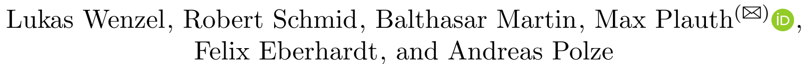
Getting Started with CAPI SNAP: Hardware Development for Software Engineers



Lukas Wenzel, Robert Schmid, Balthasar Martin, Max Plauth(✉),
Felix Eberhardt, and Andreas Polze

Operating Systems and Middleware Group, Hasso Plattner Institute
for Digital Engineering, University of Potsdam, Potsdam, Germany
{lukas.wenzel,robert.schmid,max.plauth,felix.eberhardt,
andreas.polze}@hpi.uni-potsdam.de,
balthasar.martin@student.hpi.uni-potsdam.de

Abstract. To alleviate development of FPGA-based accelerator function units for software engineers, the OpenPOWER Accelerator Work Group has recently introduced the CAPI Storage, Network, and Analytics Programming (SNAP) framework. However, we found that software engineers are still overwhelmed with many aspects of the novel hardware development framework. This paper provides background and instructions for mastering the first steps of hardware development using the CAPI SNAP framework. The insights reported in this paper are based on the experiences of software engineering students with little to no prior knowledge about hardware development.

Keywords: FPGA · Programming environment · Tutorial

1 Introduction

Embracing heterogeneous computing, hardware vendors are seeking new approaches for augmenting general purpose *Central Processing Units* (CPUs) with accelerator hardware to satisfy the ever-growing demand for compute capacity. *Field-Programmable Gate Arrays* (FPGAs) can be used in many application scenarios while being orders of magnitude more power-efficient compared to *Graphics Processing Units* (GPUs) [1]. With the Zynq SoCs, Xilinx has successfully demonstrated the consolidation of FPGA-based programmable logic with ARM-based CPU cores [17]. Following this trend of tightly coupling programmable logic accelerators with CPUs, IBM has introduced the *Coherent Accelerator Processor Interface* (CAPI) [12], making hardware accelerators such as FPGAs first-class citizens by integrating them into the processors coherent memory hierarchy.

Unfortunately, the benefits of FPGAs come at the cost of high development efforts, as it is very time consuming and difficult for software engineers to implement FPGA-based *Accelerator Functional Units* (AFUs). To optimize hardware

© Springer Nature Switzerland AG 2019
G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 187–198, 2019.
https://doi.org/10.1007/978-3-030-10549-5_15

designs, detailed knowledge about the targeted FPGA is required. Furthermore, additional effort is necessary to establish communication channels with the host application, as well as for interfacing with peripheral hardware available on the FPGA extension card. To alleviate these issues, the *OpenPOWER Accelerator Work Group* has recently introduced the *CAPI Storage, Network, and Analytics Programming* (SNAP) framework. On the side of the host application, it enables simple integration of AFUs by providing a ready-to-use job infrastructure. Complementing the hardware side, the framework provides libraries for accessing hardware components such as DRAM, NVMe flash storage and network interfaces. Covering both the software and hardware side of FPGA development, as well as the ensuing build process, the CAPI SNAP framework enables developers to focus their efforts on implementing their AFUs using Vivado HLS C/C++.

However, even with all these support mechanisms in place, we found that even graduate students in software engineering are still overwhelmed by many aspects of the novel framework, including the initial setup of the development environment, simulation of AFUs, as well as deployment on actual hardware. To help breaking down the remaining barriers for software engineers, this paper provides guidance for mastering the first steps of hardware development using the CAPI SNAP framework. The instructions reported in this paper are based on the insights of graduate students in software engineering, collected over the course of multiple student projects, with the participants having little to no prior knowledge about hardware development. The instructions apply to on-premise setups as well as to the *SuperVessel Cloud for OpenPower* [2] service.

Hereinafter, this paper is structured as follows: Enabling tight integration of accelerators, Sect. 2 introduces the basic concepts of CAPI. Section 3 provides an overview of the major traits of the CAPI SNAP framework. Afterwards, Sect. 4 reports best practices for getting started with the framework. Finally, Sect. 5 discusses related work, before an outlook is provided in Sect. 6.

