Unified OpenCL Integration Methodology for FPGA Designs

Topi Leppänen*, Panagiotis Mousouliotis†, Georgios Keramidas†,
Joonas Multanen* and Pekka Jääskeläinen*

*Tampere University, Tampere, Finland
Email: {topi.leppanen,joonas.multanen,pekka.jaaskelainen}@tuni.fi

†University of the Peloponnese, Patras, Greece
Email: {p.mousouliotis,g.keramidas}@esdalab.ece.uop.gr

***Abstract*—OpenCL is a widely adopted open standard for general purpose programming of diverse heterogeneous parallel platforms that can harness various device types such as CPUs, DSPs, GPUs, FPGAs and hardware accelerators. It is an extensive and explicit low level API serving well as a platform portability layer. However, using OpenCL for diverse heterogeneous programming in multi-vendor platforms is not practical due to device vendors each providing their own OpenCL implementations which do not interoperate efficiently, leading to inefficient execution coordination and collaborative execution between various device types from different vendors.**

To this end, this paper proposes a vendor-independent open source method for integration of custom FPGA components to a common OpenCL platform. The method relies on a streamlined memory-mapped hardware control interface implemented by the integrated components. The required OpenCL driver integration is then automatically provided, enabling easy inclusion of different types of FPGA accelerators to the control of a single OpenCL runtime.

The ease of integration and portability is demonstrated by integrating two hardware devices in two different FPGA devices. The resource overhead of the hardware interface is shown to be negligible and the clock frequency overheads small enough to not pose efficiency challenges.

***Index Terms*—heterogeneous computing, hardware integration, hardware acceleration, FPGA, OpenCL**

I. Introduction

Heterogeneous computing aims to use specialized hardware for increased energy efficiency, faster execution time or lower resource footprint. In this paradigm, ideally, each task should be executed on a hardware device which is optimized for that task at hand. However, in practice it is unrealistic to assume each task would have a dedicated hardware accelerator, but programmability is desired to add flexibility to the functionality provided by the device.

Realizing the heterogeneous computing paradigm in practice has various challenges: How to control the different types of devices and execute tasks on them? How to ensure that interoperability is maintained between devices in *diverse heterogeneous platforms* which integrate more than one specialized compute device? How to do all of this in a manner so that the heterogeneous application is *portable*, at least at the source code level, to avoid extensive re-engineering whenever even a single device of the heterogeneous platform is changed to another?

Open Computing Language (OpenCL [1]) is an open standard that provides a low level API that enables utilization of diverse heterogeneous platforms for general compute purposes. It is adopted by a wide range of hardware vendors and fulfils relatively well the *portability* aspect: In theory, when writing an application against the OpenCL API, it is possible to make it portable to any device which claims OpenCL support. However, in practice, this is happening only in a limited manner; each processor vendor has their own OpenCL implementation that can control their own devices, but not those from other vendors. This limits the capabilities of handling diverse multi-vendor heterogeneous platforms *as a whole*, implementing cross-device concerning aspects such as synchronization or data transfers in the most optimal manner.

The open source OpenCL implementation Portable Computing Language (PoCL, [2]) was created to address the concern of multiple vendor implementations having limited interoperability. Its key motivation is not only to be a framework for more easily adding single-vendor OpenCL support for new hardware, but eventually to integrate all types of devices from all vendors to a common OpenCL platform, allowing their utilization as a collaborating entity rather as multiple "isolated vendor islands".

In the recent years, the key FPGA vendors have embraced OpenCL as an input in their *High-Level Synthesis (HLS)* toolflows, significantly easing the programming of the FPGA fabrics and thus reducing the need for digital hardware design expertise to utilize them efficiently. However, the OpenCL-based HLS tools are not compatible, requiring vendor-specific source code modifications for efficient implementation [3], and cannot function efficiently with devices from other vendors in the same context.

In this paper, we propose a method, along with the necessary hardware and software, to integrate functionality implemented in FPGA devices to PoCL, a common open source OpenCL platform. We identify the key contributions of this work as the

978-1-6654-0712-0/21/$31.00 ©2021 IEEE

following:

- A common hardware interface specification for non-programmable and programmable co-processors that enables OpenCL kernel execution and device-to-device work queuing.
- Unified means of utilizing FPGA fabrics from different vendors from standard OpenCL host programs.
- Integration of FPGA-based accelerators to OpenCL platforms with CPUs from multiple vendors for seamless CPU+FPGA execution.

