenCL, a separation of kernels into work-groups and work-items is also used in this execution model. A common workflow is splitted into the two steps. First, the source code of a supported language is compiled to HSAIL. Afterwards, either at compile time or runtime, this intermediate code is *finalized* to the accelerator ISA. Finalization is very lightweight, because most time consuming steps like register allocation already happened in the previous compile process that produces low-level BRIG. This is possible due to the minimum hardware requirements defined in the Platform System Architecture Specification. With this concept the dispatch latency can be reduced compared to direct compilation from LLVM-IR/SPIR.

A reference to these kernels in target ISA is embedded in an AQL kernel dispatch packet. These packets have a special format and compliant hardware is expected to be able to interpret it. Besides the actual binary all associated meta information like the grid size are also included. To submit a job to an accelerator core such a packet just needs to be written to a user-mode queue provided by the device. All further processing is then in the responsibility of the hardware.

3.3 LibHSA Library

In order to logically and physically connect a programmable accelerator in the PL to the ARM cores in the PS a connector is needed. In particular these components must adhere to the protocol specified by the HSA Foundation. With LibHSA the first implementation of such a system was presented by Reichenbach et al. [13]. Its core component is a self-developed, lightweight, 64 bit CPU based on the MIPS III ISA. It acts as an AQL *packet processor* and manages all incoming tasks dispatched via the HSA runtime. After interpreting the AQL packet, the packet processor issues the execution command described in Fig. 1 to a suitable accelerator core. As bus protocol the widespread AXI4 standard has been incorporated to decrease the needed integration effort for new cores.

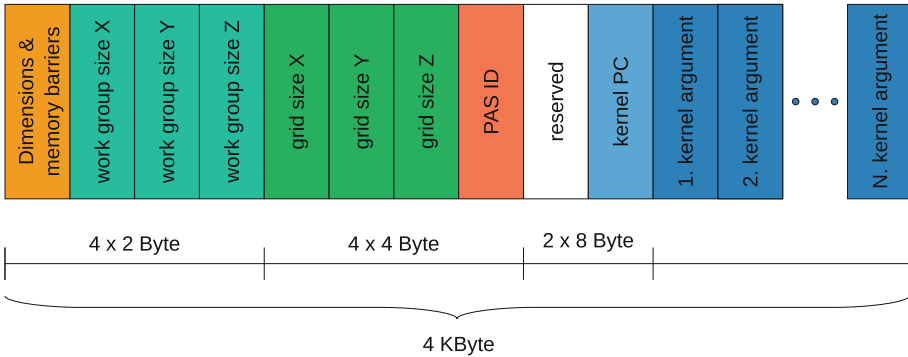


Fig. 1. Execution command format of the packet processor to the accelerator cores.

Additionally a fixed interface of an accelerator has been defined such that cores can be freely interchanged.

In the following LibHSA is used to connect the accelerator core to the ARM host in an HSA compatible way. However, in the original paper they only demonstrated a MIPS CPU in the PL as a host. An implementation for a faster and highly energy-efficient ARM CPU has not been presented yet. Therefore, some adaptations had to be made in this paper.

3.4 Nyuzi Vector Processor

The accelerator core used in this paper is the open source, 32 bit Nyuzi vector processor⁶. It was designed by Jeff Bush for highly parallel applications [7]. The simultaneous multi threading (SMT) capable, multi-core architecture incorporates floating point and integer SIMD execution units. Its memory subsystem consists of coherent, set associative L1 and L2 caches. All vector instructions support predication allowing individual lanes of the vector to be masked to avoid branches for diverging program flow paths. The data communication is established via two separate bus systems. On the one hand the IO-bus is used for uncached peripheral small data transfers without any protocol overhead besides access arbitration. On the other hand memory transactions with the remaining system are carried out by an AXI4-full interface with 32 bit addresses and adjustable data width.