2 Understanding CAPI

The *Coherent Accelerator Processor Interface* (CAPI) is an interface standard introduced with the IBM POWER8 architecture [12]. It enables accelerators to partake in the processors coherent view on the memory hierarchy. Prior to CAPI, accelerator resources had to be mapped to specific IO memory areas, where data had to be copied to and from explicitly. CAPI-enabled accelerators can access the same virtual address space as its controlling process, drastically curtailing the overhead for interacting with accelerators [13]. In its initial version, CAPI is layered on top of PCIe 3.0. In the upcoming POWER9 architecture, CAPI will be extended to support custom I/O facilities in addition to PCIe 4.0.

2.1 Architecture

CAPI involves several components on the host CPU as well as on the accelerator side. The FPGA side is comprised of *Accelerator Function Units* (AFUs),

implementing the application logic, as well as the *Power Service Layer* (PSL), which is a fixed design provided by IBM for supported FPGA cards [5].

The PSL communicates with the host part of the CAPI hardware, the *Coherent Accelerator Processor Proxy* (CAPP) via PCIe. The CAPP is part of the POWER CPU and from the point of view of the memory subsystem, it has the same status as a processor core. The software side of CAPI consists of a driver in the linux kernel, exposing installed CAPI accelerator cards as *cxl* devices. To encapsulate the interaction with raw *cxl* devices, the *libcxl* provides a user-land C API with the same functionality. Given sufficient privileges, any user application can interact with the AFUs on a cxl device by linking against *libcxl*.

2.2 Development

AFUs have to be expressed in low-level hardware description languages such as VHDL or Verilog, differing significantly from imperative languages like C in that most statements have concurrent semantics. The interface between PSL and AFU facilitates efficient communication, however its complexity imposes high efforts on AFU developers. Demonstrating the degree of complexity, Fig. 1 illustrates the state machine of a simple AFU for adding of two numbers stored in host memory. The AFU-PSL interface consists of five semi-independent sets of signals:

– The *Job-Interface* is controlled by the PSL and indicates job control and reset commands issued by the host.
– The *MMIO-Interface* exposes a register view of the AFU to the host, which can map this view into its virtual memory to control and monitor the AFU.
– The *Command-Interface* is controlled by the AFU, which can issue a variety of read or write commands with different side effects on the cache hierarchy.
– The *Response-Interface* and *Buffer-Interface* are controlled by the PSL and are used to complete pending commands (e.g. read and write).

For further details on implementing AFUs directly on top of CAPI, please refer to the tutorial "Tinkering with CAPI" by Keneth Wilke [14].

3 The CAPI SNAP Framework in a Nutshell

While CAPI provides the technical foundations for tightly coupling accelerators with CPUs (see Sect. 2), the technology is hard to adopt for software engineers. With the goal of making it as easy as possible for software engineers to leverage CAPI-enabled FPGA hardware acceleration, the *CAPI Storage, Network, and Analytics Programming* (SNAP) framework [10,11] has been introduced recently. The framework assists developers in various aspects explicated hereinafter. Also the acceleration paradigms supported by the framework are discussed.

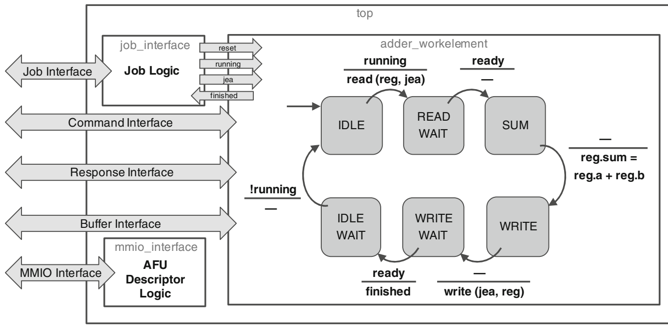


Fig. 1. The state diagram of a simple adder AFU demonstrates the complexity of developing AFUs directly on top of CAPI.

3.1 Core Features

High-Level Language Support. Having to implement application logic using low-level hardware description languages such as VHDL or Verilog, and switching from procedural to state-based thinking is very challenging for most software engineers. CAPI SNAP lowers the hurdles significantly by providing high-level language support based on HLS C/C++.

Job Interface. Calling an action using CAPI requires a lot of complexity in the calling application in order maintain all required communication channels. The framework provides a simple API for interacting with AFUs, allowing actions to be issued by creating a job based on a filled parameter struct, e.g. via the blocking call `snap_action_sync_execute_job(action, &job, timeout)`.

