This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Bitstream Database-Driven FPGA Programming Flow Based on Standard OpenCL

Topi Leppänen, Leevi Leppänen, Joonas Multanen, and Pekka Jääskeläinen

Abstract— **Field-programmable gate array (FPGA) vendors provide high-level synthesis (HLS) compilers with accompanying OpenCL runtimes to enable easier use of their devices by non-hardware experts. However, the current runtimes provided by the vendors are not OpenCL-compliant, limiting the application portability and making it difficult to integrate FPGA devices in heterogeneous computing platforms. We propose an automated FPGA management tool AFOCL, with a guiding principle that the software programmer should only need to use the standard OpenCL API to manage FPGA acceleration tasks. This improves portability since the same OpenCL program will work on any OpenCL-compliant computation device able to execute the same kernels, including CPUs, GPUs, and FPGAs. The proposed approach is based on pre-optimized FPGA bitstreams implementing well-defined OpenCL built-in kernels. This enables a clean separation of responsibilities between a hardware developer preparing the FPGA bitstreams containing the kernel implementations, a software developer launching computation tasks as OpenCL built-in kernels, and a bitstream distributor providing preoptimized FPGA IPs to end-users. The automated FPGA programming tool fetches bitstream files as needed from the distributor, reconfigures the FPGA, and manages the communication with the accelerator. We demonstrate that it is possible to achieve similar performance as the current FPGA vendor OpenCL implementations, while abstracting all FPGA-specific details from the software programmer. The cross-vendor potential of AFOCL is shown by porting the implementation to FPGAs from two different vendors (AMD and Altera), and to two different FPGA types [PCIe and system-on-chip (SoC)], and controlling all these systems with the same OpenCL host program.**

Index Terms— **Accelerator integration, field-programmable gate array (FPGA) overlay, FPGA virtualization, heterogeneous computing, open standard, OpenCL.**

I. Introduction

PROGRAMMING field-programmable gate arrays (FPGAs) in a cross-vendor portable and efficient way is challenging. The configuration flexibility of FPGAs allows them to work as highly efficient accelerators, but their use as general-purpose programmable devices is limited by their complex and very fine-grained configuration method. Typically to configure FPGAs, a digital logic circuit description is created using register-transfer level (RTL) languages. The development of digital circuit description is time-consuming and difficult, which limits the wide utilization of FPGAs.

High-level synthesis (HLS) tools enable automatic conversion of high-level language programs such as C, C++, and OpenCL C to circuit descriptions, which can then be synthesized, placed, and routed with the FPGA tooling to produce an FPGA configuration bitstream. Open standards such as OpenCL to develop heterogeneous applications are beneficial, as they can make the application descriptions portable between computing platforms and can serve as an intermediate code generation target for higher level frameworks (e.g., OpenMP, SYCL, or TVM [1]). However, OpenCL implementations by FPGA vendors require vendor-specific changes to OpenCL programs, limiting the application's portability. Additionally, the HLS tools take prohibitively long to generate the FPGA bitstreams, which prevents their inclusion in the OpenCL application runtime, leading to the end user having to manage the bitstreams themselves.

In this work, we propose a method to enable a clean separation of the following roles in FPGA programming, each requiring only their special expertise.

1) Software developers, who do not need to possess hardware engineering skills, but can write OpenCL applications.
2) Hardware developers, who optimize IPs for integration to the OpenCL platform.
3) IP distributors, who provide IPs remotely to the end users.
4) End users, who do not need to install FPGA tooling or manage FPGA bitstreams themselves.

To facilitate the implementation of the proposed role distribution, we release an extensible open-source platform AFOCL for integrating precreated FPGA IPs to an OpenCL implementation. The proposed approach adds modularity to the FPGA development, as the kernel implementation and optimization are more clearly separated from the OpenCL host application. When the end user launches a kernel for computation using the standard OpenCL API, the implementation fetches a suitable bitstream from a remote server, reconfigures the FPGA, and launches the computation.

The use of standard OpenCL is crucial, as it allows easier application porting from different acceleration devices to FPGAs. The proposed open-source platform is FPGA vendor- and device type-agnostic and does not rely on any

Received 8 April 2024; revised 31 July 2024; accepted 6 September 2024. *(Corresponding author: Topi Leppänen.)*

The authors are with the Faculty of Information Technology and Communication Sciences, Tampere University, 33720 Tampere, Finland (e-mail: topi.leppanen@tuni.fi).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2024.3458062.

Digital Object Identifier 10.1109/TVLSI.2024.3458062

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

single vendor's devices or closed-source tooling, for maximum portability.

The main contributions of this work are the following.

1) A method to clearly separate the roles in FPGA programming by divorcing the OpenCL kernel implementation and optimization from its distribution and use.
2) An open-source tool AFOCL integrated into an OpenCL implementation for automating the FPGA bitstream management and reconfiguration, all controllable via standard OpenCL.
3) A mechanism for a remote bitstream database for storing optimized kernel implementations, reusing the effort expended by the hardware experts.

