frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 20 October 2022
DOI 10.3389/fcomp.2022.945652

Cross-vendor programming abstraction for diverse heterogeneous platforms

Topi Leppänen, Atro Lotvonen and Pekka Jääskeläinen*

Customized Parallel Computing Group, Tampere University, Tampere, Finland

Hardware specialization is a well-known means to significantly improve the performance and energy efficiency of various application domains. Modern computing systems consist of multiple specialized processing devices which need to collaborate with each other to execute common tasks. New heterogeneous programming abstractions have been created to program heterogeneous systems. Even though many of these abstractions are open vendor-independent standards, cross-vendor interoperability between different implementations is limited since the vendors typically do not have commercial motivations to invest in it. Therefore, getting good performance from vendor-independent heterogeneous programming standards has proven difficult for systems with multiple different device types. In order to help unify the field of heterogeneous programming APIs for platforms with hardware accelerators from multiple vendors, we propose a new software abstraction for hardware-accelerated tasks based on the open OpenCL programming standard. In the proposed abstraction, we rely on the built-in kernel feature of the OpenCL specification to define a portability layer that stores enough information for automated accelerator utilization. This enables the portability of high-level applications to a diverse set of accelerator devices with minimal programmer effort. The abstraction enables a layered software architecture that provides for an efficient combination of application phases to a single asynchronous application description from multiple domain-specific input languages. As proofs of the abstraction layer serving its purpose for the layers above and below it, we show how a domain-specific input description ONNX can be implemented on top of this portability abstraction, and how it also allows driving fixed function and FPGA-based hardware accelerators below in the hardware-specific backend. We also provide an example implementation of the abstraction to show that the abstraction layer does not seem to incur significant execution time overhead.

KEYWORDS

heterogeneous computing, heterogeneous programming, OpenCL, hardware abstraction layer, hardware acceleration, built-in kernel

OPEN ACCESS

EDITED BY
Tobias Grosser,
University of Edinburgh,
United Kingdom

REVIEWED BY
Hiroyuki Tomiyama,
Ritsumeikan University, Japan
Luca Muratore,
Italian Institute of Technology (IIT), Italy

*CORRESPONDENCE
Pekka Jääskeläinen
pekka.jaaskelainen@tuni.fi

SPECIALTY SECTION
This article was submitted to Software,
a section of the journal
Frontiers in Computer Science

RECEIVED 16 May 2022
ACCEPTED 05 October 2022
PUBLISHED 20 October 2022

CITATION
Leppänen T, Lotvonen A and Jääskeläinen P (2022) Cross-vendor programming abstraction for diverse heterogeneous platforms.
Front. Comput. Sci. 4:945652.
doi: 10.3389/fcomp.2022.945652

1. Introduction

Specialization is a powerful paradigm that has emerged in mainstream computing in the past decades. The benefits of domain or application specialized computing devices are usually motivated by higher performance. Also, its potential to enhance power-efficiency has received increased interest outside the usual power-supply constrained IoT devices.

Steep competition between computing platform vendors has led to major performance and power efficiency advances, which have resulted in more computing power for the end-users for the same price. However, since the competition tends to steer the programmability focus toward only the vendor's own devices instead of embracing multi-vendor performance portability, software abstractions based on open widely adopted standards have stagnated in the progress. Since there is no widely-adopted cross-vendor abstraction for interfacing with hardware accelerators, cleanly layered implementation of domain-specific programming models is not possible. This is visible in the upper layers of the software stacks needing to interface with multiple different vendor-specific platform layers such as the Level Zero layer of the Intel-driven oneAPI (Intel, 2021) or the AMD's ROCR runtime of ROCm (AMD, 2022), and the NVIDIA-specified CUDA (Luebke, 2008).

The problem is not only a matter of software architectural aesthetics or complexity: In heterogeneous systems, it is beneficial that the devices are able to interoperate to most efficiently coordinate the execution of various tasks of an application. Vendor-provided implementations for their own programming models, or even for open standardized programming models, cannot directly communicate and synchronize with devices from different vendors due to the different software stacks and device control protocols. Furthermore, utilizing fixed-function or FPGA hardware in a portable way as part of heterogeneous computing systems is still limited. While two of the major FPGA vendors Intel (Altera) and AMD (Xilinx) provide their own OpenCL flows, they still require vendor-specific pragmas for best performance and have very limited interoperability between different device types, let alone across vendors.

This work proposes a software abstraction for hardware-accelerated tasks, which is based on an open heterogeneous programming standard OpenCL (Khronos® OpenCL Working Group, 2020). The abstraction relies on the built-in kernel abstraction specified in the standard. The abstraction can be used to launch implementation-defined functionality on programmable or fixed-function hardware devices. We identify a key missing aspect of this part of the standard is that it does not require the specification of the built-in kernel semantics and lacks a centralized open registry, which is necessary for their re-implementation by multiple vendors. To this end, we propose a concept called *built-in kernel registry* which is a library of tasks that are beneficially hardware accelerated by at least one vendor's device. The tasks have their calling conventions and semantics well-defined, enabling portability for both higher and lower levels of the software stack. Since the higher-level software tools can rely the semantics of the built-in kernels, it enables them to choose the optimal set of built-in kernels for a certain task. Since the built-in kernels are well defined in the centralized registry, it forms a low-level API for accelerated functionality, creating a layer for porting performance-critical applications on.

Utilizing the built-in kernel registry, the lower level of the software and hardware tools can take the built-in kernel semantics as functional specification and implement the same functionality on various existing and new hardware devices by different vendors. As an example, the built-in kernels can be hand-optimized and synthesized to FPGA bitstreams, distributed as a shared library of built-in kernel implementations, and loaded in when they are needed. This enables more flexible use of FPGA devices in heterogeneous systems since the separate offline synthesis needed in OpenCL C-defined kernels is removed.

The proposed abstraction layer makes it possible to describe asynchronous heterogeneous task graphs across devices from different vendors. To this end, we implement the abstraction in an open-source vendor-independent OpenCL implementation which can utilize devices from multiple vendors concurrently in the same OpenCL context. This can lead to performance improvements since the single common OpenCL implementation is able to optimize the data movement and the synchronization between the devices.

We will exemplify the concept using the popular ONNX as the input "domain-specific language" which interfaces with the programming abstraction without requiring other vendor-specific APIs. A proof-of-concept tool flow from ONNX-input to execution on CPU, GPU, and FPGA devices is demonstrated. We also include a method of combining deep learning inference computations with pre-and post-processing kernels.

To summarize, the contributions of this work are:

- Specification of a global registry of OpenCL built-in kernels to abstract hardware-accelerated functions.
- Integration of the built-in kernel registry abstraction to an open-source multi-vendor OpenCL implementation.
- Proof-of-concept built-in kernel implementations for CPU and GPU to show kernel portability and ability to utilize existing acceleration hardware.
- Proof-of-concept built-in kernel implementations for an FPGA device to show rapid utilization of the FPGA device and ability to abstract custom IO devices.

An example implementation of the abstraction layer is published as open-source[1] to create a common collaboration layer for both software and hardware developers. The proposed method increases the flexibility of programming heterogeneous systems while remaining performance portable to various different systems. This has the potential to significantly benefit the heterogeneous platform programming community by enabling wider options for executing their workloads on a diverse set of heterogeneous devices with minimal software changes.