Hardware Abstraction. All FPGA extension cards supported by CAPI SNAP offer peripheral hardware components such as DRAM, NVMe storage or network interfaces. Without the framework, developers would have to implement interface logic and data movers for leveraging the peripheral components, also requiring data movers for interacting with host memory. As illustrated in Fig. 2, the CAPI SNAP framework hides a lot of this complexity by providing simple interfaces, abstracting away the specific details of both peripheral components and the specifics of the FPGA chip itself.

Automated Build Process. Using the Xilinx Vivado Design Suite [16] as a foundation, the development workflow for CAPI SNAP based AFUs is comprised of many stages, including software development, hardware development, hardware simulation as well as hardware deployment. As visualized in Fig. 3, each stage requires its own complex set of tools — originating from various sources (Vivado, CAPI SNAP and CAPI) — in order to create functional builds. Orchestrating all of these tools properly is a very complex task, which is being taken

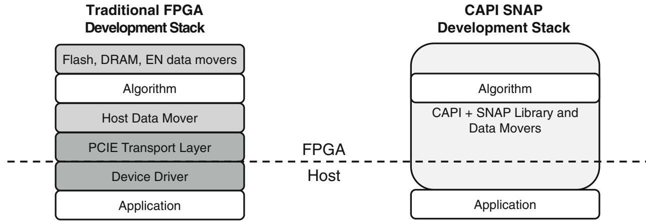


Fig. 2. CAPI SNAP hides complexity on the layers between AFUs and the host application, as well as for accessing peripheral components on the FPGA card. Illustration adapted from [6].

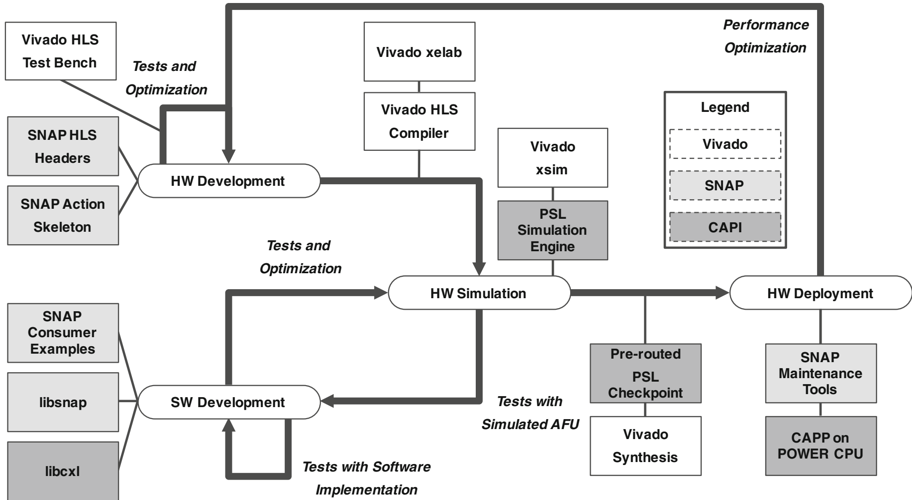
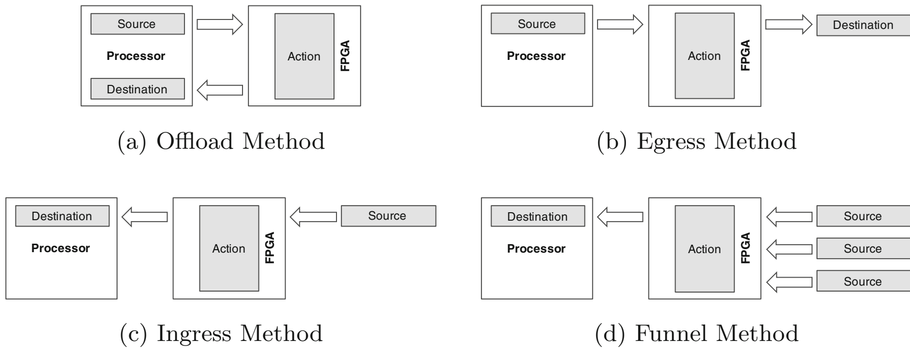


Fig. 3. CAPI SNAP automates the complex build process by orchestrating a wide range of tools originating from various sources.

care of by the CAPI SNAP framework, freeing up many resources on the developers end.