The Nyuzi processor is parameterized and can be easily modified to contain the desired number of cores. Moreover, the cache size, number of vector lanes, and threads per hardware core can be also configured. To program the parallel processor architecture of Nyuzi the project includes a complete LLVM compiler toolchain which utilizes all of the hardware features.

⁶ <http://nyuzi.org/>.

4 Environment

4.1 Hardware Structure

The general hardware setup can be seen in Fig. 2. Compared to the setup Reichenbach et al. proposed in [13] the accelerator cores are no longer limited to fixed function accelerators. Now programmable soft-core architectures are also possible. Moreover, with our extension their custom MIPS host processor can be replaced by a high-performance ARM ASIC. Our design is the same for all Zynq-based designs from low-cost *UltraZed* and *Ultra96* boards to the high-end *Sidewinder-100*. On the host side the *ARM Cortex A-53* is used for the reasons explained in Sect. 3.1. It dispatches all tasks which need to be accelerated to the *Nyuzi vector processor*. The connection between these main components is realized with the LibHSA library described in Sect. 3.3. It acts as middleware between host and accelerator to provide a uniform HSA-based interface and communication protocol. The central component is the *packet processor* which manages all submitted kernels. Since the task dispatch always follows the HSA specifications there is no difference between the instruction sets of different host processors. This means on the software side any programming language with an HSA backend is supported including upcoming, future backends. However, the physical connection on the hardware level differs making adaptations necessary.

First, the way to send and receive interrupts differs between the Zynq ARM core and MIPS host processor. While Reichenbach et al.'s custom MIPS has dedicated pins for all needed in- and outgoing interrupts, this had to be changed to GPIO for the ARM processor.

Secondly, in the Zynq system there is only one main DRAM memory region where data is shared and this is accessible with the same addresses from PS and PL. This means in contrast to a x86/PCIe setup the shared virtual memory requirement is trivially satisfied in the Zynq MPSoC system.

Lastly, in an MPSoC system the DRAM is accessed via the Zynq IP core. Here, cache coherency can be established with the integrated CCI-400. This means no additional hardware units or software changes to the packet processor are needed to fulfill this requirement. However, not all accelerators (like Nyuzi) support cache snooping via ACE, such that this property can be relaxed if the specific application doesn't need it. For that reason, to have both, high-throughput DMA and (one-way) cache coherency, the AXI HPC ports were used to access the DRAM from PL.

On the accelerator side LibHSA uses its own protocol since no requirements are stated in the HSA specification. As explained in [13] accelerator cores can be usually easily adapted with connector components provided by LibHSA. However, the Nyuzi vector processor additionally lacks abilities like system wide memory barriers to make it suitable for shared, heterogeneous processing with other hardware components. Therefore, more adaptations need to be made which are explained in the following Sect. 4.2.