1 The source code is available at: http://portablecl.org/.

2. Related work

2.1. Portability abstractions for hardware accelerators

OneAPI by Intel (2021) abstracts accelerator architectures behind either its custom *API programming* model, which consists of calls to optimized library implementations, or a *direct programming* model, where the algorithm is defined by the programmer and compiled for the hardware accelerator. In theory, the OneAPI platform can be ported to new hardware accelerators by porting the *OneAPI Level Zero*-layer of the specification. OneAPI is very close in intention to the proposed abstraction layer since the API programming model can roughly be mapped to the OpenCL built-in kernels and *direct programming* to compiled OpenCL kernels. Different to the proposed work, they do not include an extendable centralized registry of kernel specifications, but instead opt for specifying the acceleration API in a specification document. Additionally, instead of relying on a single vendor-specified hardware layer (Level Zero), the proposed abstraction is built on a open standard which is hoped to stimulate cross-vendor interoperability. Finally, the proposed method utilizes a more mature standard, OpenCL, whereas the OneAPI is a much newer specification, future of which is still directed by the commercial interests of a single company, Intel.

MLIR by Lattner et al. (2021) approaches the same problem of enabling the execution of high-level programs on various hardware platforms from a compilation point of view. With their infrastructure, the programs are progressively lowered and optimized to different *intermediate representations* to gradually approach the actual hardware the programs will execute on. New *dialects* for the intermediate representations can be created for specific purposes, which include e.g. accelerator-specific operations. While MLIR by itself has quite different intentions from the proposed method, specific dialects can be described with it that implement certain fixed functionality, similar to the proposed built-in kernels-based approach. These dialects can then have lowering passes to different backends, which execute the functionality. One example of such dialect is *Tensor Operator Set Architecture* (TOSA) by Linaro (2021) which defines a set of common operators used in deep neural networks. The operator definitions are designed to be generic enough such that they are able to be further lowered to many different types of hardware platforms. MLIR and TOSA can be seen as complementary to the proposed abstraction, a means to lower higher-level task graphs and optimize them whereas in our case we provide a mechanism on top of a portable heterogeneous platform abstraction for execution time definition of the whole multi-kernel application.

Another work (Katel et al., 2022) is able to use MLIR to utilize Nvidia's tensor cores, which are coarse-grained accelerator hardware components for accelerating tensor computation. While their method makes it possible to fuse other operations to the same program as the one using tensor cores, their method requires reconstructing the program to fit the execution model of the hardware accelerator. The proposed method enables easier utilization of coarse-grained hardware by constructing built-in kernels that already match closely the vendor-provided acceleration libraries utilizing the coarse-grained accelerator.

OpenVX by Khronos (2022) is a standard for the acceleration of computer vision applications. It is used to describe applications as graphs where nodes implement specific functionality and edges describe dependencies between them. The standard specifies the semantics for a large number of computer vision-related kernels, which the implementation can then implement with programmable or fixed-function hardware. OpenVX's registry of functions is very close to the proposed method's registry of built-in kernels. Instead of focusing on a single application domain, however, our proposed abstraction aims for cross-domain portability in order to be able to describe heterogeneous task graphs with tasks that span the whole application which is potentially defined using multiple domain-specific languages and APIs.

Intel's OpenCL extension *cl_intel_accelerator* made available by Galoppo et al. (2016) enables the control of custom accelerator devices with new extended OpenCL API calls. The extension adds methods to manage the lifetime of accelerators. The extension is meant to be further extended by new accelerator-specific extensions to implement specific functionality, so therefore they do not attempt to describe the semantics of the accelerators. Their method can be used to control hardware accelerators with built-in kernels, similar to our method. Our method utilizes the OpenCL standard built-in kernels while introducing a way of defining the built-in kernel semantics through a centralized registry, without requiring custom OpenCL vendor extensions for each new accelerator type.

Another related work adds support for machine learning acceleration at the operation level with the OpenCL extension *cl_qcom_ml_ops* (Reddy et al., 2022). Their method enables the utilization of the OpenCL runtime and custom kernels seamlessly with their proposed DNN API. However, their method requires that the hardware supports the extension with new API calls. The proposed method is more generic to also other computation than machine learning operations and does not need to extend the OpenCL API, which increases the application portability.

Henry et al. (2014) propose an OpenCL implementation SOCL that wraps other vendor-specific OpenCL implementations and then executes OpenCL programs on them with their dynamic load balancing scheduler. Since their approach works at the level of OpenCL implementations, they are limited in their ability to perform cross-vendor operations between the implementations without extending the OpenCL standard, and therefore changing the OpenCL API calls used in the host program. The proposed method wraps vendor devices at a lower level, which enables more direct possibilities for inter-vendor functionality while remaining OpenCL compliant for maximum application portability.

2.2. Deep learning compilers

One commonly accelerated application domain is deep learning (DL). Deep learning uses powerful function approximators that are able to generate models which surpass most of the handcrafted models and even human performance (He et al., 2015). The problem with these models is that they require a lot data for training and computation power for running them on different hardware. Moreover, these models contain different kinds of layers such as convolutional layers in convolutional neural networks (CNN) (LeCun et al., 1989) and long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) which require specialized computational implementations and hardware for efficient execution of the models on the hardware.

With the growing popularity of DL, multiple frameworks have been proposed for defining and training these models such as TensorFlow (Abadi et al., 2015), PyTorch (Paszke et al., 2019), and MXNet (Chen et al., 2015). The parallel development of these frameworks has led to the creation of different model formats with similar layers and ideas. Recently, more unified model formats have been proposed to help with the interoperability between the frameworks such as ONNX (Bai et al., 2019). These frameworks help developers with the training process but do not generally have optimized implementations for deploying these models for different kinds of hardware. For the deployment of the DL models several DL compilers have been proposed such as Apache TVM (Chen et al., 2018) which continuously outperforms the computational models generated by the DL frameworks and also targets a diverse set of hardware. A thorough survey of the deep learning compilers is done by Li et al. (2021).

The different kinds of computations inside DL models have inspired the generation of multiple specialized accelerators from various hardware providers. To utilize these different kinds of accelerators the hardware vendors usually offer optimized computation libraries to transform DL training and inference algorithms to efficient implementations such as cuDNN (Chetlur et al., 2014) or oneDNN which is part of the oneAPI specification (Intel, 2021). Additionally, to utilize these libraries on model and graph levels the vendors offer solutions such as TensorRT.

Frameworks such as Apache TVM attempt to support multiple of these hardware devices as backends. The fundamental issue TVM attempts to solve is to remain accelerator agnostic while generating efficient computation code for multiple hardware devices. To solve this, it implements multiple different language backends such as, LLVM, C, OpenCL, and CUDA or it explicitly generates calls to external acceleration libraries. It is also possible to extend TVM backends by adding new custom code generation for new computing hardware.

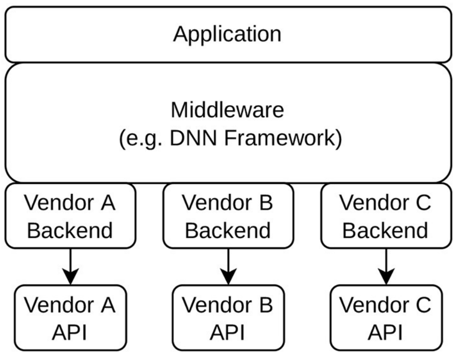

FIGURE 1
A typical software stack of an acceleration framework with multiple vendor-specific backend implementations.

While these external solutions work for different accelerators, due to a lack of an unifying cross-vendor abstraction, they require different backend implementations for each device. As is illustrated in Figure 1, this means there is duplicated development effort in the creation of multiple separate backends for each vendor. Separate backends can also create inefficiencies in systems that include multiple different accelerators. In the proposed work, these accelerators are exposed for the code generation through a unified standard interface specified by the OpenCL built-in kernel abstraction. This also enables more efficient execution of the entire task graph under a single hardware abstraction layer implementation.