3.2 Acceleration Paradigms

In addition to the well-established *Offload* paradigm commonly used in the field of GPU computing (see Fig. 4a), the availability of peripheral components on the FPGA card enables the CAPI SNAP framework to support a variety of acceleration methods. The *Egress* and *Ingress* methods (see Fig. 4b and c, respectively) can be applied in scenarios where data streams leaving or entering the system (e.g. via network or persistent storage) need to be processed on-the-fly. Use cases for these methods include transparent encryption or compression, as well as media-processing tasks. The *Funnel* method (see Fig. 4d) is eligible in scenarios where the input bandwidth of all external sources exceeds the ingestion



(a) Offload Method (b) Egress Method

(c) Ingress Method (d) Funnel Method

Fig. 4. Due to the availability of peripheral components on the FPGA card, CAPI SNAP supports various acceleration paradigms next to traditional offloading.

capabilities of the host. Potential use cases include filter or aggregation tasks on incoming sensor data, as well as database-like operations such as joins, intersections, and merges on large datasets residing on external storage.

4 Getting Started with CAPI SNAP

This section provides an overview of the most important steps for getting started with CAPI SNAP, covering the *basic setup* of the development environment, setup and execution of a *simulation model* for testing purposes, the setup of a *test bench* for validation, as well as deployment and invocation of AFUs on real *hardware*.

4.1 Basic Setup

Setting up a development environment for CAPI SNAP involves several components, including the *Vivado Design Suite*, the *Power Service Layer Checkpoint*, the *Power Service Layer Simulation Engine*, and last but not least the *CAPI SNAP Framework* itself. In the following, the setup process of all these components is documented.

Vivado Design Suite. The Xilinx Vivado Design Suite [16] provides the foundation for the CAPI SNAP framework. Being the centerpiece, the Vivado IDE is used to synthesize and layout actions, for providing *High Level Synthesis* (HLS) C/C++ support, as well as for simulating designs without the actual hardware using *xsim* (Vivado Simulator).

Power Service Layer Checkpoint. On the FPGA, the Power Service Layer (PSL) manages the communication with the host (see Subsect. 2.1). This includes translating memory addresses, handling interrupts and virtualizing AFUs if necessary. Since the PSL component needs to be part of the FPGA bitstream, IBM

provides the PSL for download as a pre-routed checkpoint (`.dcp`) file [5]. Care should be taken to pick the correct checkpoint file for the FPGA card at hands, since each card requires a different checkpoint file.

Power Service Layer Simulation Engine. In order to augment the Vivado Simulator *xsim* with CAPI-like behavior, the *Power Service Layer Simulation Engine* (PSLSE) is required additionally, which is is freely available for download [4]. The PSLSE implements the PSL in software and connects the (locally hosted) simulation server with the desired action. The host application then communicates with the (locally hosted) PSLSE server instead of actual hardware. Since hardware synthesis is a very time-consuming process, simulation is usually preferred over hardware deployment for quick testing purposes during development.

CAPI SNAP Framework. After having downloaded the CAPI SNAP framework from [11], the Vivado environment must be established by sourcing the `settings64.sh` script and exporting the location of a valid license file. To ensure that every terminal session has a Vivado environment, the lines in Listing 1.1 might be added to the local shell initialization script (e.g. `~/.bashrc`).

```
1 source /opt/Xilinx/Vivado/2016.4/settings64.sh
2 export XILINXD_LICENSE_FILE=<path to Xilinx license>
```

Listing 1.1. Setup of the Vivado environment in a new terminal session.

The SNAP build process requires the locations of several dependencies. These should be specified in the `snap_env.sh` script in the SNAP root directory. The setup is finally completed by executing `make snap_config` in the CAPI SNAP root directory. This opens an interactive menu to specify the build configuration. After saving the choices and leaving the menu, SNAP shows a summary of the chosen configuration similar to Listing 1.2.

```
 1 =======================================================
 2 == SNAP SETUP                                                    ==
 3 =======================================================
 4 =====Checking Xilinx Vivado:===========================
 5 Path to vivado        is set to: /opt/Xilinx/Vivado/2016.4/bin/vivado
 6 Vivado version        is set to: Vivado v2016.4 (64-bit)
 7 =====CARD variables===================================
 8 FPGACARD              is set to: "FGT"
 9 FPGACHIP              is set to: "xcku060-ffva1156-2-e"
10 PSL_DCP               is set to: "/tmp/cards/FGT/b_route_design.dcp"
11 =====SNAP PATH variables==============================
12 SNAP_ROOT             is set to: "/tmp/snap"
13 ACTION_ROOT           is set to: "/tmp/snap/actions/hdl_example"
14 =====SNAP simulation variables========================
15 SIMULATOR             is set to: "xsim"
16 =====SNAP function variables==========================
17 NUM_OF_ACTIONS        is set to: "1"
18 SDRAM_USED            is set to: "FALSE"
19 NVME_USED             is set to: "FALSE"
20 ILA_DEBUG             is set to: "FALSE"
21 FACTORY_IMAGE         is set to: "FALSE"
```

Listing 1.2. Output yielded from the execution of `make snap_config`.