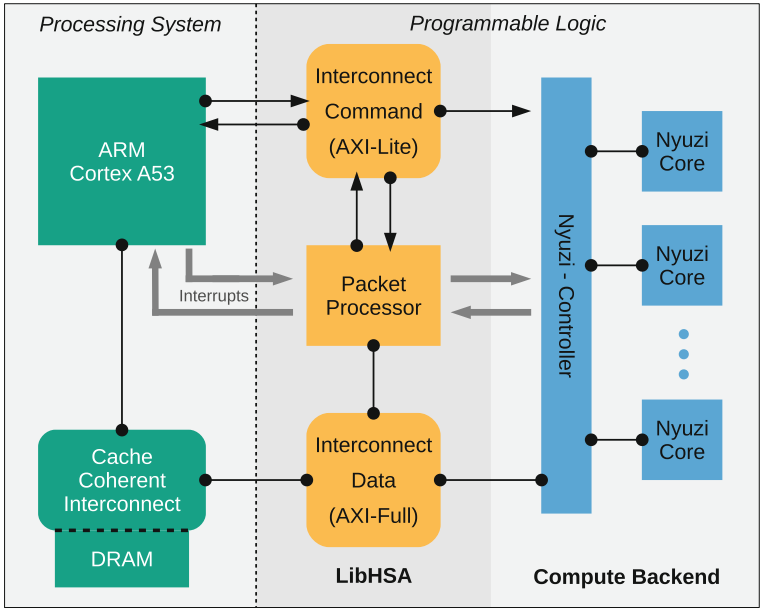


Fig. 2. Overview of the system components. The design is splitted into three main parts. On the right hand side there is the host subsystem with ARM, DRAM and cache-coherent interconnect. These parts are exclusively located in the PS. The accelerator cores can be seen on the left hand side. They are part of the programmable logic and are interchangeable at runtime. Both parts are connected via the LibHSA library in the middle section. It ensures the HSA conformance of the accelerator cores.

4.2 Nyuzi Adapter

The Nyuzi architecture is designed to run as host processor in a computing system, not as co-processor, offering two data bus systems and an interrupt interface to communicate with its environment. In order to minimize the management overhead inside the compute units a dedicated hardware component, the Nyuzi-controller, is added. This can accept AQL packets via a separate AXI-LITE interface and can schedule work items directly on specific Nyuzi hardware threads assigned to one core. After a thread is completed, it disables itself monitored by the Nyuzi controller to schedule the next work item on the now available thread.

Work item related information, e.g. item id or group dimensions, is also retained for each thread in the Nyuzi controller. To avoid cache invalidations and unwanted replacements the work item meta information is transferred over the separate IO-bus allowing simple 32 bit read and write transactions. In the original Nyuzi design each write or read will cause a pipeline rollback because the IO-bus was intended to perform for slow peripheral memory accesses. To speedup IO-bus requests the execution was pipelined into four stages according

to a best case access scenario. Therefore, only bus conflicts will lead to pipeline rollbacks.

The cache system is coherent over all cores but is not capable of communicating to external components. Therefore, the L2 cache was extended with global flushing and invalidation functionality enabling HSA memory fences. The added flush controller waits till all L1 caches have finished their write requests and flushes all dirty cache lines afterwards. For cache invalidation all valid bits in L2 and L1 caches are cleared after flushing the L2 cache.

4.3 HSAIL/BRIG Finalization

The executed kernel functions are compiled from any supported high level language into the HSA intermediate representation, BRIG. Due to developers providing an LLVM-backend for Nyuzi code generation the BRIG program is first transformed into the LLVM intermediate representation (LLVM-IR). Because of the lacking tooling support on the side of the HSA Foundation this step is accomplished using a self written tool supporting all instructions required for the example kernels. For better utilization of the vector register and arithmetic the used kernels are automatically vectorized, based on kernels' LLVM-IR for better reusability. The basic concept is to compute 16 work item in concurrent X dimension with one kernel call. Therefore, within the kernel the API-call returning the X-id is searched and replaced with a vector of ascending, adjacent ids. Instructions using this now vectorized id have to expand each other operand by either expanding the scalar value or vectorization. Using vectorized operands transforms the instruction result into a vectorized value, which is recursively for all instructions. Control flow divergence evoked by vectorized branch conditions lead to the reorganization of the control flow graph with the addition of executions masks for predicated instruction execution.

5 Results

The presented results are based on the hardware boards described in Sect. 4.1. The Nyuzi accelerator is synthesized with four different configurations regarding the amount of cores and cache sizes, two for the UltraZed board and two for the Sidewinder-100 board. Due to the selection of benchmark programs no floating point unit is configured in all variants. Table 1 shows the chosen configurations combined with the required hardware resource using Vivado version 2017.2 as synthesis toolchain. The complete FPGA resources are split into the static part, the LibHSA environment, and the configurable part, the accelerator cores.

On the software side the host program is running bare metal on one ARM core and starts various benchmark kernels using a reduced HSA-runtime. The selected pure integer programs are:

- *Vec Add*: The simple addition of two vector with 2^{15} values.
- *Mat Mul*: Multiplication of a 2048×100 and a 100×100 matrix.