This article extends our previous work [2] in which we originally introduced AFOCL. In that work, we demonstrated our implementation on AMD and Altera PCIe FPGA cards using a minimal vector addition benchmark. In this article, we provide the following additional contributions.

1) A thorough analysis of how the proposed flow enables the specialization of different roles in the FPGA programming ecosystem.
2) Bitstream distribution proof-of-concepts via either a self-hosted HTTP-server or Amazon Web Services (AWS) EC2 F1 instances.
3) Evaluation with stencil memory access patterns to show that the proposed method can handle more complex memory access patterns than shown in the original article.
4) A scalability study demonstrating the ability to increase FPGA utilization and to efficiently use high-bandwidth memory (HBM).
5) Porting of the implementation to a system-on-chip (SoC) FPGA and a custom shell for AMD FPGAs to further demonstrate the generality of the proposed approach.

The source code to generate the bitstream database is released together with the software driver that connects to the database, integrated to an OpenCL implementation PoCL.[1]

II. Background on OpenCL

OpenCL [3] is well suited as a portability layer between higher level frameworks and heterogeneous computing devices, due to its close-to-machine level of programming model and a generic runtime API with capability for platform discovery. Because of this, it has been widely implemented by many hardware vendors for various types of heterogeneous computing devices.

OpenCL defines a standard API to offload functions for acceleration on the computing devices. These accelerated functions are called *kernels*. The kernels are provided by the user either as a source code, which is then compiled by the OpenCL implementation or as *built-in kernels*, the behavior of which is defined by the OpenCL implementation.

Additionally, OpenCL defines standard API functions to move data to and from computing devices with distinct memory address spaces, enabling the control of highly heterogeneous systems from a common host processor.

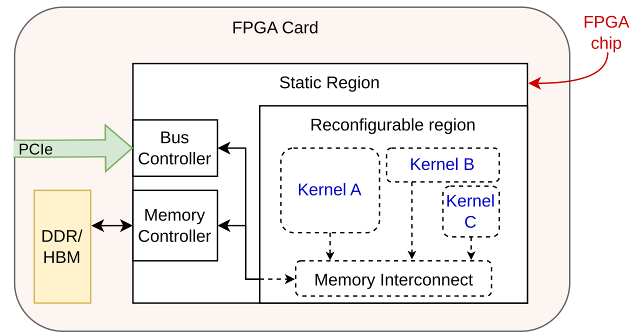

Fig. 1. Typical shell implementation for PCIe FPGAs to implement OpenCL. The static region is set up as part of the platform installation. The reconfigurable region is updated based on the OpenCL program.

A. Built-In Kernel Definitions

OpenCL v1.2 defined the semantics of built-in kernels as completely *implementation-defined* [3]. The user supplies only the built-in kernel name as a string when they are creating OpenCL program objects and launching the kernels. This gives the OpenCL implementation complete freedom in deciding what can happen when the user launches built-in kernels. Naturally, the user should not expect that the same built-in kernel name would correspond to the same behavior in two different OpenCL implementations. This prevents any portability of OpenCL built-in kernels.

To address this, in our previous work [4], we proposed a centralized registry that maps the built-in kernel name to a well-defined kernel interface and behavior. Then, the user can expect to always get the same behavior for the built-in kernels of the registry. Since then, the idea has been redefined and suggested as an OpenCL standard extension called defined built-in kernels (DBKs) [5], which standardizes a new API for the well-DBKs, instead of reusing the built-in kernel API. DBKs are not FPGA-specific and therefore work exactly with the same behavior in any OpenCL device that implements them.

B. FPGA Vendor OpenCL Implementations

OpenCL has been partially adopted by the FPGA vendors AMD and Altera, and it is possible to describe computation kernels in OpenCL C and to launch them from OpenCL host programs. However, the FPGA vendor OpenCL API implementations to control FPGAs are not compliant with the specification. In practice, this means that the user is tasked to manage some of the FPGA-specific details, and is not able to use all of the OpenCL features. This makes the OpenCL applications not portable between FPGAs and other OpenCL platforms.