The paper is structured as follows. Section II covers the key concepts in two open standards related to heterogeneous computing used in this work. Section III describes the hardware interface used to unify the "outside view" of the integrated components. Section IV provides the software point of view in terms of OpenCL implementation and the application writer. Section V shows the steps required to integrate a new hardware component to a common OpenCL platform in the proposed method. Section VI presents the measurements that were made to ensure the method produces insignificant overheads at FPGA implementation side. In Section VII we discuss how the proposed integration method relates to other previous technologies, and Section VIII concludes the paper.

II. Open Heterogeneous Computing Standards

Open Computing Language (OpenCL [1]) is an interesting API for providing the lowest layer in heterogeneous computing software stacks since it is being widely adopted, having an extensive feature set, as well as being an openly available standard.

OpenCL follows a model where there is a single host CPU running the main program that commands one or more co-processors (devices) which can execute tasks asynchronously or synchronously. The host program queues work to the devices through *command queues* (CQs). A typical *command* pushed to the queue is a *kernel command* which executes a function on the device, often in the data parallel *Single Program Multiple Data (SPMD)* fashion. Here a *kernel function* defines what a single instance (*work-item*) does in a possibly multidimensional grid of instances.

Runtime portability of software across the variety of heterogeneous devices is achieved by means of *online compilation*, which is a standardized way for invoking cross compilers targeting the device at hand *during the host program runtime*. Relevant to the proposed work, a very underused feature in the standard is the concept of a *custom device* which abstracts hardware accelerators which do *not* necessarily support online compilation of any general purpose functions, but instead present a set of *built-in kernels* for invoking fixed functions.

Whereas OpenCL has been mostly used and considered as a "single compute node" API with other (typically message passing APIs) applied on top for computer cluster utilization, there have been experiments that have shown that the programming model is suitable also for (low latency) *distributed execution* [4], extending its scope farther.

Another open heterogeneous computing standard relevant to this work is Heterogeneous System Architecture [5]. HSA is an ambitious effort of providing a standard where the heterogeneous devices are mapped to a unified coherent address space for easier data sharing across the platform. The overall concept of presenting a unified memory mapped interface which the devices (called agents in the HSA specification) can utilize for execution coordination has inspired the proposed work. Thanks to HSA being an open specification, it has been utilized heavily in the proposed integration method.

III. AlmaIF v2: The Common Hardware Interface

Driver implementation and its interfacing to the next layers in the application software stack has major non-recurring engineering costs when integrating new hardware to software platforms. The means to invoke the different functionalities required by the lowest software layers in the integrated hardware block is device-specific knowledge, which complicates software integration efforts. Furthermore, in order to implement fully asynchronous execution where the devices communicate and synchronize with each other *directly* without host CPU involvement in a peer-to-peer fashion, each device pair involved must know the means to do so.

A unified software integration solution and feasible peer-to-peer communication in diverse heterogeneous platforms calls for a unified hardware interface which enables common means to access the devices' functionalities. The unified hardware interface would act as a well-specified hardware wrapper that presents the necessary information and a common means of control to the software side and the other devices. The goal is to make the integrated hardware blocks easily "pluggable" to the software stack with minimal integration engineering work, and avoid the need for target device-specific state machines or code paths in the devices that wish to communicate with each other directly.

The link between the integrated component and the software driver in the proposed integration method is a memory-mapped hardware interface with four memory regions reserved for different purposes as exemplified in Fig. 1. The interface is called AlmaIF v2. It is named after the first research project (ALMARVI) in which the first version of it was prototyped (**Alma**rvi hardware **I**nter**F**ace) [6]. For this work, we utilized the AlmaIF v1 as a basis and extended it with support for built-in kernels to integrate fixed function hardware and additional optimizations necessary for more flexible memory usage. The roles of the memory mapped regions in the interface are explained in the following.

Control registers region includes the registers used to initialize and reset the component. It also contains constant value registers describing the essential device features for *automatic device discovery*. The data for discovering the essential features of the device are needed by the OpenCL runtime, and is made part of the hardware interface to ensure easy plugging of components with minimal data passed to the driver through other means. This includes, e.g., the structure of the memory map for initializing the memory manager of

TABLE I
ALMAIF V2: MEMORY MAPPED CONTROL REGISTERS

Offset	Bits	Name	Purpose / explanation
0x000	3	Status	Status of the accelerator. Bit 0 is high when the execution is stalled due to any reason, bit 1 is high when the external freeze signal (pauses the hardware temporarily) is active, and bit 2 is high when the accelerator reset is active.
0x100	64	Command queue read position index	Only increased (wraps around). The actual position in the CQ is determined using a modulo with the CQ size.
0x108	64	Command queue write position index	Only increased (wraps around). The actual position in the CQ is determined using a modulo with the CQ size.
0x200	3	Command	Command register to control execution. Writing 1 to this register resets the accelerator, writing 2 lifts reset and the external freeze, and writing 4 enables the external freeze signal, pausing execution (4 is an optional feature).
0x300	32	Device class	Optional OpenCL vendor ID of the component.
0x304	32	Device ID	Device ID of the accelerator. Currently unused by the driver.
0x308	32	AlmaIF version	Version number of the interface. Currently at value 2
0x30C	32	Core count	Number of compute units in the device
0x314	32	Configuration memory size	Size of the configuration memory region in bytes.
0x318	64	Configuration memory starting address	Starting address of the device's configuration memory region.
0x320	64	Command queue memory size	The command queue ring buffer fills the entire region, so the size of this memory is Max_number_of_packets * packet_size. Maximum number of packets must be a power-of-two.
0x328	64	Command queue starting address	Starting address of the device's command queue memory.
0x330	64	Data memory size	Size of the data memory region.
0x338	64	Data memory start	Starting address of the device's data memory.
0x340	64	Feature flags	Boolean feature flags. Bit semantics: Bit 0 = Bus master interface available. The device can access the whole memory space through a master interface.
0x348	16	Number of built-in (N_b) kernels supported	Defines the number of functions supported (max 64 in AlmaIF v2). The built-in kernel IDs are given in the successive memory locations.
0x34A ...	16 x N_b	The ID(s) of the supported built-in kernel(s)	There are as many as the component supports in successive memory locations.

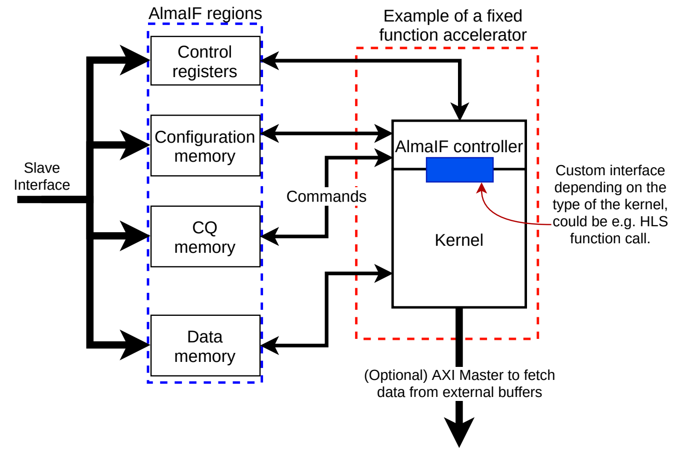

Fig. 1. AlmaIF v2 memory regions highlighted with blue dashed line, with an example wrapped hardware component (in this case a fixed function accelerator) highlighted in red.

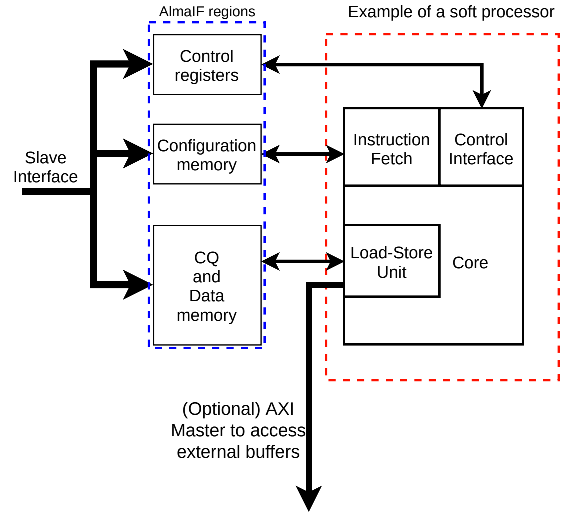

Fig. 2. AlmaIF v2 memory regions highlighted with blue dashed line, with an example wrapped hardware component (in this case a software programmable core) is highlighted in red. Here the *configuration memory* is used to store the program image. In addition, this design utilizes the same physical memory for mapping both the *Command queue* and the *Data memories*.

the software side. Table I shows the type, address, and size of the control data accessible through this region.

The region also includes the read and write indices to the command queue ring buffer, part of the *command queue memory*. The read index is assumed to be atomically incremented by the component after it has picked a command for execution, whereas the write index is maintained by the host or the peers to queue commands to the device. These indices are used by the driver to assess if there is space in the CQ for more work and by the device if there are more commands to process.

Configuration memory is optional and not needed for fixed function accelerators. It is used for uploading the program binary images in case of software programmable devices (see Fig. 2). It can be also utilized for target-specific

firmware/configuration bit purposes to better support, for example, reconfigurable multi-function accelerators (left as a future work).

Command queue memory contains a ring buffer implementing a command queue that the device should execute next. The commands in the queue follow the HSA Architected Queue Language (AQL) specification [5], which defines the command structure down to bit level accuracy, and thus serves as an excellent source for more detailed documentation. It suffices to summarize here that the integrated devices focus on two main types of AQL packets in the command queue: 1) *Kernel Dispatch Packets*, which are used to launch kernels in the device and 2) *Barrier-AND Packets* which are used to implement OpenCL event based synchronization utilizing "signaling slots" that are memory locations which are monitored for a change to zero, indicating event completion.

Data memory is a piece of on-chip memory where the OpenCL runtime can allocate data which are expected to be frequently accessed by the device when it executes kernels, thus should be stored in a physically close memory for fast access. This data includes the OpenCL kernel command's function arguments, in some cases the buffers used by the kernel, and the optional OpenCL kernel command metadata (e.g. work-group sizes).

IV. Software Interface

The software interface of the integration method is built on the open source PoCL [2] framework to benefit from its code base and to seamlessly connect to other devices it already supports. PoCL provides a user space driver layer where a new general driver we call *custom* was implemented. It interacts with the common hardware interface and in the typical case requires none or very little custom software side code to be added for new integrated FPGA components.

There are two main categories of integrated devices that the *custom* PoCL driver supports: 1) Fixed-function accelerators, that cannot be software programmed, but are exposed as OpenCL custom devices with a set of built-in kernels and 2) co-processors which provide support for any OpenCL kernel compiled from source code or one of the intermediate languages supported by the standard. The latter can also support fixed acceleration functionality. We outline the support for these accelerator types in the following.

A. Fixed-function accelerators

The fixed-function accelerators can only execute built-in functionality, meaning something with predefined semantics which cannot be reconfigured. The wrapped component is expected to advertise the built-in kernels it supports by exposing them in its control interface.

The built-in kernels are invoked by using integer identifiers, which map the kernel to its semantics. Since the semantics of the built-in kernels in the OpenCL specifications are undefined, and only referred by their name (C string) in the OpenCL-using programs, there has to be a specification for the meanings of known built-in kernels (with an assigned integer ID) which provides the guidance to the OpenCL application writers. This semantical connection is provided as part of the integration method: A registry is maintained over time that lists known and supported built-in kernels along with their exact semantics as well as the bit exact definitions of parameter passing. This registry is maintained within the PoCL source code base and will be kept up-to-date with functionalities of interest. A range of IDs is dedicated for custom functionality defined by the FPGA designer that needs not be defined in the registry.

The built-in kernels are invoked in the wrapped accelerator by specifying the integer ID to launch in the kernel command packet queued to the device. It is then the responsibility of the component to invoke the functionality requested by utilizing the arguments in the data memory.

B. Co-processors with software programmability

Software programming capabilities as defined by the "online kernel compilation support" of the OpenCL specifications are supported by the integration. The feature relies on the PoCL's standard compilation passes for converting the input definitions down to executable binaries by utilizing the LLVM [7] Project as the compilation infrastructure.

The integrated component can advertise the compiler support by defining a special built-in kernel ID in its list of supported built-in kernels. The special ID is the maximum built-in kernel ID value of 0xFFFF in AlmaIF v2.

Fig. 3 shows how the integrated component can tap into the OpenCL kernel compilation mechanics of PoCL: When the device driver is initialized for the component in question, *custom* driver queries from the *control registers* region if it finds the special built-in ID. If it is found, the next step is to figure out how to generate code for the device. For this, the vendor ID is utilized; it is used to find the correct LLVM "target triple" for the device at hand which is needed to perform target-specific code generation.

In case the target has an LLVM backend available in the LLVM the PoCL was built against, there is no need to define anything in the driver for invoking the correct compiler since the PoCL kernel build process is portable and generic in most parts. Only the code generation phase differs for each instruction-set architecture. However, if special compilation steps are required, the compilation steps can be overriden per triple. One such (likely rare) special case is the TCE (also known as the OpenASIP) [8] created devices which require a target description file to be given at code generation time to drive the instruction scheduler due to the customizable processor instruction set architecture.

V. Steps to integrate a new component

Since the contribution of this paper is a new method for easy integration of custom FPGA components to a unified and open OpenCL platform, it is necessary to summarize the steps which the FPGA component integrators need to go through:

1) **Hardware component wrapping.** To be able to plug in a new hardware component, it is enough to create a hardware

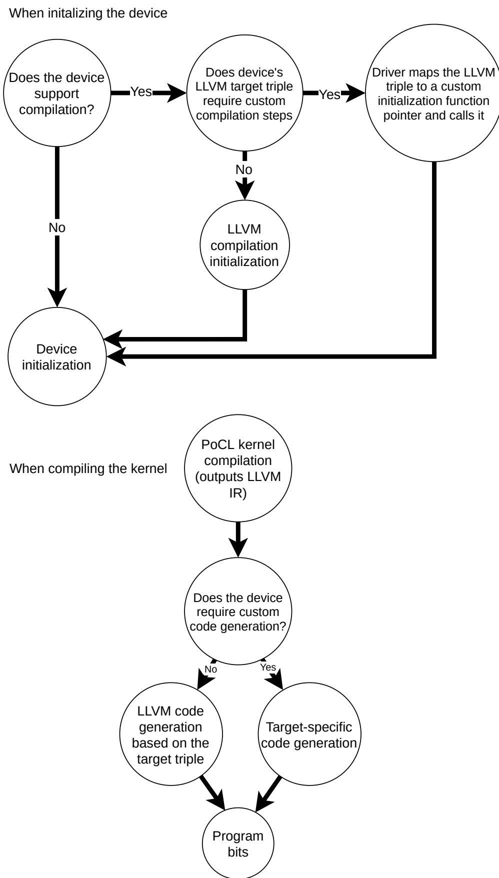

Fig. 3. Supporting kernel compilation based on either LLVM target triple or custom target-specific compilation flow.

device with the memory map described in Section III and make sure that its memory map is accessible from the host CPU running PoCL. The component must be able to process the command queue, thus perform *packet processing* tasks and honor the SPMD execution model. This is made straightforward by providing a ready made template implementation in simplified C which passes through the most popular HLS flows and calls the kernel functionality for each work-item in the SPMD fashion. In the usual case, this is the only action needed.

2) *Optional step for completely new FPGA platforms:* **Enable access to the FPGA device's physical memory in the host.** The driver code relies on physical regions memory mapped to the host virtual address space. Therefore, it is enough to be able to map the FPGA physical memory regions to the host, or access them using FPGA device-specific APIs. For a typical user, this part is not needed, since several FPGA memory access methods are supported by the *custom* driver and more will be added in the future. For this paper, two methods were implemented as examples in the *custom* driver: Using simple **mmap**ing of the physical memory space for *system-on-chip* SoC based FPGAs and a Xilinx-specific access to the memory through library calls (for utilizing their PCIe card based FPGAs).

3) *Optional step for new built-in kernel functionality:* **Register new built-in kernels.** In case the integrated component implements a kernel that is not yet in the built-in kernel registry, the kernel can be added to the registry or a custom built-in kernel ID is used. Adding new built-in kernels is simple: Registering the kernel to the driver by defining a name and integer identifier pair and specifying the number and types of the arguments. This way, the driver knows how to fill the argument buffer for the built-in kernel. The arguments can be pointers to buffers or scalars.