Table 1. FPGA resource utilization for LibHSA environment and Nyuzi-core configurations on the UltraZed and Sidewinder MPSoC platforms

Platform	Component	LUTs [k]	FFs [k]	BRAM	DSPs
UltraZed	LibHSA Environment	8.3 (11.8%)	7.1 (5%)	8 (4%)	0 (0%)
	1-Core, Cache-Sizes: 32 KB-L1 256 KB-L2	40.9 (58%)	24.5 (17%)	96.5 (45%)	160 (44%)
	2-Cores, Cache-Sizes: 4 KB-L1 16 KB-L2	57.4 (81%)	33.6 (24%)	78.5 (36%)	128 (36%)
Sidewinder-100	LibHSA Environment	8.3 (1.6%)	7.0 (0.7%)	8 (0.8%)	0 (0%)
	4-Cores, Cache-Sizes: 16 KB-L1 128 KB-L2	117 (22%)	65 (6.2%)	171 (17%)	256 (13%)
	8-Cores, Cores-Sizes: 32 KB-L1 256 KB-L2	229 (44%)	124 (12%)	345 (35%)	512 (26%)

Table 2. Runtime comparison of different benchmark program kernels running on different Nyuzi configurations and the ARM Cortex A53

	Vec Add	Mat Mul	Gauss 3×3	Gauss 5×5	Diff of Gauss
1 Nyuzi-Cores @150 MHz	10.6 ms	15.3 s	164.4 ms	336.2 ms	588.4 ms
2 Nyuzi-Cores @100 MHz	4.83 ms	129.2 s	86.0 ms	186.7 ms	377.9 ms
4 Nyuzi-Cores @150 MHz	1.87 ms	312.5 ms	15.0 ms	28.3 ms	57.8 ms
8 Nyuzi-Cores @100 MHz	1.90 ms	90.0 ms	10.4 ms	14.2 ms	38.8 ms

- *Gauss 3×3* : Application of a 3×3 convolution filter to a 512×512 image.
- *Gauss 5×5* : Application of a 5×5 convolution filter to a 512×512 image.
- *Diff of Gauss*: This programs calculates the absolute value of the difference of *Gauss 5×5* and *Gauss 3×3* .

The resulting execution times for one kernel call can be seen in Table 2. Furthermore, it demonstrates the achieved PL-frequencies. The time measurements are accomplished using the ARM’s real time clock, and are averaged over 100 runs. Attention should be paid to the difference in memory access times of both hardware platforms with the Sidewinder-100 board performing around four times faster than the UltraZed board. The kernel execution includes the transfer from the ARM’s cache into the Nyuzi’s cache, the actual kernel execution, and the flushing backing into the cache system of the ARM. Therefore, primarily memory bound kernels, like *Vec Add*, hardly scale with the rising amount of computing cores. This scaling can be distinguished clearly for the three convolution filter kernels. The unproportional trend of execution time for the *Mat Mul* kernel can be explained with the variation of the cache size matching the problem size superiorly.

6 Conclusion

In this paper we presented a methodology to utilize highly configurable and programmable soft-core accelerators by making MPSoC systems HSA compliant. The high flexibility regarding the front end programing language and standardized communication interface substantially could improve the accessibility these accelerators.

We could demonstrate a HSA based heterogeneous system connecting an ARM host CPU to the existing GPU like Nyuzi processor using software and hardware components of LibHSA [13]. The necessary extensions to the Nyuzi core on hardware level and the LibHSA environment were described. On the software side a finalizer provides the conversion from intermediate BRIG code to Nyuzi ISA.

All in all we could show that the HSA Foundation standards can reduce the overall complexity of heterogeneous platforms, like the Zynq UltraScale+. Moreover, the Zynq capabilities itself are well suited to implement an HSA-based system on top of it. Furthermore, the overhead in the system introduced by the HSA standard is only negligible. The HSA runtime API allows easy dispatching of tasks to the accelerator cores with no knowledge of hardware specifics required.