3. Diverse heterogeneous platform software layer

Open Computing Language (OpenCL) is a heterogeneous platform programming standard specified by the Khronos group. It was first introduced in 2009, and achieved some initial popularity across multiple vendors until the 1.2 version, but suffered a decline in adoption rate after its 2.0 version. It is widely believed the vendor adoption rate was adversely impacted by a high number of new mandatory features introduced in 2.0. However, in version 3.0 most of the features were made optional, which has again made the standard more popular as a general-purpose computing portability layer for heterogeneous platforms.

One of the features of OpenCL is its inherent capability to define heterogeneous task graphs that can be executed across different devices supported by the OpenCL platform implementation. There is no explicit concept of a task graph in the standard, but instead the runtime can construct the graphs implicitly: The devices are controlled using a concept of a command queue, which supports event synchronization between the kernels pushed to the devices through them. The concepts of a kernel (a function executed on a device), a command queue (a way to push work to a device), and an event (signaled by a completion of a command that can be waited on) together can be used to form complex parallel heterogeneous computing across a diverse heterogeneous platform with multiple different device types (Jääskeläinen et al., 2019).

Hahnfeld et al. (2018) show that to get significant speedups from executing heterogeneous task graphs, the devices must be able to interoperate efficiently to interleave the communication with the computation. We believe this aspect, however, is still rather underused and OpenCL is utilized like other offloading APIs such as CUDA to mainly let CPUs launch single isolated throughput-oriented kernels on GPUs.

In this work, we rely on the concept of heterogeneous task graphs and build on the open standard OpenCL's capabilities to define them. Another powerful aspect of OpenCL is its theoretical capability to control all kinds of compute devices such as CPUs, GPUs, and DSPs through the common API. A modern class of compute devices are coarse-grained hardware accelerators specialized for neural network acceleration. For a proper hardware abstraction layer API, hardware accelerators should be naturally supported to integrate them to heterogeneous task graphs along with kernels launched to programmable devices. The main benefits of such a programming model are clear: It would allow asynchronous execution across the accelerator devices without requiring back-and-forth synchronizations and data transfers with the CPU that orchestrates the application execution.

A further benefit is the fundamental engineering gains of a layered software architecture where the lowest layer is for portability through a low-level API that can be used to control all kinds of compute devices in an efficient and interoperable manner. This layer is then used by the implementations of the higher programming abstraction levels in order to maximize the application engineering productivity. This type of layered software architecture with OpenCL providing the portability is illustrated in Figure 2. The common abstraction layer can coordinate the multiple device types at a very low level, which creates the possibility of highly efficient inter-device functionality as exemplified in the figure.

A new feature for abstracting coarse-grained hardware accelerators was introduced to OpenCL in version 1.2 of the standard. The abstraction is called a *built-in kernel* and it continues to be available in OpenCL version 3.0. Built-in kernels provide a name-based handle to a functionality implemented by an accelerator device which can be then launched similarly to software-defined kernels by pushing their invocations to a command queue which can then contain a mix of built-in or software-defined kernels. This abstraction, while rather simple, can be very powerful thanks to its seamless integration to the heterogeneous task graphs that can be built from command queues. However, the adoption of the built-in kernel API is minimal in vendor implementations, and we believe a reason for that is that it misses a small but crucial concept of well-defined semantics for the kernels. In its current form, the accelerator devices can advertise to support a set of built-in kernels with driver-chosen name strings that might be meaningful or not. This means that the semantics of the kernels are vendor-specific and thus suffer from the portability problem: A kernel with the same name can map to a completely different functionality depending on the implementation. And on the other hand, a client of the API has to resort to vendor-specific knowledge to be able to call such kernels for desired functionality.

In the following section, we propose to enhance the built-in kernel abstraction of OpenCL with a centralized ”built-in kernel registry,” which can be described as a library of hardware-accelerated function fingerprints that provides the lowest layer of execution model in a platform including hardware-accelerated tasks.

4. Built-in kernel registry

4.1. Defining the built-in kernel registry

Since the built-in kernel functionalities are defined in the OpenCL standard to be implementation-defined, the portability of the built-in kernels to different hardware platforms is extremely limited. The OpenCL standard does not provide any ways to define the built-in kernel semantics, which means that such information must be passed outside of the current standard. This means that the user using built-in kernels in their programs must know exactly what the built-in kernel does. The built-in kernel implementation should therefore document this behavior and make it available to the user. Extracting the behavior definitions to a library of commonly accelerated operations is thus a required extension of the concept to reduce the user effort required to use built-in kernels.

In this work, we propose a *built-in kernel registry* as a centralized way to communicate the semantics of the built-in kernels. It can be described as a shared library of interface definitions. The shared registry makes it possible to abstract the platform-specific built-in kernel implementations from the programs that use them. It also makes it possible for multiple vendors to implement the same built-in kernel functionality for wildly different hardware platforms.

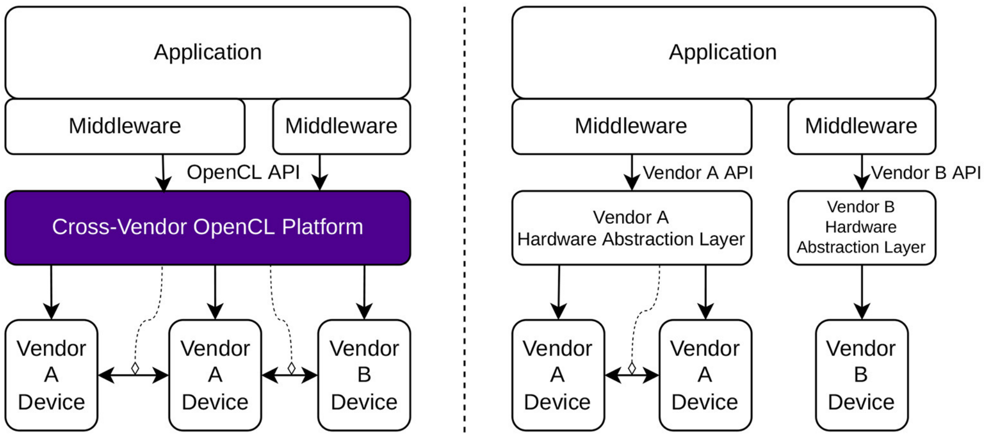

FIGURE 2
(Left) Proposed method of controlling various hardware devices with a common OpenCL platform. **(Right)** Current status of the field with multiple vendor-specific hardware abstraction layers. The vertical arrows correspond to the control commands being passed down the software stack. The horizontal arrows between the devices describe inter-device functionality such as direct memory access and synchronization. The dashed arrows show the management of such inter-device functionality.

The identification of the built-in kernels in the registry is based on their name. The same names are also used in OpenCL programs when using built-in kernels. Every built-in kernel in the registry must have a unique name. We propose a hierarchical naming scheme that follows the structure: *pocl.<category>.<function>.<optional qualifier(s)>.<datatype>*. The naming scheme increases the human readability of the registry, but it is not meant to be used by the tools for semantic information.

The proposed registry must contain all the information required to be able to implement the built-in on different hardware. Therefore the semantics of the built-in kernel must be clearly defined. In the current abstraction, the semantics of the built-in kernels are described either in OpenCL C or SPIR-V (Khronos, 2021). The choice of languages is natural since they are already supported by the OpenCL standard. Furthermore, they also have their language semantics defined at a strict-enough level with minimal aspects left as implementation choices, which is needed to remove ambiguities in the built-in kernel semantic definitions. OpenCL C or SPIR-V based semantic definitions have another advantage of being usable as a software or FPGA high-level synthesis tool fallback in the cases where a programmable device does not have optimized implementation for a certain built-in kernel that is convenient to call from the upper layers of the stack.

Another option for describing the semantics would be an abstract operation graph consisting of other built-in kernels and primitive built-in kernels. The benefits of such a hierarchical description of the built-in kernels would be the easier development of tools that utilize the built-in kernels such as the one example shown in Figure 3. Implementing a hierarchical description of the semantics was not done in this work, but will be investigated in the future.

An example snippet of a built-in kernel registry is shown in Table 1. The built-in kernel registry is designed to be extended in a "hardware first" fashion, where new kernels are added to it whenever hardware acceleration support for them is added the first time. This is different from more traditional library specifications such as TOSA (Linaro, 2021), where the functions are carefully designed to include as few primitive operators as possible. Instead of carefully balancing between hardware specialization and general usability of the functions, the proposed method opts to include even the most specialized versions of the functions in the registry as independent entries.

Since every built-in kernel has to have a unique name if their semantics vary even slightly, it means that the registry is expected and allowed to grow large. The possible variations include different data types, data layouts, kernel specializations, fusings of multiple kernels, etc. The number of these variations combine multiplicatively which means that the registry will easily grow to over thousands of built-in kernels. However, with automatic tools parsing the registry, the size of it is not estimated to become problematic.

Initially, the built-in kernel registry is implemented inside the open-source OpenCL implementation PoCL (Jääskeläinen et al., 2015), which means that the updates to the registry will be merged in by the PoCL developers after community proposes them through pull requests. However, in the future, the registry will be proposed to be merged to the Khronos repository. The registry is designed to be easy to extend for hardware and application developers.

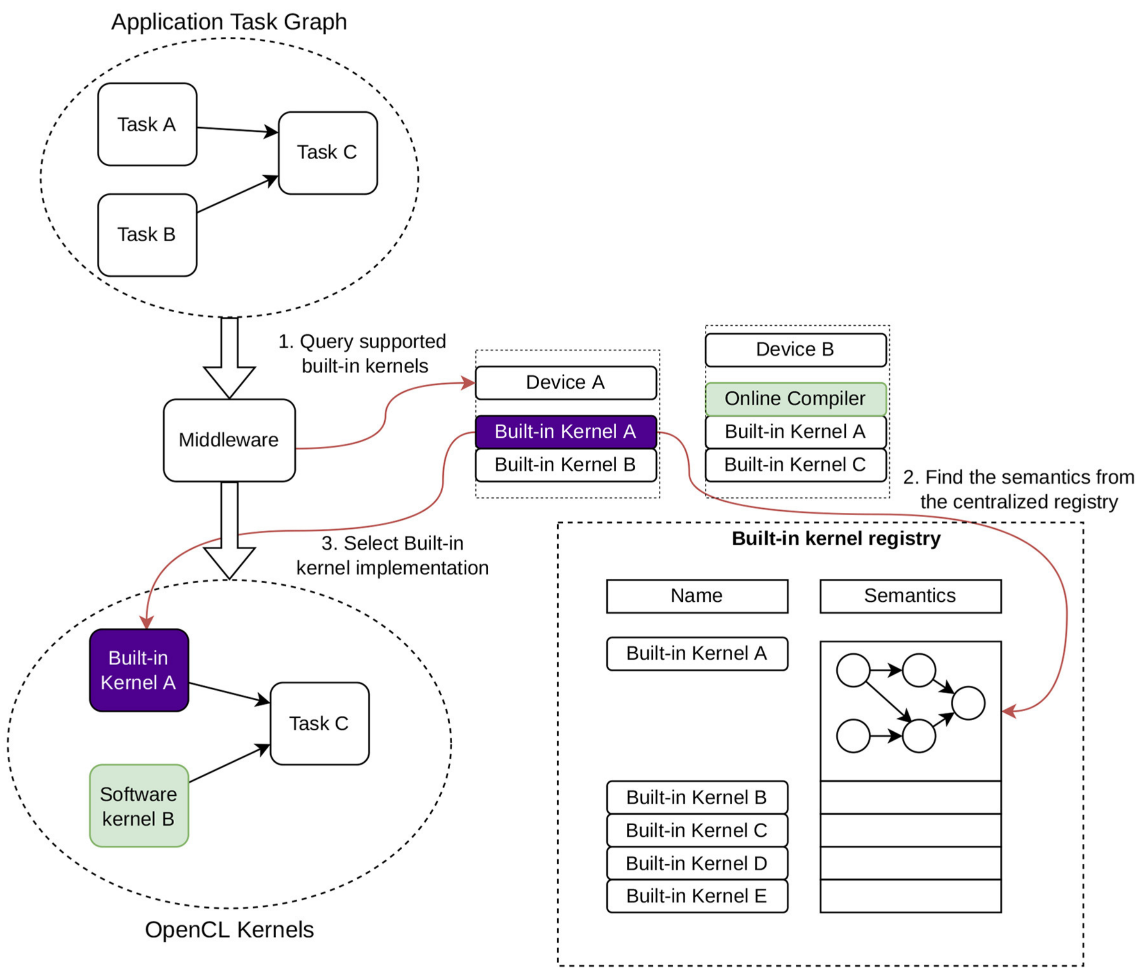

FIGURE 3
Task graph lowering by a middleware software component. The middleware queries the available devices for built-in kernel implementations and lowers the task graph to an OpenCL "task graph" implemented with kernels and event dependencies. It accesses the static built-in kernel registry to match the semantics of the built-in kernel to the input graph semantics.

4.2. Executing the built-in kernels

The fixed high-level specification of the built-in kernels makes it possible to execute them efficiently on different hardware platforms. The abstraction proposed in this work does not define any specific ways of creating the built-in kernel implementations but instead leaves it for the implementation, similar to what is specified in the OpenCL standard. Therefore, the OpenCL implementation is responsible to manage the execution of the built-in kernels. This gives a lot of freedom for the implementation to use the most effective execution method available for each device. Some practical aspects of implementing the OpenCL built-in kernels in the OpenCL implementation are discussed in this section.

In the simplest case, if the host device is also an OpenCL device, it can simply execute the built-in kernel functionality in the host software. Since the built-in kernel abstraction hides the implementation details, the built-in kernel implementation can also consist of calls to an external acceleration library. This enables fast development of built-in kernels through the use of ready-made acceleration components.

Alternatively, if the target device supports online compilation, the implementation can compile the semantic specification as if it was the kernel source code. We call this feature *software fallback*, since it allows the utilization of the built-in kernels even if the device does not have a specialized implementation for that kernel. In this case, the development effort of new built-in kernels is equivalent to developing OpenCL software kernels. Additionally, it is possible to create specialized kernel source-based implementations that can utilize intrinsics or domain-specific languages as long as they are behaviorally equivalent to the semantic specification.

One of the motivations of the built-in kernel abstraction is to enable easier utilization of specialized *fixed-function*

TABLE 1 Example snippet of a built-in kernel registry.

Kernel name	Verbal description of semantics
Arithmetic operations	
pocl.arith.add.i8	Element-wise addition for two arrays of *n* elements.
pocl.arith.mul.i32	Element-wise multiplication for two arrays of *n* elements.
pocl.arith.sqrt.f32	Element-wise square-root for an array of *n* elements.
Deep neural network Operations	
pocl.dnn.conv2d.nchw.f16	2D convolution with parameterizable input and output dimensions, stride, dilation, and padding.
pocl.dnn.conv2d.nchw.f32	2D convolution with parameterizable input and output dimensions, stride, dilation, and padding.
pocl.dnn.conv2d.nchw.relu.i8	2D convolution and ReLU with parameterizable window size and stride for *n* input channels.
pocl.dnn.dense.relu.i8	Dense layer with ReLU activation.
pocl.dnn.maxpool.i8	2D maxpool with parameterizable window size and stride.
OpenVX operations	
khronos.openvx.scale_image.bl.u8	Image rescale with bilinear interpolation.
khronos.openvx.tensor_convert_depth.u8.f32	Converts an 8-bit integer tensor to a 32-bit floating-point tensor with parameterizable norm and offset.
IO operations	
pocl.io.stream.in.i8	Reads *n* bytes from a streaming source to an output buffer.
pocl.io.stream.out.i8	Writes *n* bytes from an input buffer to a streaming source.

The kernels are identified by their name. For illustration purposes, the exact bit-level semantics are not written out in the table. Instead, a short verbal description is included.

hardware implemented as a custom digital circuit. These can also include highly specialized configurable accelerators with limited programmability. Utilizing an FPGA device as a deployment platform for customized accelerators is attractive since FPGA devices enable cost-effective realization of customized logic and memories.

FPGA devices are nowadays programmable by means of high-level languages (also OpenCL C) through high-level synthesis (HLS) tools, thus raising the question of whether HLS-supported FPGAs should be considered as programmable devices instead of hardware accelerators in the proposed abstraction. For now, FPGAs are treated as a collection of IP blocks of which functionality can be invoked through the built-in kernel mechanism. FPGA designers can implement built-in kernels as IP blocks with various methods and tools, without the OpenCL application having to take into account FPGA device-specific details. However, since the proposed abstraction is designed to also include a semantic definition that serves as a software fallback, it could be interesting to provide an alternative means to utilize FPGAs through on-the-fly generated IP blocks by employing HLS tools to generate missing built-in-kernel implementations, or even by integrating them to the OpenCL platform also as online compiler-capable devices.

The specialized hardware only has to be able to execute the specific kernels it has been designed to execute. Therefore, it doesn't have to be generally programmable. However, it needs to be able to connect to and communicate with the OpenCL implementation. This means that for the most portable solutions, the hardware interface must be defined at a quite low level. For example, communication and synchronization through a shared memory hierarchy between different devices should be taken into account when defining the hardware control interface. Examples of such hardware interfaces are HSA (HSA^{TM} Foundation, 2018) and AlmaIF (Leppänen et al., 2021), the latter was used in the evaluation of this work. In the longer term, we believe efficient interoperability between heterogeneous devices still requires further interface specification work, similar to what was done in HSA, in order to probe for an efficient peer-to-peer communication and synchronization means without resorting to device-pair-specific intelligence in the driver layer.

5. Utilization of built-in kernels in higher-level programs

There are multiple different ways to utilize the proposed abstraction layer as an acceleration platform to achieve portability while still having competitive performance due to the use of highly specialized built-in kernels. The proposed abstraction layer only defines the kernel registry to manage the OpenCL built-in kernels and to store and share their semantics. For demonstration purposes, an example method of utilizing the registry by a higher-level program is presented in this section.

The higher-level tool cannot directly use the built-in kernel registry as its backend target, because not all built-in kernels in it are available in the final execution platform. Instead, the higher-level tools need to query the platform for the built-in kernel *implementations*. This is done by a standard OpenCL API call, which returns a list of built-in kernels each device is able to execute. After that, the higher-level tool can lower the internal representation of the program to a task graph consisting of OpenCL kernels.

The concept of a such tool is shown in Figure 3. The tool still utilizes the built-in kernel registry to fetch the built-in kernel semantics based on the unique built-in kernel name, which it can use to choose the optimal set of built-in kernels to execute the program in question. It's important to note that the tool has to always begin by querying the platform for a set of supported built-in kernels. Then out of that set, the tool picks the built-in kernels that produce the best result based on the defined metrics. In principle, this problem of choosing the optimal set of kernels to cover the task graph is very close to the traditional *instruction selection* problem, which can be solved by, e.g., graph covering algorithms (Floch et al., 2010).

The green boxes in Figure 3 correspond to software-based OpenCL kernels, which can be freely intermixed with built-in kernels in devices that support both. The tool queries this information as well at the beginning of its analysis.

For this article, we created a proof-of-concept using the open-source DL compiler TVM (Chen et al., 2018). The implementation replaces certain hand-picked operations from the internal representation of TVM with built-in kernels, which it then passes to its pre-existing OpenCL backend for either direct execution or for the generation of a stand-alone application. Not all the operations need to be replaced by the built-in kernels, the proof-of-concept implementation can pass some parts of the intermediate representation to the OpenCL backend of TVM, which can then fuse them into OpenCL compiled kernels, which are inserted into the same OpenCL command queue as the built-in kernels. In the future, this implementation will be developed further to more closely match the idea shown in Figure 3, which will enable a more generic use of the tool for different neural networks than the ones used for the abstraction evaluation purposes.

The proposed abstraction layer is not meant to be used as just a backend for a single middleware (e.g., DL framework), instead, its purpose is to share the same OpenCL platform with multiple such tools which work together to insert their OpenCL kernels into shared OpenCL command queues, like shown in the left subfigure of Figure 2. This combines the full application phases consisting of also pre- and post-processing to an OpenCL task graph, implemented with kernels, events, and command queues. The implementation can then take advantage of the task-level parallelism and the low-level control of the hardware to most efficiently execute the complete application on a heterogeneous multi-vendor platform.

Once the OpenCL program with the built-in kernels has been constructed, the implementation will then be able to execute it on various types of devices. At this point, the OpenCL program is completely conformant to the standard, which means that it can be passed to any OpenCL *implementation* for execution, as long as they support the built-in kernels in it.

6. Evaluation

In order to test the abstraction layer in practice, an open-source OpenCL implementation PoCL (Jääskeläinen et al., 2015) was extended to support the efficient execution of built-in kernels. To evaluate the portability of the proposed method to different hardware platforms, implementations for neural network inference built-in kernels were created for three different types of hardware platforms (CPU, GPU, and FPGA) from three different vendors (Intel, Nvidia, Xilinx, respectively).

In order to demonstrate the proposed method's ability to execute complete programs, multiple neural network inference applications were mapped to the abstraction. The applications chosen were semantic segmentation, audio classification and object detection: The first case, semantic segmentation, estimates the performance overhead of the proposed method. While the proposed method is not expected to incur significant performance overhead, it is verified by means of a proof-of-concept implementation. The built-in kernel abstraction simply "hardens" the kernel to a fixed specification. Therefore, any software kernel could simply be hardened to a built-in kernel to achieve at least the same performance as the original kernel. Additionally, this case study shows the portability of the proposed method by executing the same OpenCL program on CPU and GPU devices while utilizing existing acceleration libraries.

The second case, audio classification, demonstrates the benefits of the abstraction layer in implementing complete processing pipelines with multiple devices while hiding the low-level implementation details of FPGA-based designs. From the point of view of the higher-level software, the FPGA device is abstracted as a set of regular OpenCL devices implementing the built-in kernels.

And finally, the third case, object detection, demonstrates how the proposed abstraction enables the use of a shared OpenCL context between multiple middleware frameworks. The example application performs preprocessing in OpenVX and neural network inference in TVM, both which internally use OpenCL API to utilize the proposed method. Additionally, this case study shows the flexibility of the method in utilizing heterogeneous platforms, since the preprocessing and the inference can also be executed on different devices.

6.1. Semantic segmentation demonstrator

The first demonstrator was a semantic segmentation application called fastseg (Eric, 2020) based on the MobileNetV3 architecture (Howard et al., 2019). The network was trained with the Cityscapes dataset (Cordts et al., 2016) to annotate 19 different categories. Example input and output can be seen in Figure 4.

The ONNX description of the network was imported to TVM as described in Section 5, where the convolution operations were replaced with calls to built-in kernels. This

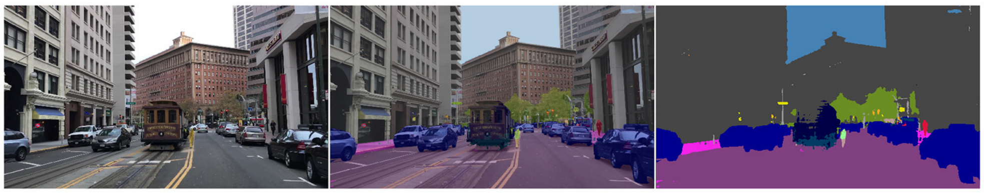

FIGURE 4
Example input and output of the fastseg CNN (Eric, 2020). The network takes an RGB image as an input and outputs a segmented image with an annotation from 19 different categories.

demonstrator utilized TVM's OpenCL runtime to launch the kernels.

The built-in kernel *pocl.dnn.conv2d.nchw.f32* from Table 1 was chosen as the test built-in kernel for this demonstrator. It performs a 2D convolution for N channels with parameterized stride, dilation, and padding. The network consists of also other operations than convolutions. To evaluate the effect of just the convolution built-in kernel, the other operations were implemented with TVM's OpenCL C code generation backend.

6.1.1. Semantic segmentation on GPU

GPU implementation for *pocl.dnn.conv2d.nchw.f32*-kernel was created. It was implemented as a very thin wrapper on top of the cuDNN API by Chetlur et al. (2014), which is designed for neural network acceleration on NVIDIA GPUs. The implementation was built on top of PoCL's support for CUDA devices. The built-in kernel arguments were forwarded to the cuDNN's convolution implementation and the convolution computation was launched.

PoCL compiled the OpenCL C kernels of the remaining non-convolution operations with LLVM's NVPTX backend similar to work done by Hahnfeld et al. (2018). The OpenCL buffer management was implemented with CUDA driver API. This means that the input and output buffers for both NVPTX-compiled and cuDNN kernels can be shared which made it possible to use the same data pointers for both, eliminating unnecessary data copies when compiled and built-in kernels were mixed in the same command queue.

The execution times of the network where some operations were replaced with built-in kernels are compared to the original network executed by TVM's CUDA and cuDNN backends. The GPU used for the measurements was NVIDIA RTX 2080 Ti. The inference runtimes are listed in Table 2. For comparison also the version of the network where all kernels were kept as software kernels is included. The "With accelerated convolutions"-column corresponds to the version of the network where convolutions were performed using the cuDNN-library either using direct external calls from TVM's runtime or using the proposed built-in abstraction layer. The

TABLE 2 Inference runtime in milliseconds for the fastseg semantic segmentation network with NVIDIA RTX 2080 Ti GPU with 1,920 × 1,080 RGB input.

Runtime (ms)	Device	Software kernels	+Accelerated conv	+Tensor cores
TVM's CUDA Backend	GPU	158 (CUDA)	115 (cuDNN)	68 (cuDNN)
TVM's OpenCL backend with built-ins (proposed)	GPU	157 (OpenCL)	110 (cuDNN)	67 (cuDNN)
TVM's OpenCL backend with built-ins (proposed)	CPU	127,699 (OpenCL)	49,868 (oneDNN)	

The column "Software kernels" corresponds to the original application where the kernels are compiled OpenCL C or CUDA kernels. The "+Accelerated conv" -column uses cuDNN/oneDNN for acceleration. The baseline on the first row is TVM's CUDA/cuDNN backend, whereas the proposed method replaces the convolution kernels with built-in kernels, which are then internally executed with calls to cuDNN/oneDNN library. The "+Tensor cores" -column uses cuDNN again, but with fp16 arithmetic, which allows for the utilization of tensor core acceleration hardware.

"With tensor cores"-column is a slightly modified network with fp16 arithmetic, which can take advantage of the specialized acceleration hardware available in the GPU for half precision floating point tensor arithmetic.

As expected, it can be seen from the results that the proposed portability layer does not add performance overhead. Actually, the execution time is slightly lower, but this is due to minor implementation differences between TVM's CUDA Backend and the proposed method utilizing PoCL. Another reason for the slight runtime discrepancy is that for this network the accelerated convolutions may use different internal convolution algorithms. The last column of the results shows that the proposed method enables the use of specialized hardware acceleration blocks by utilizing GPU's tensor cores without incurring an overhead.

6.1.2. Semantic segmentation on CPU

The same built-in kernel *pocl.dnn.conv2d.nchw.f32* was also implemented on Intel i7-3930K CPU. Similar to the GPU implementation, the CPU implementation of it was implemented in PoCL's driver layer as a thin wrapper over oneDNN API (Intel, 2021) designed for neural network acceleration supported by mainly Intel CPUs and GPUs. Similar to the GPU test case, the non-convolution operations were kept as OpenCL C kernels, which PoCL compiled using LLVM's x86 backend.

It is important to note that the OpenCL host program was not modified when the execution platform was switched from GPU cuDNN implementation to CPU oneDNN implementation. This demonstrates the portability of the proposed abstraction layer between different device types from different vendors.

The inference results from the CPU device are shown in the last row in Table 2. It can be seen that utilizing the oneDNN acceleration library improved the performance by a factor of 2.5. The speedup is likely due to the DNN-optimized acceleration library being able to utilize data-level parallelism (multiple CPU cores and SIMD instructions) better than the compiled kernel.

6.2. Audio classifier demonstrator

The second application chosen for evaluation of the proposed abstraction layer was an *audio classification* program. The application classified audio samples to belong to one of 10 classes. The classifier was based on a convolutional neural network by Abdoli et al. (2019), which was further quantized to use 8-bit signed integer values.

Slightly different from the previous semantic segmentation application, all the operations of the network were now replaced with built-in kernels, with no OpenCL C kernels being used. The kernels used for this demonstrator were *pocl.io.stream.in.i8, pocl.add.i8, pocl.dnn.conv2d.nchw.relu.i8, pocl.dnn.dense.relu.i8* and *pocl.dnn.maxpool.i8* from Table 1. The application was split between two devices with the first kernel of the aforementioned list executed on the first device and the rest on the second device.

As opposed to the previous demonstrator, now the TVM's OpenCL runtime was not utilized, but instead, its OpenCL backend was utilized to generate a standalone OpenCL program. This program was then manually augmented with the *pocl.io.stream.in.i8* built-in kernels for the first device and a double-buffering scheme was created for the input data passed between the devices.

A full system-level case study shown in Figure 5 was conducted on an FPGA, starting from a microphone input signal to a classification result, all running in real-time for real input data. PYNQ-Z1 was chosen as a demonstration platform since it includes an ARM CPU, programmable logic (PL) region, and a built-in microphone connected to the PL region. All the data

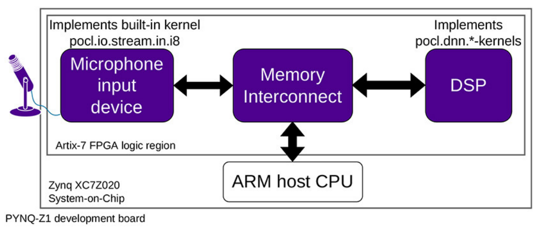

FIGURE 5
Standalone FPGA board demonstration setup which performs a real-time classification of an audio signal.

processing was performed on the FPGA device and the ARM CPU was used as the OpenCL host device. The standalone OpenCL program was transferred to the host CPU where it was compiled as a regular OpenCL program. This avoided having to compile the TVM's full OpenCL runtime on the small embedded CPU.

The built-in kernel implementations were created as custom hardware components, which were instantiated on an FPGA device as two separate OpenCL devices. The first device simply moved the stream of data to on-chip memory, while the second device processed the data further to produce a classification result. The devices implement the hardware interface by Leppänen et al. (2021) to allow them to be connected to the OpenCL implementation PoCL. The common interface also enabled them to communicate and share data directly with each other on-chip.

The application took an input signal directly from the microphone and classified the sound to belong in one of 10 categories. The convolutional neural network used for classification was quantized to an 8-bit signed integer, since implementing floating-point operations on the small FPGA was deemed too costly for this application. The demonstrator worked in real-time to process a 16 kHZ audio stream in 0.5 s windows.

6.2.1. Microphone input device

Built-in kernel abstraction allows for a natural description of IO devices in OpenCL programs. IO device is seen by the OpenCL implementation as a regular OpenCL device that implements the built-in kernel to load the data.

In this demonstrator, the input was coming as a stream of 8-bit PCM-encoded signal values. The signal was available on the FPGA programmable logic region as a stream with valid/ready handshaking from where it should be moved to an on-chip memory component so that it could be processed by the DNN accelerator device.

The device that implements the *pocl.io.stream.in.i8* kernel must already have the streaming interface connected to its datapath. The single argument of the kernel is the pointer to the output memory address that the data must be written to. The first dimension of the work-item range was used to define how many consecutive items should be moved. This makes the first device to be very similar to typical stream-to-memory-mapped DMA components. The main difference is the hardware interface (Leppänen et al., 2021) it implements so that it can be connected to PoCL and controlled just like any OpenCL device by enqueuing built-in kernels. The device's hardware description was created with Vitis HLS tools starting from C language input, which demonstrates that the built-in kernel implementations can be generated with vendor tools, but used as part of vendor-independent abstraction.

6.2.2. Programmable DNN processor

The DNN accelerator was implemented with the OpenASIP toolset by Jääskeläinen et al. (2017). The DNN accelerator was a very small soft processor running on the FPGA fabric. The device had an associated firmware that implemented the kernels required by the application. The firmware was modified by hand to use assembly *intrinsics*. The proposed abstraction layer makes it possible to utilize the custom intrinsics since the kernel functionality is hidden under the built-in kernel abstraction. If the kernels were compiled as OpenCL C kernels, the source code for them would have had to be modified specifically for this device, which would have reduced the portability of the OpenCL program by making it non-compliant to the standard.

The network did not fit into the limited scratchpad memory of the light-weight DNN accelerator at once. Therefore, the network was executed layer by layer, with the host CPU managing the intermediate and coefficient buffers such that the device memory did not run out.

The accelerator ran the inference for the entire network every 500ms to produce a single classification result. The host CPU then took the vector of class probabilities from the OpenCL output buffer, accumulated it together with five previous frames classification results, and displayed the final classification estimate to the console.

The audio classification demonstrator shows that the proposed abstraction layer can be used to create complete applications with multiple devices working in parallel processing real-world data. The well-defined kernel semantics of the abstraction layer was observed to serve as a clear synchronization point between the software engineer working on the application and the hardware engineer creating the built-in kernel implementations.

6.3. Object detection demonstrator

The third case study demonstrates the ability of the proposed abstraction to describe complete task pipelines consisting of multiple domain-specific frameworks and ways to execute them on heterogeneous platforms consisting of multiple device types.

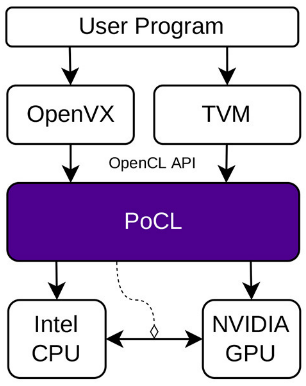

FIGURE 6
Object detection demonstration setup. The user program defines the application in standard OpenVX and TVM library calls, which internally share the OpenCL context with each other.

The application chosen for demonstration was an object detection network YOLOv3-tiny (Redmon and Farhadi, 2018). The network finds bounding boxes for objects in an image and classifies them to belong to a specific category. The complete demonstration application also included an image pre-processing step to show how the proposed abstraction enables efficiently adding pre- and post-processing kernels to neural network-based applications.

The pretrained neural network expects the input data to be in the same format as was used in its training. The YOLOv3-tiny expects a 3 $\times$ 416 $\times$ 416 tensor with a 32-bit floating-point number in range 0.0–1.0. Therefore, conversion from integer storage format is required. The input image has to be rescaled and necessary padding has to be added to the image edges, so the aspect ratio of the image is preserved. TVM implements the preprocessing steps with OpenCV and NumPy-python libraries. However, having the entire application described as a task pipeline targeting a common acceleration backend would be beneficial due to the opportunities related to efficient data sharing and parallelization.

Therefore, for this case study, the pre-processing step was created as a OpenVX (Khronos, 2022) program. A minimal OpenVX implementation was created that implements the API calls needed in the demonstrator. The OpenCL interoperability extension of OpenVX was utilized to allow passing an external OpenCL context, command queues and buffers to the OpenVX implementation. The OpenVX implementation used OpenCL memory objects as backing storage and implemented the OpenVX node functions as built-in kernel calls to OpenCL command queue. For this demonstrator, two new built-in kernels *khronos.openvx.scale_image.bl.u8* and *khronos.openvx.tensor_convert_depth.u8.f32* were added to the

TABLE 3 Execution times for YOLOv3-tiny object detection neural network.

Pre-processing method	Pre-processing time (ms)	Neural network time (ms)
TVM (CPU)	4.9	
OpenVX (GPU)	2.1	33
OpenVX (CPU)	2.3	

The pre-processing consisted of rescaling the 768 × 576 8-bit RGB image to a 416 × 416 image using floating-point format. TVM's pre-processing used OpenCV and NumPy. The OpenVX pre-processing used the proposed built-in kernel abstraction through the OpenVX programming model. The measurements were performed with NVIDIA Quadro K620 GPU and i7-6700 CPU. The neural network execution time was measured to be the same for all of these methods.

built-in kernel registry. The semantics of the built-in kernels are very close to the similarly named functions of the OpenVX standard. The difference is the additional specialization parameters that are hardened in the built-in kernel specification. It is possible to add support for other OpenVX datatypes and specializations inside the OpenVX implementation at the point when the built-in is chosen. The built-in kernels were implemented using the *software fallback*-mechanism described in Section 4.1, which means that they were described as OpenCL C source code, and the OpenCL implementation then compiled the description at run time for the device user is asking for. This made it possible to execute the preprocessing built-in kernels on either CPU or GPU with minimal user effort.

For this demonstrator, the pre-trained YOLOv3-tiny network was executed in TVM's OpenCL runtime and the convolution operations were replaced with the same built-in kernel *pocl.dnn.conv.2d.nchw.f32* as was done in the earlier demonstrator in Section 6.1. The network was then executed on NVIDIA Quadro K260 GPU using PoCL's CUDA driver similar to the semantic segmentation demonstrator. TVM's OpenCL runtime was modified to be able to utilize externally created OpenCL contexts, command queues and input buffers. This way both the OpenVX and TVM could internally push commands to a common OpenCL command queue.

The top-level user application in Figure 6 for this demonstrator was a Python program. The user application managed the OpenCL context, command queues, and the intermediate data buffers. It used OpenVX API to execute the pre-processing commands and called the TVM library to execute the neural network. Figure 6 reflects the goal stated in the Figure 2, where now the two middleware frameworks used the abstraction layer as their target and the execution platform consisted of devices from different vendors.

The runtime results in Table 3 show that the proposed method was able to execute the pre-processing faster than the TVM's current method utilizing OpenCV and NumPy. This is likely due to using a lower-level parallel programming method that minimized unnecessary data copies. The GPU pre-processing is slightly faster than the CPU pre-processing due to the more data-level parallel structure of the GPU. However, it would be possible to get the best system-level performance by creating an interleaved task pipeline where the CPU would perform the pre-processing while GPU would be busy computing the previous frame's detection.

The execution time of the neural network was the same for all the methods in Table 3. Even though the GPU device performing the pre-processing leaves the processed input image in the GPU memory, the transfer time of the input image is not visible in the inference time. This is due to the relatively small size of the image (2.1 MB).

The evaluation framework supports the easy addition of more OpenVX kernels for pre-and post-processing. Thus, this demonstrator showed how the proposed abstraction can be targeted simultaneously by multiple domain-specific frameworks. Additionally, this example demonstrated how the proposed abstraction can execute in heterogeneous platforms with the CPU performing the pre-processing and the GPU accelerating the neural network kernels. The device can be chosen using the standard OpenCL API and the cross-vendor OpenCL platform handles inter-device functionality.

7. Conclusions

Heterogeneous computing platforms have potential for highly efficient execution of different parts of applications, thanks to the specialized hardware. However, programming them efficiently still often requires vendor-specific approaches, which limits the portability of the applications. Abstractions have been proposed in the past, but they still lack especially on the *cross-vendor* portability aspects, leading to duplication of efforts when engineering the heterogeneous software stacks.

This article proposed a software abstraction that is based completely on the current open OpenCL standard. The proposed abstraction leans on the idea of a shared centralized built-in kernel registry that includes built-in kernel semantics to facilitate cross-vendor implementation. In order to demonstrate the abstraction with a cross-vendor OpenCL implementation, the abstraction was implemented on PoCL, an open-source OpenCL implementation. The performance portability of the built-in kernels was demonstrated by implementing the same convolution kernel on CPU and GPU devices utilizing commercial neural network acceleration libraries. The overhead of the built-in kernel abstraction was shown to have minimal impact on the execution time. Additionally, the ability to use the proposed abstraction as a target of multiple input languages was shown with the case study combining OpenVX and TVM both utilizing the proposed abstraction with portable OpenCL API. Benefits of the built-in kernels approach were also identified for utilizing fixed-function and soft processor-based devices implemented on an FPGA fabric. The case study also demonstrated cooperative execution between multiple OpenCL devices on a single FPGA programmable logic fabric to execute full task pipelines.

In order to stimulate adoption and further development of the abstraction layer, we will merge the example built-in kernel implementation facilities to the upstream PoCL project and publish the built-in kernel registry as a community-driven open source repository. In the longer term, we aim to refine the registry to the point that it can be integrated to official Khronos open source repositories. Technical plans for the future include adding support for more domain specific languages on top with general lowering methods and further streamlining the FPGA support.

Data availability statement

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

Author contributions

TL developed the support for built-in kernel abstraction to the OpenCL implementation and created the example built-in kernels used for evaluation for GPU, CPU, and FPGA devices. AL created the proof-of-concept integration of the abstraction to TVM (Section 5) and prepared the evaluation applications for Section 6 and ran the measurements. PJ contributed significantly to the conception and the design of the study. All authors wrote sections of the manuscript and contributed to its revision, read, and approved the submitted version.

Funding

This work was supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 871738 (CPSoSaware).

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., et al. (2015). *TensorFlow: Large-Scale Machine Learning On Heterogeneous Systems*. Available online at: tensorflow.org.

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

AMD (2022). *The AMD ROCm 5 Open Platform for HPC and ML Workloads*. Available online at: https://www.amd.com/system/files/documents/the-amd-rocm-5-open-platform-for-hpc-and-ml-workloads.pdf

Bai, J., Lu, F., and Zhang, K. (2019). *ONNX: Open Neural Network Exchange*. Available online at: https://github.com/onnx/onnx

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., et al. (2015). MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv: 1512.01274 [Preprint]*. Available online at: https://arxiv.org/abs/1512.01274

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., et al. (2018). "TVM: an automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA), 578–594.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., et al. (2014). cuDNN: efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*. doi: 10.48550/ARXIV.1410.0759

Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., et al. (2016). "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (Las Vegas, NV), 3213–3223.

Eric, Z. (2020). *Fastseg*. Available online at: https://github.com/ekzhang/fastseg (accessed May 13, 2022).

Floch, A., Wolinski, C., and Kuchcinski, K. (2010). "Combined scheduling and instruction selection for processors with reconfigurable cell fabric," in *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors* (Rennes), 167–174.

Galoppo, N., Hansen-Sturm, C., Guo, Y., and Ashbaugh, B. (2016). *OpenCL Extension cl_intel_accelerator*. Available online at: https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_accelerator.txt

Hahnfeld, J., Terboven, C., Price, J., Pflug, H. J., and Müller, M. S. (2018). "Evaluation of asynchronous offloading capabilities of accelerator programming models for multiple devices," in *Accelerator Programming Using Directives*, eds S. Chandrasekaran and G. Juckeland (Cham: Springer International Publishing), 160–182.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). "Delving deep into rectifiers: surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision* (Santiago), 1026–1034.

Henry, S., Denis, A., Barthou, D., Counilh, M.-C., and Namyst, R. (2014). "Toward OpenCL automatic multi-device support," in *Euro-Par 2014 Parallel Processing*, eds F. Silva, I. Dutra, and V. Santos Costa (Cham: Springer International Publishing), 776–787.

Hochreiter, S., and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.* 9, 1735–1780.

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., et al. (2019). "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision* (Seoul), 1314–1324.

HSA^{TM} Foundation (2018). *HSA Platform System Architecture Specification v1.2*. HSA^{TM} Foundation

Intel (2021). *OneAPI Specification 1.1*. Available online at: https://spec.oneapi.io/versions/latest/oneAPI-spec.pdf

Jääskeläinen, P., Korhonen, V., Koskela, M., Takala, J., Egiazarian, K., Danielyan, A., et al. (2019). Exploiting task parallelism with OpenCL: a case study. *J. Signal Process. Syst*. 91, 33–46. doi: 10.1007/s11265-018-1416-1

Jääskeläinen, P., Sanchez de La Lama, C., Schnetter, E., Raiskila, K., Takala, J., and Berg, H. (2015). pocl: a performance-portable OpenCL implementation. *Int. J. Parall. Programm*. 43, 752–785. doi: 10.48550/arXiv.1611.07083

Jääskeläinen, P., Viitanen, T., Takala, J., and Berg, H. (2017). *HW/SW Co-Design Toolset for Customization of Exposed Datapath Processors* (New York, NY: Springer International Publishing), 147–164.

Katel, N., Khandelwal, V., and Bondhugula, U. (2022). "MLIR-based code generation for GPU tensor cores," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, CC 2022* (New York, NY: Association for Computing Machinery), 117–128.

Khronos (2021). *SPIR-V specification 1.6*. Available online at: https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.html

Khronos (2022). *The OpenVX Specification 1.3.1*. Available online at: https://www.khronos.org/registry/OpenVX/specs/1.3.1/html/OpenVX_Specification_1_3_1.html

Khronos® OpenCL Working Group (2020). *The OpenCL^{TM}Specification*. Khronos® OpenCL Working Group.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., et al. (2021). "MLIR: scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Seoul), 2–14.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., et al. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Comput*. 1, 541–551.

Leppänen, T., Mousouliotis, P., Keramidas, G., Multanen, J., and Jääskeläinen, P. (2021). "Unified OpenCL integration methodology for FPGA designs," in *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)* (Oslo), 1–7.

Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., et al. (2021). The deep learning compiler: a comprehensive survey. *IEEE Trans. Parall. Distrib. Syst*. 32, 708–727. doi: 10.1109/TPDS.2020.3030548

Linaro (2021). *TOSA 0.23.0 Specification*. Available online at: https://developer.mlplatform.org/w/tosa/

Luebke, D. (2008). "Cuda: scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro* (Paris), 836–838.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Red Hook, NY: Curran Associates Inc.

Reddy, S. R. K., Wang, H., Bourd, A., Golikeri, A., and Calidas, B. (2022). "OpenCLML integration with TVM," in *International Workshop on OpenCL* (Bristol), 1.

Redmon, J., and Farhadi, A. (2018). YOLOv3: an incremental improvement. *arXiv: 1804.02767 [Preprint]*. Available online at: https://arxiv.org/abs/1804.02767