4) *Optional step for online compiler supported components:* If the component supports **online kernel compilation**, it should advertise 0xFFFF in its supported built-in kernel IDs list. Then, to add the software side support, the steps depend on whether the target has a backend in the LLVM the PoCL was compiled with or not. In case it does not, a new entry has to be added to the *custom* driver, which overrides the necessary parts of the default PoCL kernel compilation process with the required alternative ones. One key parameter to define is whether the target inputs SPMD kernels (work-item functions) directly or if it needs the de-SPMD passes supported by PoCL to generate "work-group functions" which execute all the work-items in the work-group. The rest of the steps are handled by the framework: The program image is uploaded to the *configuration memory* region when the kernel needs to be launched and the compiled-in device main program handles also command queue processing tasks.

VI. Evaluation

In order to evaluate the implementation feasibility of the integration method and to demonstrate the applicability over different means to generate the integrated components, two example hardware components were integrated to two FPGA devices.

The integrated components were created with two different methods: One was a customized software programmable soft core design created using the OpenASIP tools [8] and the other was produced by feeding a C language-based implementation of the packet processing functionality and the built-in kernels to the commercial FPGA HLS tool Vitis HLS 2020.2 [9].

The portability of the integration method was tested by using two devices with different sizes and different methods for accessing the FPGA's physical memories. The smaller FPGA device was a SoC-based Xilinx Zynq XC7Z020 and the large one a PCIe card with an Xilinx Alveo U280.

For the purpose of focusing solely on the overheads from the integration method itself, the integrated components implement only two very simple built-in kernels: 32-bit integer element-wise vector addition and multiplication. The kernels are thus one dimensional; the single global size parameter indicates the length of the vector to process.

In order to evaluate the feasibility of the hardware wrapping needed for integration, the resource overhead and the clock frequency impact are the most interesting characteristics to assess. As seen from the results in Table II, the overhead from the control interface is small enough to be considered

TABLE II
FPGA RESOURCE UTILIZATION OF TWO HARDWARE COMPONENTS WRAPPED IN THE ALMAIFV2 WITH TWO DIFFERENT FPGA DEVICES. GIVEN AS TOTAL RESOURCES AND AS A PERCENTAGE OF ALL RESOURCES AVAILABLE ON THE FPGA FABRIC. THE RIGHTMOST COLUMN SHOWS THE MAXIMUM CLOCK FREQUENCY OF THE DESIGN IN THE MORE RESOURCE CONSTRAINED ZYNQ.

	LUTs	Registers	Block RAMs	FMax (MHz)
OpenASIP soft core				
Zynq XC7Z020	1952 (3.7%)	2166 (2.0%)	2 (1.4%)	162
Alveo U280	1931 (0.15%)	2163 (0.08%)	2 (0.10%)	
Vitis HLS generated component				
Zynq XC7Z020	561 (1.1%)	721 (0.68%)	1 (0.71%)	152
Alveo U280	627 (0.05%)	620 (0.02%)	1 (0.05%)	

negligible related to the total resources of the FPGA. This is especially visible with the Vitis HLS wrapped component implemented with the FPGA vendor's own proprietary tools. Here, the OpenASIP soft processor has approximately three times higher utilization than the minimal HLS-based component. This is due to the added logic and memory required by the software programmability.

We also performed a validation run where a CPU (in that case an Intel desktop CPU driven by the PoCL's *pthread* driver) and the FPGA component executing in the Alveo board were running in parallel in the same PoCL OpenCL context, controlled with different command queues. This proved that the unified CPU+FPGA collaborative execution functions in an integrated manner.

VII. Related Work

Intel Open Programmable Acceleration Engine (OPAE, [10]) is similar in its goals to the proposed integration method. It enables unified integration of components in Intel FPGAs to the higher-level software stacks, by means of a vendor-specific C library for the lowest level access. Similarly, Xilinx XRT [11] is a stack for managing Xilinx FPGA and ACAP devices. It works for both PCIe and MPSoC type Xilinx devices. The kernel control interface includes a method for launching, resetting and pipelining kernel executions along with kernel arguments.