The host code is currently running bare metal on an ARM core. In the future an adaptation for a full Linux operating system can make it even simpler to deploy an easily usable, heterogeneous, multi-user system on Zynq basis. In addition, other accelerator cores could replace the currently used Nyuzi processor. Furthermore, it is conceivable to integrate LibHSA's packet processor as a separate ASIC in the MPSoC system and use its capabilities to bring the HSA functionality to embedded devices without needing additional logic resources.

Acknowledgments. We want to thank Xilinx and Fidus Systems for providing the used Zynq hardware platforms necessary to conduct our research.

References

1. Al-Dujaili, A., Deragisch, F., Hagiescu, A., Wong, W.: Guppy: a GPU-like soft-core processor. In: 2012 International Conference on Field-Programmable Technology, FPT 2012, Seoul, Korea (South), 10–12 December 2012, pp. 57–60. IEEE (2012)
2. Altera: Implementing FPGA Design with the OpenCL Standard, November 2013. https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf
3. AMD: ROCm: Open Platform For Development, Discovery and Education around GPU Computing, April 2016. <https://gpuopen.com/compute-product/rocm/>
4. Andryc, K., Merchant, M., Tessier, R.: Flexgrip: A soft GPGPU for FPGAS. In: 2013 International Conference on Field-Programmable Technology (FPT), pp. 230–237, December 2013
5. Balasubramanian, R., et al.: MIAOW - an open source RTL implementation of a GPGPU. In: 2015 IEEE Symposium in Low-Power and High-Speed Chips, COOL CHIPS XVIII, Yokohama, Japan, 13–15 April 2015, pp. 1–3. IEEE (2015)
6. Bush, J., Dexter, P., Miller, T.N., Carpenter, A.: Nyami: a synthesizable GPU architectural model for general-purpose and graphics-specific workloads. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, 29–31 March 2015, pp. 173–182. IEEE Computer Society (2015)
7. Bush, J., Khasawneh, M.A., Mahmoud, K.Z., Miller, T.N.: NyuziRaster: Optimizing rasterizer performance and energy in the Nyuzi open source GPU. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 204–213. IEEE (2016)

8. Canis, A., et al.: LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2011, ACM, New York, NY, USA, pp. 33–36 (2011)
9. Choi, J., Brown, S., Anderson, J.: From software threads to parallel hardware in high-level synthesis for FPGAs. In: 2013 International Conference on Field-Programmable Technology (FPT), pp. 270–277. IEEE (2013)
10. HSA Foundation: HSA Foundation Specification Version 1.1, May 2016. <http://www.hsafoundation.com/standards/>
11. Kadi, M.A., Huebner, M.: Integer computations with soft GPGPU on FPGAs. In: 2016 International Conference on Field-Programmable Technology (FPT), pp. 28–35, December 2016
12. Mukherjee, S., Sun, Y., Blinzer, P., Ziabari, A.K., Kaeli, D.: A comprehensive performance analysis of HSA and OpenCL 2.0. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 183–193. IEEE (2016)
13. Reichenbach, M., Holzinger, P., Häublein, K., Lieske, T., Blinzer, P., Fey, D.: LibHSA: one step towards mastering the era of heterogeneous hardware accelerators using FPGAs. In: 2017 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1–6. IEEE (2017)
14. Samsung: A Mobile Processor That Goes Beyond Mobile Innovation, April 2016. <http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-8895/>
15. Vaas, S., Reichenbach, M., Fey, D.: An application-specific instruction set processor for power quality monitoring. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 181–188, May 2016
16. Xilinx: The Xilinx SDAccel Development Environment (2014). https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-background.pdf
17. Xilinx: Vivado Design Suite User Guide: High-Level Synthesis, October 2014. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf
18. Xilinx: Xilinx Zynq UltraScale+ Device Technical Reference Manual, December 2017. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf