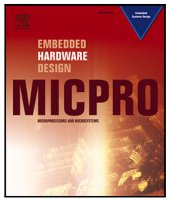
Microprocessors and Microsystems 97 (2023) 104772

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro



Efficient OpenCL system integration of non-blocking FPGA accelerators

Topi Leppänen [a,*], Atro Lotvonen [a], Panagiotis Mousouliotis [b], Joonas Multanen [a], Georgios Keramidas [b], Pekka Jääskeläinen [a]

[a] Tampere University, Tampere, Finland
[b] University of the Peloponnese, Patras, Greece

ARTICLE INFO

Keywords:
Heterogeneous computing
Hardware integration
Hardware acceleration
FPGA
OpenCL

ABSTRACT

OpenCL functions as a portability layer for diverse heterogeneous hardware platforms including CPUs, GPUs, FPGAs, and hardware accelerators. However, OpenCL programs utilizing multiple of these devices in the same computing platform suffer from poor coordination between OpenCL implementations of different hardware vendors. This paper proposes a vendor-independent open source method for integrating custom FPGA accelerators into a common OpenCL platform. The accelerators are wrapped in a common hardware interface to enable efficient synchronization and data sharing between devices on the same chip. The provided software connects the accelerator to OpenCL runtime and enables the control of diverse FPGA accelerators with OpenCL command queues.

The benefits of the integration methodology are demonstrated by creating FPGA accelerators with different development tools and integrating them together on two different types of FPGA devices while showing minimal integration overhead. Direct memory access of the accelerator to external memory is shown to increase the performance by a factor of 8. Non-blocking execution enabled by the on-chip synchronization between devices is shown to remove a 250 µs overhead from dependent kernel launches. Additionally, as a proof of concept and a case study, a fully OpenCL-controllable computing platform with two devices is implemented on an FPGA to compute CNN inference on a real-world input signal.

1. Introduction

Heterogeneous computing systems consist of multiple different types of processing devices. Having multiple specialized hardware components on the same system can significantly improve the energy, performance, and resource usage efficiency. Specialized hardware is tailored for a certain application or an application domain and can only efficiently execute the tasks it was designed to execute. The specialized hardware can still be software programmable, which adds flexibility to it.

Programming heterogeneous systems poses several challenges. The code must be compiled for all programmable components of the system, and the runtime must take care of device-specific quirks to control them during run time. Many tasks are collaborative, which means that multiple devices of the heterogeneous systems work together on the same tasks. Programming for heterogeneous systems requires platform-specific engineering efforts, which must be re-done whenever some part of the system or the application is changed. Ideally, the software algorithm is written once, after which it can be deployed to various different computing platforms with minimal effort. However, this level of portability is often lacking, which leads to significant costs when software is ported from one system to another.

Open Computing Language (OpenCL, [1]) is an open standard for programming heterogeneous systems. OpenCL program using low-level API calls is portable to very different types of OpenCL-conformant computing platforms. This makes OpenCL very useful as a portability layer between application libraries and heterogeneous platforms.

Many hardware vendors provide their own OpenCL implementations, which allow control of their devices through OpenCL. However, the vendor-specific implementations are only designed to handle their own hardware devices, which leaves many interoperability possibilities of the OpenCL standard under-utilized. For example, when complex collaborative task pipelines are developed for heterogeneous systems, it is important that the devices are able to interoperate well in aspects such as data sharing and synchronization.

Field Programmable Gate Arrays (FPGA) can be *reconfigured* to implement any digital circuit. Reconfiguration makes it possible to change the functionality of FPGA devices after they have been manufactured. In terms of performance and energy efficiency, they are clearly behind

* Corresponding author.
E-mail address: topi.leppanen@tuni.fi (T. Leppänen).

https://doi.org/10.1016/j.micpro.2023.104772
Received 4 May 2022; Received in revised form 27 September 2022; Accepted 12 January 2023
Available online 16 January 2023

fixed-function accelerators due to the reconfigurability overhead [2]. Still, their fine-grained control over the datapath enables better efficiency than CPUs or GPUs in certain applications [3]. Traditionally, FPGAs are programmed with *hardware description languages* which describe the application functionality as timed digital circuits. A significant effort has been put forward to raise the level of abstraction towards high-level languages (e.g. C) to reduce application development effort for FPGA devices. Synthesizing digital circuits from high-level languages is called *High-Level Synthesis* (HLS).

Major FPGA providers have already included OpenCL as an input language for their HLS flows. While this enables rapid development of parallel accelerators, the development flows still require vendor-specific source code modifications [4]. Additionally, the generated accelerators are designed to be controlled by the runtime provided by the vendor and cannot efficiently work together with different devices in the same OpenCL context.

To solve the poor interoperability between vendor implementations we utilize *Portable Computing Language* (PoCL, [5]), which is an open source OpenCL implementation. It can be used to integrate very different device types such as CPUs, GPUs, FPGAs, and fixed-function accelerators into the same platform. This enables highly efficient collaborative execution platforms with various device types from multiple vendors.

This paper extends our previous work [6] in which we proposed a common hardware interface specification for OpenCL accelerators and the necessary software to integrate them into PoCL OpenCL runtime. The interface enables portable integration of programmable and non-programmable accelerators to a common OpenCL platform. The integration method enables the utilization of different FPGA types together with CPU devices. In this work we present the following additional contributions:

- Asynchronous (non-blocking) kernel execution enabled by on-chip synchronization and direct data sharing between kernels.
- Support for *direct memory access* (DMA) to external memory from the accelerators.
- An end-to-end audio *Convolutional Neural Network* (CNN) application case study starting from microphone input to a classification result.
- The first open source release of the proposed integration flow is made available when this article is published.[1]

The paper consists of the following sections. Section 2 includes background information on the related heterogeneous computing standards that this work is based on. Section 3 presents previous related integration methods and how the proposed method relates to them. Section 4 defines the hardware interface that provides a consistent memory-mapped view to various different types of integrated components. Section 5 describes the software developed in this work to connect the hardware interface to the OpenCL implementation. Section 6 shows practical instructions on how the integration method can be used to integrate new components. Section 7 includes overhead measurements and performance evaluations. Section 8 concludes the paper.

2. Open heterogeneous computing standards

OpenCL provides the lowest layer of heterogeneous computing software stacks. It has an extensive feature set to control diverse hardware devices with a common API. OpenCL has been widely adopted due to being an openly available standard.

The platform model of OpenCL consists of a single host CPU and one or multiple devices, which execute tasks specified by the host. The host uses *command queues* (CQs) to push work to the devices. These commands include *kernel commands* that launch a specific function on the device and *memory read/write commands*. OpenCL allows describing kernel commands as data-parallel *Single Program Multiple Data* (SPMD) functions, which perform the function for a multidimensional grid of kernel instances (*work-items*). The user provides a program description of what a single work-item is supposed to perform, and OpenCL applies that for multiple work-items. The work-items may then be executed sequentially or in parallel. The work-items can be grouped into work groups, which enables fine-grained synchronization between work-items of the same work group.

Task-level parallelism is described in OpenCL with command queues and *events* [7]. *In-order command queues* execute the commands in the order they are pushed to it, or if they are re-ordered by the implementation, the results must be the same as if they were not re-ordered. *Out-of-order command queues* can execute the commands in an arbitrary order by default. Explicit event synchronization is required when using out-of-order command queues or when multiple command queues, on possibly different devices, operate on shared data.

An event describes the completion status of a command. The user can describe dependencies between commands with events by providing a *wait-list* which is a list of events that must be completed before the command can be executed. This way the events can be used to construct command dependency task graphs. The in-order command queues automatically construct a task graph where every command depends on the previous one, whereas when using out-of-order command queues this graph must be explicitly defined by the user using events and event wait-lists. User-defined dependencies between two command queues connect the task graphs together.

OpenCL provides software portability for different heterogeneous systems by *online compilation*, which means that the kernel code is *(cross-)compiled* during run time just before it is needed. OpenCL also supports *custom devices* which do not have to support online compilation of kernel code but can execute a set of *built-in kernels* with semantics defined outside OpenCL. Built-in kernels can be used to expose fixed-function hardware or otherwise difficult to utilize hardware to OpenCL programmer.

Heterogeneous System Architecture (HSA, [8]) is another heterogeneous computing standard. It attempts to provide a way to map devices to a common unified coherent address space to enable simple data sharing between the devices. It introduces the idea of having devices implement well-defined memory-mapped interfaces for easier execution coordination. It has heavily inspired the proposed work, which does not attempt to implement the HSA specifications, but rather takes some parts which seem useful for the FPGA component integration methodology. The command queue packet format described later is almost directly as defined in the 1.2 of the Systems Architecture specification's *Architected Queue Language* (AQL) section, thus serving as a good source for further information.

3. Related work

Intel *Open Programmable Acceleration Engine* (OPAE, [9]) is an open source effort led by Intel to abstract the FPGA resources for Intel FPGA *Programmable Accelerator Cards* (PAC) with Xeon CPUs. They utilize a 256 KiB memory-mapped hardware interface called CCI-P in the component for control and status register usage. These can be used to control the *Application Functional Unit* (AFU). This is quite similar to our proposed method since we also define a hardware interface that the component must implement. Their implementation is limited to only Intel Xeon CPUs with PACs, whereas the proposed method is meant to be portable between various device types. However, the integration methodology proposed in this work could in the future utilize OPAE as a backend portability layer to support Intel FPGA PACs.

Xilinx provides an open source library called *Xilinx Runtime* (XRT, [10]) to provide similar management of their FPGA and ACAP devices. Their library supports both PCIe and *Multiprocessor System on a Chip* (MPSoC) type Xilinx devices. This method includes a kernel

[1] Source code available at: http://code.portablecl.org/

control interface in the integrated component to configure, launch and reset the accelerator on the FPGA. Configuration of kernel arguments is done through a slave control interface to a memory map specific to each kernel. In the proposed integration method, XRT's API is utilized to access the accelerator's memory map implemented on a Xilinx PCIe accelerator card.

Xilinx XRT and OPAE are both quite close to our integration method. The difference in the hardware interfaces is that our method shifts more functionality to the accelerator's side which makes it easier to decentralize the control of the accelerator. In both Intel's and Xilinx's approaches, the synchronization and data sharing between devices require more involvement from the host.

Furthermore, both Intel and Xilinx-provided OpenCL flows are still tied to their own FPGA devices and require the use of their software drivers to control the accelerators. Cross-vendor functionality has not been a priority for vendor OpenCL implementations, even though including multiple different types of devices in the same OpenCL context would enable efficient multi-command queue execution with event-based synchronization.

OpenCAPI [11] enables coherent host memory access to accelerator devices such as FPGAs and fixed function accelerators. The accelerators can use virtual pointers to user-space applications running on the host CPU. While their approach is architecture or vendor agnostic, hardware support is needed for both the host CPU and the FPGA accelerator card. The proposed method could in the future utilize OpenCAPI devices as one of the device platforms.

HOpenCL [12] is an OpenCL-inspired execution model which can be used to execute both software and hardware kernels on FPGA devices. While their intention is very similar to the proposed method, they do not stay within the OpenCL execution model, but instead develop their own. This significantly reduces the portability of applications written for their execution model. Our proposed method utilizes only standard OpenCL features with no requirements for extensions, which makes the applications developed for it portable to other OpenCL-supported platforms.

Steinert et al. [13] utilized FPGA for cloud acceleration. Their method uses a custom API to call the accelerator. Similar to the proposed work Holland et al. [14] develop a unified hardware interface they call USURP to control FPGA accelerators with MPI programming model. Another integration approach targeting MPI is Galapagos [15]. Our work focuses on the OpenCL standard which enables portability to a more diverse set of heterogeneous platforms.

Similar to the proposed method, Ashraf & Gioiosa [16] also use the OpenCL implementation PoCL to execute built-in kernels. They emulate the hardware devices as host CPU threads for a design space exploration use case. However, they do not attempt to integrate their components to an FPGA.

4. AlmaIF: The common hardware interface

Hardware accelerators execute tasks set by the host CPU. Therefore there must exist a well-defined way for the host CPU to communicate with the accelerator device to coordinate the execution of the OpenCL kernels. To accomplish this, the host CPU has a driver that manages the accelerator. Having to redesign or modify this driver for each possible hardware accelerator would be unreasonable, so there should exist a method for the driver to adapt to the device-specific details.

There are two clearly separate issues to solve when hardware devices are connected to the host CPU:

1. During initialization, the accelerator is *discovered*.
2. During run time, the accelerator is requested to execute tasks.

During initialization, the driver discovers information about the accelerator and updates its internal data structures to then be able to submit tasks to the accelerator device with minimal latency. Rediscovering the device every time a new task is launched would be wasteful, which is why the device initialization is performed only once at the beginning of the program.

The initialization of a new accelerator requires metadata of the accelerator parameters. This metadata can be passed as a separate metadata file which is created at the accelerator generation-time. Alternatively, some of this information can be embedded in the memory-mapped interface of the accelerator. The driver then either reads the metadata from the metadata file or from the interface at the device initialization time. If the metadata is embedded in the interface, accelerators could also discover other accelerators without having to have access to the host filesystem. In the future, this could be useful for e.g. hierarchical accelerators, where accelerator devices command each other.

During run time, the accelerator is controlled to launch the requested kernels at the specified time. If the accelerator can have multiple different functionalities, the driver has to be able to call the correct functionality. The control mechanism has to still be portable enough to not require significant modifications to the driver whenever new accelerators with new functionalities are designed.

The run time control and data movement responsibilities can also be split arbitrarily between the host and the accelerator. At one end, the accelerator can be highly autonomous and fetch its own data from the host memory. Alternatively, the host CPU can have very tight control of the accelerator to only launch it after making sure that the data it needs is fed into its ports. An example approach somewhere in the middle would be the host initiating the data move to a close-by memory of the accelerator before launching it. The advantage of autonomous accelerators is that they can operate asynchronously to the host CPU by handling the data movement themselves and synchronizing with each other to make progress.

A unified hardware interface with specifications for both initialization metadata and a protocol for run time control enables easy integration of new accelerators to the driver. With enough information in the physical interface itself, peer-to-peer discovery and control are possible. However, the interface must still be simple enough to be easily implementable in various existing and new accelerator designs.

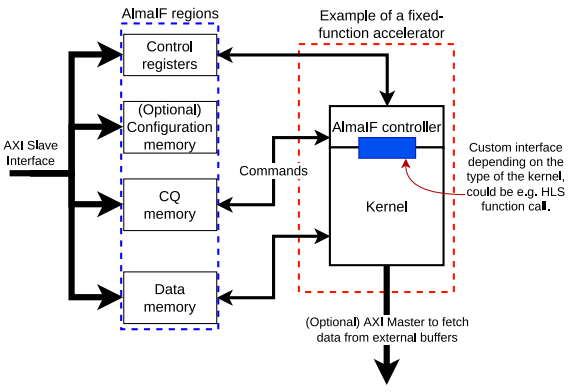
The proposed interface has been under active development under the name AlmaIF [17]. It is implemented as a memory-mapped hardware interface with four memory regions as shown in Fig. 1. Two example scenarios for integrating devices are shown: One for a fixed-function accelerator, and another for a software programmable core.

The four memory-mapped regions used in AlmaIF are the following:

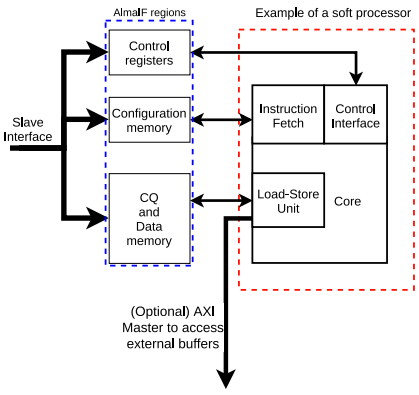
Control registers region is the first region and it includes both read-only registers for *automatic device discovery* and a control write-only register to reset and start the core. The control region defines the memory layout of the regions following it to easily enable different sized and packed memory regions. Therefore these values must always be read first before the other regions are accessed. The full register map of the control region is described in Table 1.

Configuration memory is an optional region designed mainly for programmable accelerators. It is defined to be a region where the host driver writes optional initialization bits specific to a kernel or a set of kernels. In the case of programmable accelerators, this region maps naturally to the instruction memory of the core, and the written bits are the program image that implements the kernel. Fixed-function accelerators can still utilize this region for target-specific purposes.

Command queue memory is used for run time control of the accelerator. It contains a command queue in which the host pushes kernel execution and synchronization command packets. The structure of the packets is based on the HSA AQL specification [8]. The kernel execution commands are called *kernel dispatch packets* which include information about the kernel functionality, work-item range, and argument addresses. The synchronization packets are described more in Section 5.4.



(a) Fixed-function accelerator.



(b) Software programmable core. The *configuration memory* is used to store the program image. In addition, this design utilizes the same physical memory component for mapping both the *command queue* and the *data memories*.

Fig. 1. Two different types of accelerators implementing the proposed hardware interface. AlmaIF memory regions highlighted with blue dashed line and the accelerator highlighted with red.

Table 1
AlmaIF: Memory mapped control registers.

Offset	Bits	Name	Purpose/explanation
0 × 000	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.
0 × 200	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).
0 × 300	32	Device class	Optional OpenCL vendor ID of the component.
0 × 304	32	Device ID	Currently unused by the driver.
0 × 308	32	AlmaIF version	Version number of the interface. Currently at value 2.
0 × 30C	32	Core count	Number of compute units in the device.
0 × 314	32	Configuration memory size.	Can be 0.
0 × 318	64	Configuration memory starting address	
0 × 320	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.
0 × 328	64	Command queue starting address	
0 × 330	64	Data memory size	
0 × 338	64	Data memory starting address	
0 × 340	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.

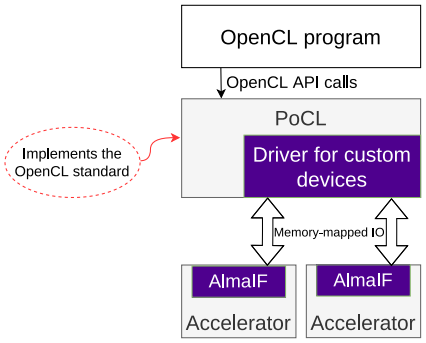


Fig. 2. Software stack of the integration method. The purple components correspond to the hardware and software interfaces proposed in this work.

The command queue is implemented as a ring buffer. Its read and write indexes are included in a header at the start of the command queue memory region. The device executes the command queue in order, updating the read index as it processes the command packets. Since the device executes the command packets in order, the barrier packets are sufficient to implement OpenCL event-based synchronization.

Data memory is a region close to the accelerator that can be used by the OpenCL runtime to allocate frequently accessed data. This can be e.g. OpenCL kernel arguments, small buffers or other OpenCL kernel command metadata (e.g. work-group sizes).

5. Software interface

Software interface for the proposed method is implemented as a user space driver in OpenCL implementation PoCL [5]. PoCL implementation handles the OpenCL API calls and forwards some of the calls and other tasks to the proposed driver. It is then the driver's responsibility to implement the device-specific functionality and manage the device to execute the OpenCL kernels. The common hardware interface definition makes it possible for the same driver to work with very different types of hardware accelerators. The proposed software interface is shown as a layer in Fig. 2 in between the generic PoCL implementation and the proposed hardware interface.

The proposed PoCL driver supports two types of accelerators:

1. Fixed-function accelerators, that are not software-programmable. They can only execute a set of built-in kernels chosen at design-time.

2. Programmable accelerators, which support online compilation of kernels from OpenCL standard specified inputs. Additionally, they can also implement built-in kernels.

The following two sections describe the support for these accelerator types in more detail.

5.1. Fixed-function accelerators

Fixed-function accelerators are highly specialized to perform only a predefined task or set of tasks. The functionality cannot be changed after fabrication and the accelerator cannot be programmed. The user must know before using such accelerators, which built-in kernels it can execute, and what are their semantics.

The OpenCL 3.0 specification does not define the semantics of built-in kernels. In OpenCL API calls, only the name (C string) of the kernel is used to identify it. Therefore, the naming system of the built-in kernels should be carefully designed to make the built-in kernels portable between different accelerator implementations. Otherwise, the user calling a poorly defined built-in kernel (e.g. "Convolution") might not know what kind of convolution the device is performing and what data types and layouts it is expecting.

To make the built-in names portable, the kernel semantics should be well-defined in a central registry. The registry could be a database with columns for kernel names and bit-exact definitions of the semantics and the data layout the kernel expects. One option could be to include an example OpenCL C source code implementing the built-in kernel functionality to serve as a reference for what the built-in kernel implements. The source code would also serve as a software fallback implementation for programmable devices that do not support the functionality as a built-in kernel.

If the device supports multiple built-in kernels at the same time, the software interface includes a 16-bit integer ID to the kernel execution command to denote which kernel it wants the device to execute. This is to avoid an expensive name string comparison in the device during run time. The integer ID is an index to the alphabetical list of the built-in kernels the device implements. Since the software interface knows all the built-in kernels the device can execute, it can order this list alphabetically and use the index as the ID.

5.2. Software programmable accelerators

The integration method also supports *online kernel compilation* as defined in the OpenCL standard. The method utilizes PoCL's LLVM-based [18] compilation passes to convert input program into executable binaries.

This integration method is most easily adapted for custom compilation flows that already utilize the LLVM compilation framework. Fig. 3 shows how new programmable accelerator compilation flows can be integrated into the proposed integration method. PoCL has support for generating the LLVM intermediate representation of the kernel for a single (SPMD-style) or multiple work-items (work-item loop) at a time.

In an ideal case, the target to be integrated already has an LLVM backend available, in which case the LLVM target triple is enough to compile code for it. However, if the target requires e.g. custom code generation passes, it is possible to add custom compilation callbacks that are called for the specific target. An example of this are devices based on OpenASIP [19], which need an architecture description file and custom compilation callbacks since the code generation is highly architecture-specific due to extreme instruction-set specialization. Currently, only the OpenASIP-based programmable accelerators are supported by the integration method. In the future, supporting more programmable accelerators is prioritized to ensure the generality of the method.

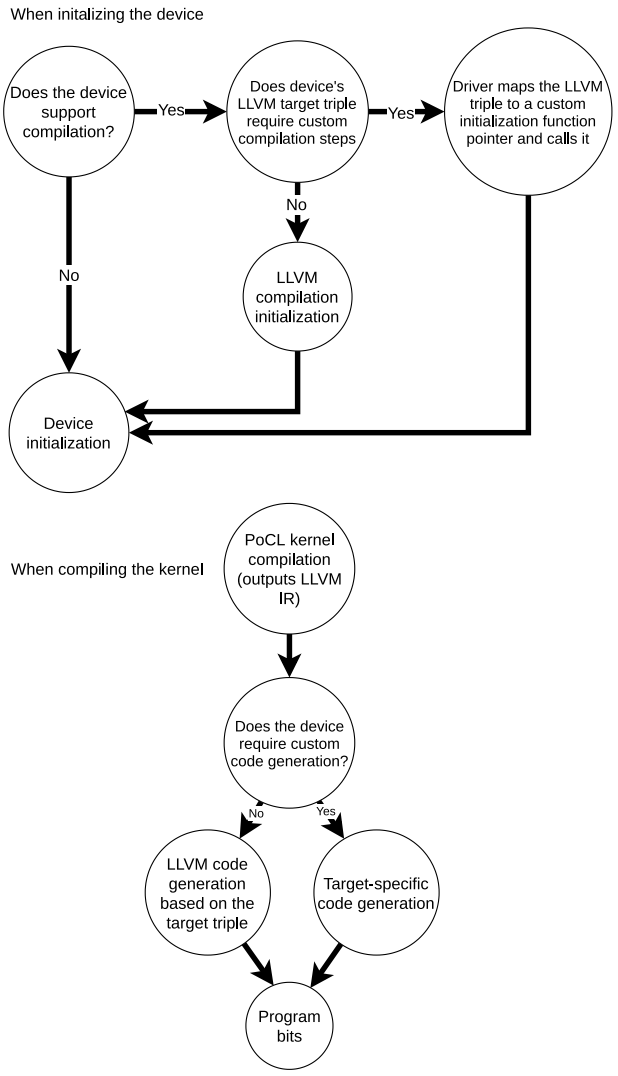


Fig. 3. Supporting kernel compilation based on either LLVM target triple or custom target-specific compilation flow.

5.3. Software programmable accelerators with built-in kernels

The proposed method supports accelerators that are programmable, but where the programmability is not possible to be exposed through the usual OpenCL kernel compilation flow. Built-in kernel abstraction enables the use of highly hand-optimized assembly or target-specific intrinsics since it is no longer restricted by the kernel language specifications. Creating efficient compilers for novel programmable accelerators is often difficult and can lag behind architecture development. However, the architecture can have highly efficient hand-written assembly programs for a limited set of tasks. The proposed method enables the use of such pre-made program binaries through the built-in kernel abstraction.

In this use case, the user must provide the *firmware* binary with the attached information of which built-in kernels the firmware implements. The integration method simply loads in the program firmware to the device *configuration region* at the device initialization time. In the future, this feature is to be extended to automatically load in the correct firmware from a set of user-provided firmwares based on the built-in kernels the user asks for in the OpenCL program.

5.4. On-chip synchronization

On-chip synchronization between AlmaIF devices enables *asynchronous command queue execution*. If the devices can coordinate with each other to respect dependencies between them, the host does not have to block on a kernel launch call even if the kernel depends on a previously launched non-finished kernel. This frees up the host to perform other tasks and can help ease system bus and memory hierarchy pressure by reducing the need for the host to poll for event completion.

The dependencies between kernels are expressed as HSA AQL-inspired *barrier packets* [8] in the device command queue region. A barrier packet consists of pointers to other signaling variables. Once the device comes across a barrier packet, it will check the values of the signaling variables and stall until all of the signaling variables are set to complete. The signal values can exist on data memories of different devices, which is why a unified memory address space is required. Although the synchronization could also be performed in a separate on-chip or external memory component, in the proposed approach we chose to utilize the AlmaIF data memory regions since unified access to those is already beneficial due to the data sharing opportunities.

The host will automatically create the barrier packet to the command queue whenever it would otherwise have to wait on a dependency between two kernels on different suitable devices. Currently, only the devices on the FPGA devices with AXI master support are suitable for on-chip synchronization. In the future, similar support could be added for CPU+FPGA synchronization, as long as reliable physical addresses of the CPU completion signals could be provided for the FPGA device. Dependencies between two kernels on the same device do not need barrier packets, since the device command queues are processed in-order.

Barrier-AND-packet includes pointers to 5 signaling variables. If there are more than 5 dependencies on a kernel, the host will write as many consecutive barrier packets as necessary.

After the host CPU has launched the kernels in a non-blocking way it must still be able to know once they have finished. There can be other dependencies to other devices or to the host program that are waiting for the non-blocking kernels to finish. To implement this, the host creates a single background thread that polls the kernels running on the FPGA devices for completion. This polling does not have to be as aggressive as would be the case if the host would have to resolve all the dependencies. It is not an issue if the other device on the FPGA has already processed the completion signal in question and started computing a dependent kernel. It is enough that the background thread is able to report back on finished kernels in a reasonable time for external synchronization purposes. This time depends on the latency requirements of the application and therefore in the proposed method, the polling frequency is an adjustable parameter.

Utilizing interrupts for device-to-host signaling would be another alternative to the current polling-based approach. Polling has been chosen due to its easy portability to new systems and the possibility for low latency synchronization due to the adjustable polling frequency. In the future, interrupts from device to host could be used as a wake-up signal for the host to go check on the devices it is controlling. This would remove the negative effects of polling on the memory hierarchy.

As shown in Fig. 4, moving some dependencies to be resolved at the device-side reduces the number of host synchronizations required. This has a chance to significantly reduce the execution time of programs with a large number of dependent kernel launches.

5.5. Direct memory access

Relying on the host to provide the data to the device's *data memory region* can be ineffective in applications with large amounts of data. Giving the device direct access to external memory can increase the performance significantly. This is due to the memory access not having to pass through more congested system memory interconnects. Additionally, the host CPU time spent copying data is away from performing other more useful tasks.

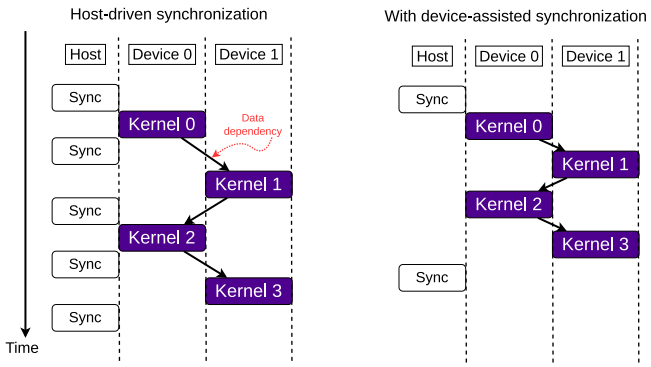


Fig. 4. Host synchronization differences with blocking and non-blocking kernel launches in case of dependent kernel launches.

Random access to an external memory from an FPGA programmable region can have significant overheads, even if it is using a separate interface to do it. High-performance FPGA-to-DDR interfaces are optimized for large burst transfers of contiguous data. Therefore, the device should also be able to perform burst transfers in order to get high throughput from its DMA operations. Since the latency of a single memory operation is high, the data should be *pre-fetched*, such that the device does not need to stall due to the memory access latency. In an ideal scenario, the data transfers are interleaved with the computation such that both the device and the memory subsystem can consistently perform at their maximum capacity.

When the device on the FPGA is accessing data that exists in host CPU's virtual address space, the virtual-to-physical conversion has to be handled carefully. If the FPGA device has access to a shared memory-management unit, it can perform the necessary address translations by itself. In the proposed method, we assume that such a component does not exist. This means that the host CPU must provide the device the *physical pointer* to the data buffer it has to access, and the data behind the pointer must be **physically contiguous**.

In the proposed method, the user must set an environment variable that describes a physically contiguous region's starting address and size. The provided software interface will then manage the buffers allocated in that region while performing the necessary physical-to-virtual and virtual-to-physical translations. The OpenCL application will then be able to transparently allocate OpenCL buffers into the region. If such a buffer is used as a kernel argument, the implementation will pass the physical pointer to the device. The device must be able to decode the buffer address to either access its own data memory region or the external physically contiguous region. This can be implemented as a hardware address decoder.

The built-in kernel abstraction allows the transparent use of FIFOs as shown in Fig. 5. The device itself can configure a DMA unit to transfer the data from external memory to an internal FIFO and use the data from there for the computations. Therefore, the built-in kernel abstraction allows hiding the DMA unit and its configuration inside the device itself. The host does not need to know whether the device is processing the data through a FIFO or not. The only limitation is the use of physically contiguous external memory, which must be enabled by the user by giving the contiguous region information to the implementation.

6. Component integration

The integration method provides a methodology and the necessary software tools to integrate custom FPGA components to an open

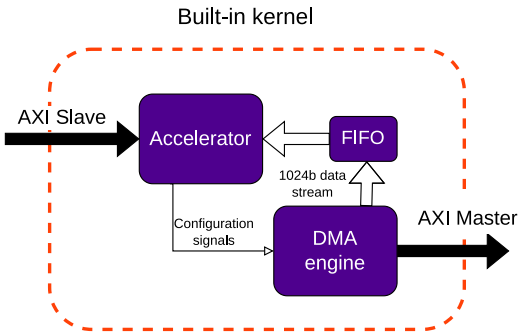


Fig. 5. Example design which demonstrates how the DMA engine can be hidden under the built-in kernel abstraction. The accelerator configures the DMA engine to initialize a data transfer to internal FIFO.

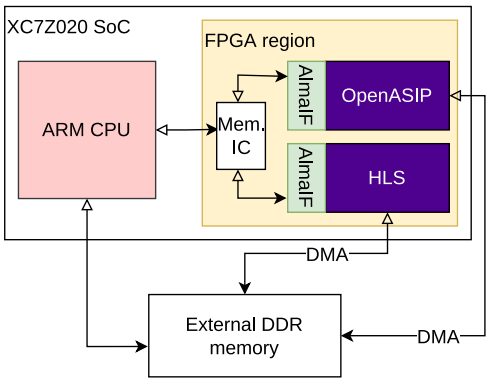


Fig. 6. XC7Z020 resource overhead and portability evaluation setup with two AlmaIF devices.

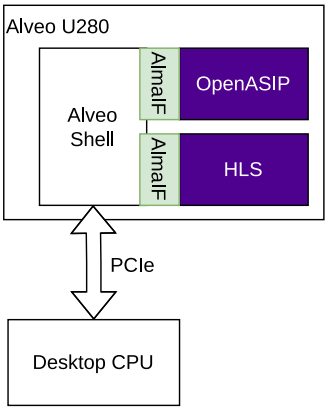


Fig. 7. Alveo U280 resource overhead and portability evaluation setup with two AlmaIF devices. DMA access to Alveo's DDR and *High Bandwidth Memories* (HBM) is not implemented.

and unified OpenCL platform. The practical steps required to use the integration methodology are summarized in this section.

(1) **Hardware component wrapping.** To add OpenCL support for a new or existing hardware component, it must be modified to implement the memory map described in Section 4. It must be able to process the command queue packets from the command queue region. Minimally, it must implement the processing of *kernel dispatch packets* for the built-in kernels it claims to support. Example implementation written in HLS-compatible C has been tested to pass through Vitis HLS [20] tool.

(2) *Optional step for completely new FPGA platforms:* **Enable access to the FPGA device's memory map in the host.** The provided software interface controls the hardware interface through a memory map. Therefore, there must exist a path from the host CPU to the device memory map on the FPGA. To make it easy to add new platforms, all the memory accesses from the host driver to the device are made through a C++ abstract class.

(3) *Optional step for new built-in kernel functionality:* **Register new built-in kernels.** If the accelerator to be integrated implements some previously unseen functionality, the new built-in kernel arguments must be registered to the software interface. The number and the types (buffers or scalars) of the arguments need to be specified in a configuration file of the driver. After that, the driver can check the validity of the OpenCL kernel arguments and can correctly fill the argument buffer before launching the kernel. The functionality of the built-in kernel does not need to be defined in the driver. However, if the built-in kernel is desired to be portable to different systems, its name and semantics should be published as described in Section 5.1.

(4) *Optional step for online compiler supported components:* **Add a compilation configuration entry.** To support online kernel compilation as described in Section 5.2, all the necessary information about the compilation flow must be specified to the driver. An example main program that implements the command queue processing and calls the kernel through a function pointer is provided. The integrator can configure whether the function executes a single work-item (SPMD-style) or multiple work-items in a loop. The driver then takes care to upload the program image to the device *configuration region* as described in Section 4.

7. Evaluation

The integration method is evaluated by creating two different types of accelerators using different development flows and evaluating both of them on two types of FPGA devices.

The first accelerator is a customized software programmable architecture generated with OpenASIP tools [19]. The second is created with Vitis HLS 2020.2 [20] commercial high-level synthesis tool utilizing their C language-based synthesis flow.

To demonstrate the portability of the method to different types of FPGA, both of the accelerators are implemented on an SoC-based FPGA (Xilinx Zynq XC7Z020) shown in Fig. 6 and a PCIe FPGA accelerator card (Xilinx Alveo U280) shown in Fig. 7.

Access methods to the FPGA device's memory map must be created as described in Section 6. First, on the SoC-based FPGA the physical memory of the accelerator's control interface is **mmap**ped with a Linux system call. Second, Xilinx PCIe accelerator card is accessed by utilizing the Xilinx-provided XRT library.

Both of the accelerators implement the same two built-in kernels: 32-bit element-wise addition and multiplication along the first work-item dimension. The kernels are purposefully kept simple to evaluate the overheads of the hardware interface.

Table 2
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%)	

Table 2 contains the resource utilization and the clock frequencies of the accelerators. The resource utilization as a ratio of total system resources is shown to be low enough to not be an issue even with multiple devices. The programmable OpenASIP accelerators have approximately three times the resource utilization of the HLS-generated one due to the added flexibility it has due to programmability.

7.1. Collaborative execution with CPU and FPGA devices

The integration methodology aims to provide flexibility in utilizing multiple different device types simultaneously. To validate this aspect, two demonstration cases are created with collaborative CPU+FPGA execution, one with the previously used Alveo PCIe card, and one with the previously used Xilinx SoC FPGA.

The demonstration cases first perform computation with the CPU device, after which the FPGA device continues operating on the output data of the CPU device. The implementation takes care to correctly migrate the buffers between the devices based on the event synchronization set up between the two command queues. These demonstrators prove that the collaborative execution with a CPU device works with both SoC and PCIe-based FPGAs.

7.2. On-chip synchronization

As discussed in Section 5.4, direct device-to-device synchronization enables faster program execution time, since the host CPU is not involved in between dependent kernel launches. The performance effect of this should be most visible with a large number of small kernels since those can be bottlenecked by the synchronization overhead.

For performance testing, two devices are implemented on the XC7Z020 FPGA running at 100 MHz. They both have their own separate AlmaIF interfaces. However, they also have their AXI master interfaces connected to each other's data memories, and their data address spaces are mapped to the same global address space, so they can transparently access each other's data memories using pointers. The evaluation setup is similar to the one shown in Fig. 8 with two devices and a host CPU connected together with a memory interconnect, except in this setup both of the devices are *Digital Signal Processor* (DSP) devices that implement only pocl.add.i32 kernel.

To measure the performance effect of on-chip synchronization, a synthetic benchmark is created. The benchmark program increments a variable in two devices one after another. The devices have access to each other's data memories, so the variable does not have to be moved in between kernels. The host thread launches a few hundred kernel iterations to two device command queues. Since there is no data movement between kernel launches, and the computation load is minimal, most of the total execution time will be spent on synchronization and kernel launch overheads.

The device-side synchronization can be disabled by having the host wait when it notices a dependency to a previously launched kernel. This is done to get the baseline performance numbers before the proposed method is applied. The average run-time of a single incrementation kernel is 313 μs.

With the proposed device-side synchronization enabled, the host will add a barrier packet as described in Section 5.4 whenever it notices a dependency to a previously launched kernel. After that, it will write the new kernel execution packet without waiting. After launching all of the kernels to both command queues, it waits for the final event in the second command queue before reading back the final result. With the proposed method, the average run-time is now brought down to 67 μs.

The synthetic benchmark shows that the average cost of the host-side synchronization compared to the on-chip synchronization is 246 μs. It can be approximated that even in other applications and systems a similar latency reduction can be expected at every dependent kernel launch. In a latency-critical application with a large number of small kernels, this can add up to significant overhead.

7.3. Direct memory access

The performance effect of DMA is measured on the XC7Z020 FPGA, which has high-performance master interfaces to external DDR memory. The host CPU can access the programmable region with a general-purpose slave interface, which is not optimized for performance. Therefore, moving the performance-critical transactions to high-performance interfaces should give significant throughput improvements.

The first benchmark used for the first five rows of Table 3 tests data movement overhead of relatively small arrays to find the break-even point where using DMA gives a performance improvement. In this benchmark, the DMA engine is configured in the user program to slightly reduce the hardware complexity needed to configure the DMA from the device side. The device is created with Vitis HLS tools as an HLS design with a streaming data port and an accompanying FIFO.

A clearly visible DMA startup latency can be seen in Table 3, which can make the DMA not as suitable for very small amounts of data in latency-critical applications. According to this benchmark, the break-even point where DMA and BRAM-based approaches reach the same performance is at a transfer size of about 4000 bytes. With arrays smaller than this, it makes sense to let the host perform the copy to on-chip BRAM, while with larger arrays setting up a DMA transfer is worth it.

Table 4 shows the resource utilization difference with the system including the DMA and the one without. The system with the DMA engine can use a smaller on-chip BRAM data memory, which reduces BRAM usage. The addition of the DMA engine and the FIFO clearly increases the LUT, LUTRAM and FF utilization, due to there being additional hardware components for performing the burst transfers and for FIFO storage of the pre-fetched data. However, the total resource usage is still kept reasonable. The differences in the data access method also affect how the final accelerator gets implemented, which causes the small difference in DSP utilization.

To reduce the effect of the DMA engine's configuration and kernel launch overhead, the DMA is also tested with a very large data array in an attempt to saturate the DMA bandwidth of the system. The benchmark loads in an $800 \times 600 \times 4$ image and performs a small pixel-wise operation on it. The total execution time is measured and compared against a device with no DMA support which relies on the host CPU to write the input data to the device's on-chip data memory region through an AXI slave port. The input image is so large that it does not fit into the BRAM at once. This means that it has to be split into smaller pieces, and the host must therefore synchronize the data movement with the accelerator which adds further overhead. With the DMA design, this split is not necessary, since the accelerator will pull the data in as it needs it, interleaving transfers with the computation.

In the saturating benchmark application in the last row of Table 3, the device on FPGA with DMA support configures its internal DMA IP to pull data from physically contiguous external memory to its internal FIFO with burst transfers through an AXI master interface. This is similar to the system shown in Fig. 5. Because the device utilizes a separate internal DMA engine for the transfer, it can start operating on the data in the FIFO as soon as it appears while the DMA transfer of the remaining data is still taking place.

In the test case with a large amount of data, the system with DMA is able to finish the application at best 8 times faster compared to the baseline case with the copy initiated by the host CPU. The exact speedup is always application and system-specific, but the principle of using a dedicated high-performance interface to get large throughput should be portable to various kinds of systems.

7.4. Audio CNN demonstrator

To demonstrate a whole application implemented with the proposed integration method, a case study is created. Additionally, the case study demonstrates the usage of a built-in kernel to abstract external IO

Table 3
DMA speedup with an increasing number of elements. BRAM column describes a device with no DMA engine and where the data is moved to on-chip BRAM by the host CPU. The column DMA describes a device with a DMA engine. The last column corresponds to the speedup in runtime the DMA device is used instead of the BRAM. The last row is a bandwidth saturation test performed with a slightly different device setup.

Elements (bytes)	BRAM (µs)	DMA (µs)	DMA vs BRAM Speedup
40	779	880	0.89
400	820	896	0.92
4000	948	920	1.03
8000	1047	960	1.09
16000	1279	1040	1.23
Bandwidth saturation test			
1920000	468500	58550	8.00

Table 4
FPGA resource utilization of the devices used for first five rows of Table 3. The device with the stream interface and the DMA engine does not need as large of a BRAM for the AlmaIF as the one without it. This is because the BRAM is not being used for the data movement.

Resource	Available	BRAM 128K AlmaIF Util.	BRAM 32K AlmaIF + Stream IF Util.
LUT	53200	1893 (4%)	10350 (19%)
LUTRAM	17400	254 (1%)	1723 (10%)
FF	106400	2126 (2%)	12125 (11%)
BRAM	140	32 (22%)	17 (12%)
DSP	220	16 (7%)	12 (5%)

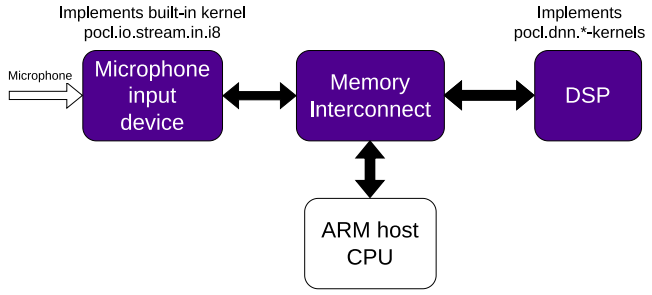


Fig. 8. Audio CNN demonstration setup which implements the kernels from Table 5.

signals. The system shown in Fig. 8 consists of a microphone input device and a DSP device. The microphone input device produces an input signal and writes it to on-chip memory. The DSP device executes a CNN inference on it and classifies the audio signal to belong to one of 10 classes. The system is deployed on a PYNQ-Z1 development board with the XC7Z020 FPGA. It executes in real-time, processing one 500 ms sample every half a second. The input is double-buffered so that the devices can work independently to produce a classification result every 500 ms.

The microphone input device samples the built-in microphone component of the development board. The built-in microphone produces *Pulse-Density Modulated* (PDM) signal, which is converted to *Pulse-Code Modulated* (PCM) signal utilizing hardware components to low-pass filter and decimate the signal. The final signal consists of 8-bit signed integer samples with a sampling rate of 16 kHz. The microphone input device writes the samples directly to the DSP device's data memory to one of the two buffers used in the double-buffering scheme.

The user commands the microphone input device using a built-in kernel. The built-in kernel *pocl.io.stream.in.i8* has a single argument called *output*, which is an OpenCL buffer that has been allocated on the DSP device. Once the microphone input device has produced 8000 samples, it sets the completion signal, which signals the DSP device to start processing the samples. After that, the microphone input device continues to write samples to the other input sample buffer.

Table 5
Audio CNN demonstrator built-in kernels.

Kernel name	Semantics
Microphone input device	
pocl.io.stream.in.i8	Produces n values from a streaming source to an output buffer.
DSP device	
pocl.add.i8	Element-wise addition for n elements.
pocl.dnn.conv2d.relu.i8	2D convolution and ReLU with parametrizable window size and stride for n input channels.
pocl.dnn.dense.relu.i8	Dense layer with ReLU activation.
pocl.maxpool.i8	2D maxpool with parametrizable window size and stride.

The DSP device executes the classification CNN to produce the class label for the 500 ms input sample. The CNN is a quantized version of a 1D network introduced in [21]. The numerical values of the CNN are quantized to 8-bit to take advantage of the specialized 8-bit vector hardware present in the DSP accelerator.

The DSP device is a soft processor created with OpenASIP toolset [19] and specialized especially for 8-bit convolution computation. The processor RTL is designed and optimized for ASIC implementation. Synthesizing it to FPGA directly without performing any FPGA optimizations is a way to prototype and evaluate the processor with realistic workloads. The relatively slow clock frequency 43 MHz of the non-FPGA-optimized design is not an issue in this case, as the 500 ms constraint to execute the inference is long enough that the entire CNN inference can be executed. Faster FPGA-optimized accelerator could provide better result quality since the extra computation budget could be used to scale up the neural network.

The DSP device is able to execute the four built-in kernels described in Table 5. The host program calls the kernels one after another to execute the quantized network layer by layer. The kernels are implemented in the device firmware. The built-in kernel abstraction allows it to be manually optimized and to utilize DSP-specific intrinsics to achieve higher performance. The same firmware contains implementations for all of the kernels, so it is not changed after it has been loaded in at initialization time.

Due to the limited on-chip memory (64 KiB) of the DSP-core, the entire network weights and intermediate values cannot fit in it at the same time. However, for performance reasons, it is a good idea to have the values close to the core when they are being used for computation. Therefore, the host also takes care to allocate and deallocate the OpenCL buffers in such a way that the device memory does not run out. This is done by the user in the OpenCL program by freeing the buffers after the layer has been executed and they are no longer needed. This deallocates the buffer so that the memory can be used for the buffers of the next layer.

The utilization breakdown of the case study can be seen in Table 6. The DSP device is running at 43 MHz. The relatively low clock frequency is due to it being a prototype of an ASIC-optimized processor. The AXI interconnect that is used to connect devices to each other and to the host CPUs interconnect takes up most of the area in this case. This shows that the devices themselves consume relatively small amounts of area. The interconnect has to take care of connecting every device to every other device, and perform clock domain crossings when necessary which makes it quite a complicated block.

8. Conclusions

The integration method of fixed-function and programmable accelerators to a unified OpenCL platform was presented. The proposed method makes it possible to integrate FPGA components to OpenCL

Table 6
FPGA case study resource utilization.

	LUTs	Registers	BRAMs (4kB)	FMax (MHz)
Microphone input device	1514	2739	8.5	96
DSP	6497	4517	32	43
Interconnect	8905	11003	0	43,96
Total	16916	18259	40.5	

platforms without relying on vendor-specific OpenCL implementations. The previous work [6] was extended to add support for direct memory access to external memory which was shown to increase the performance by a factor of 8. Another extension to previous work was the non-blocking kernel execution enabled by on-chip synchronization, which removed 250 µs host synchronization overhead when dependent kernels were launched. Additionally, a deployment-ready audio-processing demonstrator was created to prove that the proposed method can be used to build complete working systems with external IO wrapped inside the built-in kernel abstraction.

The current version of the method supports streaming data transfers for built-in kernels. In the future, the work will be expanded to support OpenCL pipes for more efficient on-chip streaming for software kernels. Additionally, automatic bitstream and firmware deployment based on the submitted built-in kernels will be created to further increase the usability of the proposed method. Evaluating the method on a wider range of FPGA devices from different major vendors could expose further portability improvements. Furthermore, supporting OpenCL 2.0 *Shared Virtual Memory* (SVM) would be a very useful addition for platforms with the necessary hardware support for advanced memory management.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code repository used for this research is included as a footnote.

Acknowledgments

The work for this publication was 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, 2021, https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, accessed: 2021-08-27.

[2] I. Kuon, J. Rose, Measuring the gap between FPGAs and ASICs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 26 (2) (2007) 203–215, http://dx.doi.org/10.1109/TCAD.2006.884574.

[3] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, S. Zhang, Understanding performance differences of FPGAs and GPUs, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, 2018, pp. 93–96, http://dx.doi.org/10.1109/FCCM.2018.00023.

[4] S. Lahti, P. Sjövall, J. Vanne, T.D. Hämäläinen, Are we there yet? A study on the state of high-level synthesis, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 38 (5) (2019) 898–911.

[5] P. Jääskeläinen, C. Sanchez de La Lama, E. Schnetter, K. Raiskila, J. Takala, H. Berg, Pocl: A performance-portable OpenCL implementation, Int. J. Parallel Program. 43 (5) (2015) 752–785.

[6] T. Leppänen, P. Mousouliotis, G. Keramidas, J. Multanen, P. Jääskeläinen, Unified OpenCL integration methodology for FPGA designs, in: 2021 IEEE Nordic Circuits and Systems Conference (NorCAS), 2021, pp. 1–7, http://dx.doi.org/10.1109/NorCAS53631.2021.9599861.

[7] P. Jääskeläinen, V. Korhonen, M. Koskela, J. Takala, K. Egiazarian, A. Danielyan, C. Cruz, J. Price, S. Mcintosh-Smith, Exploiting task parallelism with OpenCL: A case study, J. Signal Process. Syst. 91 (1) (2019) 33–46, http://dx.doi.org/10.1007/s11265-018-1416-1.

[8] HSA™ Foundation, HSA Platform System Architecture Specification v1.2.

[9] E. Luebbers, S. Liu, M. Chu, Simplify software integration for FPGA accelerator with OPAE (white paper), 2017, URL https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf.

[10] Xilinx, Xilinx Runtime (XRT) Architecture, URL https://xilinx.github.io/XRT/master/html/index.html.

[11] J. Stuecheli, W.J. Starke, J.D. Irish, L.B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, B. Allison, IBM POWER9 opens up a new era of acceleration enablement: Opencapi, IBM J. Res. Dev. 62 (4/5) (2018) 8:1–8:8, http://dx.doi.org/10.1147/JRD.2018.2856978.

[12] H. Ding, 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, http://dx.doi.org/10.1109/ReConFig.2014.7032563.

[13] F. Steinert, P. Kreowsky, E.L. Wisotzky, C. Unger, B. Stabernack, A hardware/software framework for the integration of FPGA-based accelerators into cloud computing infrastructures, in: IEEE International Conference on Smart Cloud (SmartCloud), 2020, pp. 23–28, http://dx.doi.org/10.1109/SmartCloud49737.2020.00014.

[14] B. Holland, J. Greco, I.A. Troxel, G. Barfield, V. Aggarwal, A.D. George, Compile- and run-time services for distributed hetergeneous reconfigurable computing, in: ERSA, 2006.

[15] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, P. Chow, Galapagos: A full stack approach to FPGA integration in the cloud, IEEE Micro 38 (6) (2018) 18–24, http://dx.doi.org/10.1109/MM.2018.2877290.

[16] R.A. Ashraf, R. Gioiosa, Exploring the use of novel spatial accelerators in scientific applications, in: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, ICPE '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 47–58, http://dx.doi.org/10.1145/3489525.3511690.

[17] J. Hoozemans, J. van Straten, T. Viitanen, A. Tervo, J. Kadlec, Z. Al-Ars, ALMARVI execution platform: Heterogeneous video processing SoC platform on FPGA, J. Signal Processing Systems 91 (1) (2019) 61–73, http://dx.doi.org/10.1007/s11265-018-1424-1.

[18] C. Lattner, 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, 2004.

[19] P. Jääskeläinen, T. Viitanen, J. Takala, H. Berg, HW/SW co-design toolset for customization of exposed datapath processors, in: W. Hussain, J. Nurmi, J. Isoaho, F. Garzia (Eds.), Computing Platforms for Software-Defined Radio, Springer International Publishing, 2017, pp. 147–164.

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

[21] S. Abdoli, P. Cardinal, A. Lameiras Koerich, End-to-end environmental sound classification using a 1D convolutional neural network, Expert Syst. Appl. 136 (2019) 252–263, http://dx.doi.org/10.1016/j.eswa.2019.06.040.

Topi Leppänen reveived the M.Sc. degree in Electrical Engineering in 2021 from Tampere University, Finland, where he is currently pursuing a Ph.D. degree. His research interests include heterogeneous platforms and hardware acceleration. His current research focuses on easier development and programming of diverse systems with specialized programmable and nonprogrammable accelerators.

Atro Lotvonen MSc. Information Technology and Communication Sciences, Graduated from Tampere University in 2021 with a major in Signal Processing and Machine Learning. Previously worked on the CPC-VGA group in Tampere University on topics related to real-time computer graphics and virtual reality algorithms. More recently, work is focused on executing machine learning models on heterogeneous platforms.

Panagiotis Mousouliotis graduated from the ECE department of Aristotle University of Thessaloniki (AUTh) in 2011 and since then he has worked in projects ranging from digital logic design on FPGAs to C/C ++ application development for embedded systems. Currently, he is a Ph.D. student in the ECE AUTh department working on algorithm acceleration in FPGAs using HLS tools.

Joonas Multanen received his M.Sc. degree in Electrical Engineering in 2015 from Tampere University of Technology and his Ph.D. degree in 2021 from Tampere University (TAU), Finland. He is currently a postdoctoral researcher at the Faculty of Information Technology and Communication Sciences in TAU. His research interests include energy efficient computer architectures.

Georgios Keramidas Assistant Professor at the School of Informatics of the Aristotle University of Thessaloniki, Greece and a technology consultant in Think Silicon S.A. Dr. Keramidas main research interests are in the areas of low-power processor/memory design, multicore systems, VLIW/multi-threaded architectures, graphic processors, power modeling methodologies, FPGA prototyping, and compiler optimizations techniques. He has published more than 75 papers, two books, and he also holds 13 US patents (4 more patents are under evaluation). His work received more than 1190 citations. He is a regular reviewer and program committee member in high quality conferences, workshops and transactions and a member of the HiPEAC European Network of Excellence.

Pekka Jääskeläinen Leads the Customized Parallel Computing group (http://cpc.cs.tut.fi) of Tampere University. He has worked on heterogeneous platform customization and programming topics since early 2000s. In addition to his academic publication activities, he is responsible of two heterogeneous computing related open source projects; OpenASIP (http://openasip.org) and Portable Computing Language (PoCL, http://portablecl.org). His research interests include methods and tools to reduce the engineering effort involved in design and programming of diverse heterogeneous platforms, and hardware and compiler techniques to reduce the energy consumption of programmable processors.