Both OPAE and Xilinx XRT stacks resemble ideas in our integration method. A key difference is that their method has more of the functionality handled in the host code, whereas our proposed method is designed to be decentralized to enable asynchronous peer-to-peer execution between devices. Furthermore, while both Intel and Xilinx provide complete OpenCL flows from program description to the kernel execution in FPGAs, their methods are still specific to their own FPGA devices. Even though these vendors provide some of the sources for their flows in open source, there are no significant efforts for easy and efficient cross-vendor unification happening in practice. Additionally, there is no easy way to integrate different types of devices or devices from other vendors to their OpenCL context for asynchronous multi-command queue execution with event-based synchronization. This creates limitations on device-to-device inter-operability related to e.g. synchronization or efficient data transfers. In our proposed method the FPGA vendor-specific tools might only needed to access the FPGA physical memory, if even that.

HOpenCL [12] is an execution model inspired by OpenCL which supports the execution of both software kernels on soft processors and hardware kernels generated by e.g. high-level synthesis tools. The intention is similar to the proposed method, but in our case we do not deviate from the OpenCL specifications; our integration flow does not require any extensions to the OpenCL standard, but utilizes only its core specification features. This is important in order to maintain the OpenCL application-level portability between different platforms. Steinert et al. [13] proposed cloud acceleration with FPGAs, but the approach similarly relies on a custom API for invoking the accelerators instead of using a standardized execution model. Similarly, Galapagos [14] is a promising integration approach, but targets the MPI programming model whereas our work focus on the widely adopted OpenCL standard for maximal heterogeneous platform diversity.

VIII. Conclusions

In this paper we proposed a method to integrate FPGA-based accelerators to a unified OpenCL platform. The hardware overhead of the required component wrapper interface was measured to be insignificant. We identify the benefits of this work in enabling FPGA designers to loosen the dependency on vendor-specific OpenCL HLS flows and APIs, allowing them to treat their FPGA designs as any other OpenCL device in a unified OpenCL platform. In the future we plan to expand this work for efficient OpenCL 2.0 pipe implementation for portable on-chip streaming support and test the method on a wider range of FPGA devices from different vendors.

Acknowledgment

The work for this publication was funded by ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy. It was also supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware) and Academy of Finland (decision #331344). We would also like to thank Xilinx for donating the Alveo FPGA and its related software used in this work and HSA Foundation for the financial support and the useful specification work.

REFERENCES

[1] Khronos® OpenCL Working Group, "The OpenCL™ Specification," https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, accessed: 2021-08-27.

[2] P. Jääskeläinen, C. Sanchez de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015.

[3] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.

[4] J. Solanti, M. Babej, J. Ikkala, V. K. M. Vadakital, and P. Jääskeläinen, "PoCL-R: A Scalable Low Latency Distributed OpenCL Runtime," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS) XXI*, July 2021.

[5] HSA™ Foundation, "HSA Platform System Architecture Specification v1.2," http://hsa.glossner.org/wp-content/uploads/2021/02/HSA-SysArch-1.2.pdf, accessed: 2021-08-27.

[6] J. Hoozemans, J. van Straten, T. Viitanen, A. Tervo, J. Kadlec, and Z. Al-Ars, ""ALMARVI Execution Platform: Heterogeneous Video Processing SoC Platform on FPGA"," *Journal of Signal Processing Systems*, vol. 91, no. 1, pp. 61–73, Jan. 2019.

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[8] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, *HW/SW Co-design Toolset for Customization of Exposed Datapath Processors*. Springer International Publishing, 2017, pp. 147–164.

[9] *Introduction to Vitis HLS*, Xilinx. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/introductionvitishls.html

[10] E. Luebbers, S. Liu, and M. Chu, ""Simplify Software Integration for FPGA Accelerator with OPAE (white paper)"," 2017. [Online]. Available: https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf

[11] *Xilinx Runtime (XRT) Architecture*, Xilinx. [Online]. Available: https://xilinx.github.io/XRT/master/html/index.html

[12] H. Ding and M. Huang, "A unified OpenCL-flavor programming model with scalable hybrid hardware platform on FPGAs," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014, pp. 1–7.

[13] F. Steinert, P. Kreowsky, E. L. Wisotzky, C. Unger, and B. Stabernack, "A hardware/software framework for the integration of fpga-based accelerators into cloud computing infrastructures," in *2020 IEEE International Conference on Smart Cloud (SmartCloud)*, 2020, pp. 23–28.

[14] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow, "Galapagos: A full stack approach to fpga integration in the cloud," *IEEE Micro*, vol. 38, no. 6, pp. 18–24, 2018.