Depending on which card is used, the variable `FPGACARD` has to be set correspondingly. At the time of writing, valid options for `FPGACARD` are `N250S`, `ADKU3`, and `S121B` for the Nallatech 250S, the Alpha Data KU3, and the Semptian NSA-121 FPGA-cards, respectively. The variables `SNAP_ROOT` and `SIMULATOR` are set automatically, making `xsim` the default simulator. However, `ACTION_ROOT` and the CAPI SNAP function variables (`SDRAM_USED`, `NVME_USED`) have to be set based on the action that should be build. Per default, the example `hdl_example` is build, requiring neither access to DRAM nor NVMe storage.

4.2 Simulating an Action

Simulation is a powerful tool during the development phase, as it enables developers to test the correct communication between AFUs and the host application by tracing the flow of binary signals. Simulation speed itself is much slower than the execution on real hardware. Therefore the simulation model does not include the PSL nor any part of the host side hardware. Nevertheless these components are essential for a host application to access the AFU under test. This issue can be sidestepped by using the PSLSE server. The PSLSE implements a higher level and thus faster model of the internal CAPI components. It provides a modified version of the *libcxl*, which uses the PSLSE server to access the virtual device instead of real CAPI hardware. The simulated PSL merely acts as a proxy, whose behavior on the signal level is controlled by the PSLSE server.

After CAPI SNAP has been configured, a simulation model of the user design can be built by running `make model` from the `SNAP_ROOT` directory. In this step, all framework components and the user design are compiled into a simulation model as well as a simulator configuration. The setup of the PSLSE server and its connection to the simulator is automated by the `sim` make target.

Executing `make sim` creates an interactive terminal session with the environment correctly set up to run applications on the simulated hardware. Before the action can be tested, it needs to be initialized as part of the discovery process implemented by the `snap_maint` tool. Afterwards the actual host application can interact with the simulated hardware action.

Leaving this session also stops the underlying simulation environment. During the simulation, traces of all signals are recorded in a wave database. Afterwards, the detailed operation of the hardware action can be explored by viewing the recorded traces with the `xsim --gui hardware/sim/xsim/latest/top.wdb` command.

4.3 Debugging in the Test Bench

Simulation is only rarely feasible for debugging AFUs implemented in HLS C/C++: The HLS code will be converted into VHDL/Verilog blocks that are quite hard to match to the HLS code. To facilitate debugging and validation of HLS code, setting up a test bench in Vivado enables developers to validate the correct behavior of their code by executing HLS code like a regular C/C++ program in a software debugger.

In order to enable software-based execution in the test bench, a main function needs to be added to the HLS code as explicated in Listing 1.3. To avoid the synthesis of the main function in later development steps, the function should be enclosed by the preprocessor conditional `#ifdef NO_SYNTH ... #endif`.

```
1 #ifdef NO_SYNTH
2 int main()
3 {
4     bf_halfBlock_t left = 0xda7a, right = 0xb10c;
5     printf("encrypt(0x%08x, 0x%08x) -> ", left, right);
6     bf_encrypt(left, right);
7     printf("0x%08x, 0x%08x\n", left, right);
8 }
9 #endif
```

Listing 1.3. In order to use CPU-based execution in the work bench, the HLS code needs to be augmented with a main function.

Before the test bench can be executed, the tested HLS source file needs to be added as a simulation source by right clicking *Test Bench* in the project explorer and selecting *Add Files*. Furthermore, the SNAP specific `CFLAGS` documented in Listing 1.4 must be set up by opening the *Project/Project Settings* dialog and editing the `CFLAGS` of the HLS source file in the *Simulation* tab. Afterwards, the execution can be started by pressing the *Run C Simulation* icon in the toolbar. After the execution has started, the *Debug* view will be entered, where the usual functionality of a C/C++ debugger is available.

```
1 -DNO_SYNTH -I./include -I../../software/include -I./<action_directory>/include
```

Listing 1.4. `CFLAGS` necessary for the work bench setup.

4.4 Running on Hardware

Once the AFU has been successfully tested in the test bench and the simulator, it can be deployed to the FPGA hardware. For that purpose, the command `make image` needs to be executed from the `SNAP_ROOT` directory in order to synthesize bitstream images. Synthesizing the bitstream image is a compute-intensive process and can take any time from several minutes up to a couple of hours, depending on the complexity of the action at hands. Once the build process has successfully finished, the resulting bitstream files can be found in the `hardware/build/Image` folder. The file ending in `*.bit` can be flashed to the FPGA using a JTAG programmer; the `*.bin` file is intended to be flashed using the `capi-flash-script`.

Programming via JTAG Programmer. For a new FPGA card straight from the factory, the operating system will not detect it as a CAPI-enabled device, since the pre-installed image on the factory partition of the FPGA doesn't support CAPI. Hence, a suitable image needs to be flashed onto the user partition using an external JTAG programmer. While this process is slightly cumbersome, it usually has to be performed only once in the lifetime of the FPGA card. Afterwards, new bitstreams can be flashed from the host system.

On the machine connected to the JTAG programmer, a light-weight version of Vivado including the hardware server tool `hw_server` is sufficient. Once the Vivado *Hardware Manager* has successfully connected to the FPGA, the activity LEDs on the programmer should turn on.

If the FPGA card has not been detected as a CAPI device yet, the user partition of the FPGA will be cleared upon each power cycle of the POWER machine. Hence, for initialization purposes, a bitstream image needs to be flashed after the system has been powered on, but before the operating system performs the PCIe walk. The timespan in between the power cycle and booting the operating system kernel should be sufficient to finish the programming process before the host operating system has completed the boot process. Once this procedure has been completed, the FPGA should be appear under `/dev/cxl`.

Programming from the Host Machine. Once the FPGA is detected as a CAPI-device appearing under `/dev/cxl`, the `capi-flash-script` utility can be used to flash new bitstreams directly from the host. The tool is part of the `capi-utils`, which are available on GitHub [3].

5 Related Work

There are several technologies for leveraging FPGA compute resources in applications using high-level programming languages. The approaches can be loosely or tightly coupled. Intel offers a tightly coupled integration with *The Open Programmable Acceleration Engine* (OPAE) [9]. In many aspects, the approach is similar to IBM CAPI SNAP. It consists of libraries and kernel drivers offering resource management and abstraction of the underlying FPGA technology to the application developer. The OPAE C-Library [7] (`libopae-c`) is used by the applications to communicate with the FPGA. The building blocks on the FPGA device are comprised of a static part, the FPGA Management Engine (FME) and as many slots with accelerated function units (AFUs) as the device supports [9]. The AFUs an be partially reconfigured during runtime. One slot and one AFU form a function which can either be physical or virtual. The kernel driver supports SR-IOV so that virtual functions can be assigned to virtual machines [18]. The OPAE and CAPI SNAP are similar but also differ in several aspects, f.e. in OPAE there is no Job Management, the interface to the AFUs is given via a freely defined 256 KB Registers which have to be mapped into the address space of the host process to communicate.

There are also other approaches for leveraging FPGA accelerators using high-level programming languages. With *SDAccel* [15] Xilinx offers a development environment to execute C, C++ and OpenCL Kernels on FPGA Hardware. The *Intel FPGA SDK for OpenCL* [8] offers a similar development environment. Due to the lack of coherent host memory access, both technologies offer a more loosely coupled integration of the FPGA resources.

6 Outlook

To alleviate the complexity of developing FPGA-based accelerator functions for software engineers, the *OpenPOWER Accelerator Work Group* has recently introduced the *CAPI Storage, Network, and Analytics Programming* (SNAP) framework. Over the course of multiple graduate student projects, we have observed that the high level of abstraction provided by CAPI SNAP in conjunction with HLS, the framework enabled students to implement common algorithms in hardware and evaluate these accelerator-based resources within one semester. However, even though CAPI SNAP is well documented and comes with many examples, we have noticed that graduate students in software engineering found themselves challenged with certain details of the novel hardware development framework. At the same time, we also found that the framework helped students to improve their understanding of hardware development, as CAPI SNAP allowed them to concentrate on implementing application logic using a hardware description language without having to consider the complexity of any interface and management logic. In this paper, we consolidated these insights into a getting started guide, providing the background knowledge and the first instructions necessary for breaking down the remaining barriers for software engineers.

With the CAPI SNAP framework being a relatively young technology compared to well-established frameworks for heterogeneous computing, we think that it offers great potential for bridging the gap between hardware development and software engineering, allowing software engineers to tap into the extended solution space offered by the more flexible resources that FPGAs can offer. For users without access to IBM POWER systems and CAPI-supported FPGA cards, we recommend using the *SuperVessel Cloud for OpenPower* [2] service, which offers cloud-based access to CAPI-enabled resources for academic researchers. Also, we would like to stress that the active community behind CAPI SNAP has been very open-minded and forthcoming regarding feedback we provided. In general, the community-character is a welcome change to the closed, vendor-specific nature of other ecosystems met in the field of GPU-computing.

Since the limited space of a paper does not offer the ideal venue for a detailed hands-on guide, this paper is augmented with an extended online tutorial, which is available at https://www.dcl.hpi.uni-potsdam.de/capi-snap. The extended online tutorial covers several aspects of the CAPI SNAP framework, including setup, configuration and debugging in greater detail. Furthermore, it provides an additional section that documents the process of developing a new HLS-based AFU step-by-step, using the blowfish encryption algorithm as an exemplary workload.

Acknowledgements. We would like to thank everyone at IBM who held close contact and helped us during the project, including but not limited to: Frank Haverkamp, Jörg-Stephan Vogt, Sven Boekholt, Thomas Fuchs, Bruno Mesnet, Nicolas Mäding, and Bruce Wile.

References

1. Fowers, J., Brown, G., Cooke, P., Stitt, G.: A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2012, pp. 47–56. ACM, New York (2012)
2. IBM China Research Lab: SuperVessel Cloud for POWER/OpenPower. https://ptopenlab.com/
3. IBM Corporation: capi-utils package. GitHub. https://github.com/ibm-capi/capi-utils
4. IBM Corporation: Power Service Layer Simulation Engine (PSLSE). GitHub. https://github.com/ibm-capi/pslse
5. IBM Corporation: PSL Checkpoint Files for the CAPI SNAP Design Kit. https://www-355.ibm.com/systems/power/openpower/tgcmDocumentRepository.xhtml?aliasId=CAPI
6. IBM Corporation: CAPI SNAP Education Series: Module #1 - CAPI SNAP Overview (2017) (Presentation)
7. Intel Corporation: Github Organisation for the Open Programmable Acceleration Engine. https://github.com/OPAE
8. Intel Corporation: Intel FPGA SDK for OpenCL, December 2017. https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
9. Luebbers, E., Liu, S., Chu, M.: Simplify Software Integration for FPGA Accelerators with OPAE (White Paper). https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf
10. OpenPOWER Accelerator Work Group: CAPI Storage, Network, and Analytics Programming (SNAP) Framework. IBM developerWorks. https://developer.ibm.com/linuxonpower/capi/snap/
11. OpenPOWER Accelerator Work Group: CAPI Storage, Network, and Analytics Programming (SNAP) Framework Repository. GitHub. https://github.com/open-power/snap
12. Stuecheli, J., Blaner, B., Johns, C.R., Siegel, M.S.: CAPI: a coherent accelerator processor interface. IBM J. Res. Dev. **59**(1), 7:1–7:7 (2015)
13. Wile, B.: Coherent Accelerator Processor Proxy (CAPI) on POWER8, October 2014. presented at Enterprise 2014
14. Wilke, K.: Tinkering with CAPI. Such Programming, January 2016. https://www.suchprogramming.com/tinkering-with-capi/
15. Xilinx Corporation: The Xilinx SDAccel Development Environment. https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf
16. Xilinx Corporation: Vivado Design Suite. Product Website. https://www.xilinx.com/products/design-tools/vivado.html
17. Xilinx Inc.: Xilinx Introduces Zynq-7000 Family, Industry's First Extensible Processing Platform, March 2011. Press Release
18. Zhang, Z.: Getting Started With Open Programmable Acceleration Engine, August 2017. Webinar. https://www.brighttalk.com/webcast/10773/275799?utm_source=Intel+-+Data+Center+Group&utm_medium=brighttalk&utm_campaign=275799