Typical OpenCL implementations for PCIe FPGA devices have a structure similar to Fig. 1. The programmable logic region is split into a static shell and a dynamically reconfigurable region. The shell contains the boilerplate logic required to communicate with external IO, such as PCIe and

[1] Source code to generate an example bitstream database at: https://github.com/cpc/AFOCL. The source code for the OpenCL implementation at: http://code.portablecl.org.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

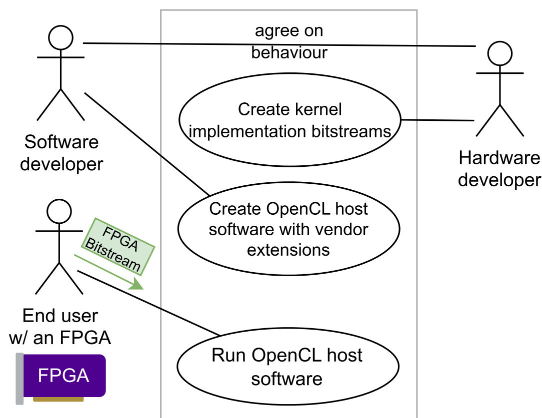

Fig. 2. Current state of the FPGA programming ecosystem with vendor OpenCL implementations. The end user provides the FPGA bitstream to the OpenCL implementation with a *clCreateProgramWithBinary*-call. The software and hardware developer are both making vendor-specific changes to their parts of the application while coordinating with each other to achieve the target behavior of the application.

DDR interfaces so that the FPGA can be partially reconfigured without taking down the PCIe link. A large reconfigurable region is left to implement the logic of the kernels.

To accelerate a kernel on this type of platform, the kernel program source code is passed to an HLS tool, which generates RTL descriptions of the kernels. Then, the RTL kernels are synthesized with FPGA tooling, and the resulting netlist is placed and routed to fit into the partially reconfigurable region of the shell. This method follows the spirit of the OpenCL programming model, in which the kernel compilation is separated from the host application compilation.

As the synthesis from high-level languages to FPGA bitstreams is such a complicated task, the runtime of the HLS tools can easily be multiple hours (depending on the complexity of the kernel and the size of the partially reconfigurable region). This prevents using the *just-in time* compilation feature of OpenCL, creating a separation in time between the kernel compilation and application runtime. Management of this issue is typically left to the user, who has to make sure to have the FPGA bitstream files ready on their local machine before launching the OpenCL application. This is different to CPU and GPU OpenCL implementations, which typically performs the kernel compilation as a part of the application runtime.

The separation of the program compilation from the application runtime is convenient for another reason, which is why it is unlikely that a generic OpenCL kernel would reach good performance without any FPGA-specific optimizations. Therefore, the kernel bitstreams are typically prepared and optimized in advance by a hardware expert as shown in Fig. 2. For the best results, the developer has to perform hardware design with the HLS tools, utilizing premade libraries and direct the RTL generation by augmenting the program with pragmas [6]. These transformations are often vendor-specific, which leads to a vendor lock-in effect, where the user cannot easily switch between hardware vendors, because the cost of porting and reoptimizing the codebase is higher than the potential savings available from more competitive hardware.

FPGA vendor OpenCL implementations do not support all the generic features of OpenCL host programs. For example, the OpenCL *program* object creation is tied to the FPGA reconfiguration, as illustrated in Fig. 3. Also, in the AMD's implementation for Alveo FPGAs, FPGA reconfiguration is prevented if there are OpenCL buffers allocated on the device, tying the OpenCL buffer lifetimes to the FPGA reconfiguration. This seems unnecessarily restricting, as the buffers are actually on a separate memory chip, and the reconfiguration of the FPGA does not need to invalidate the external memory contents.

In standard-compliant OpenCL implementations, it should be possible for the user to first create all the OpenCL program objects beforehand, and use kernels in them in any order the user wishes. Additionally, the program objects should not be tied to the buffer objects, meaning that the user should have a convenient way to use the same buffer objects for kernels originating from different OpenCL program objects.

III. AFOCL: Automated Kernel Bitstream Management

To describe how the proposed approach is constructed, we first describe its main component AFOCL (consisting of a bitstream database and a runtime driver), and then how the methodology is interacted with by the different roles (end user, software developer, hardware developer, and bitstream distributor). A visual summary of the responsibilities of the different roles is shown in Fig. 4.

A. Bitstream Database

At the core of the proposed approach is the vendor- and FPGA-type agnostic tool AFOCL, integrated into an OpenCL implementation. AFOCL combines our earlier work [4] on the well-DBK semantics with automated and cross-vendor FPGA reconfiguration management. The proposed FPGA programming flow hides the FPGA-specific details from the OpenCL end user, by moving the low-level FPGA management to the OpenCL implementation.

On the FPGA, the proposed approach uses a similar shell as vendor OpenCL FPGA implementations shown in Fig. 1. The reconfigurable region of the shell is automatically reconfigured with a preoptimized and synthesized partial bitstream, which implements an AlmaIF accelerator, as specified in our earlier work [7]. The bitstreams are downloaded from a remote database by the implementation as needed. The same database can be used to host partial bitstreams for different FPGAs.

The built-in kernel implementations in the database follow the kernel semantics from the built-in kernel registry, similar to DBKs proposal [5]. This means that every built-in kernel implementation in the database has exactly the behavior defined in the built-in kernel registry, identified by the same human-readable name, as shown in Fig. 5. This creates a clean abstraction between the database creation and use.

B. Runtime Driver

The implementation of the automated bitstream database management includes a driver component integrated to a

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

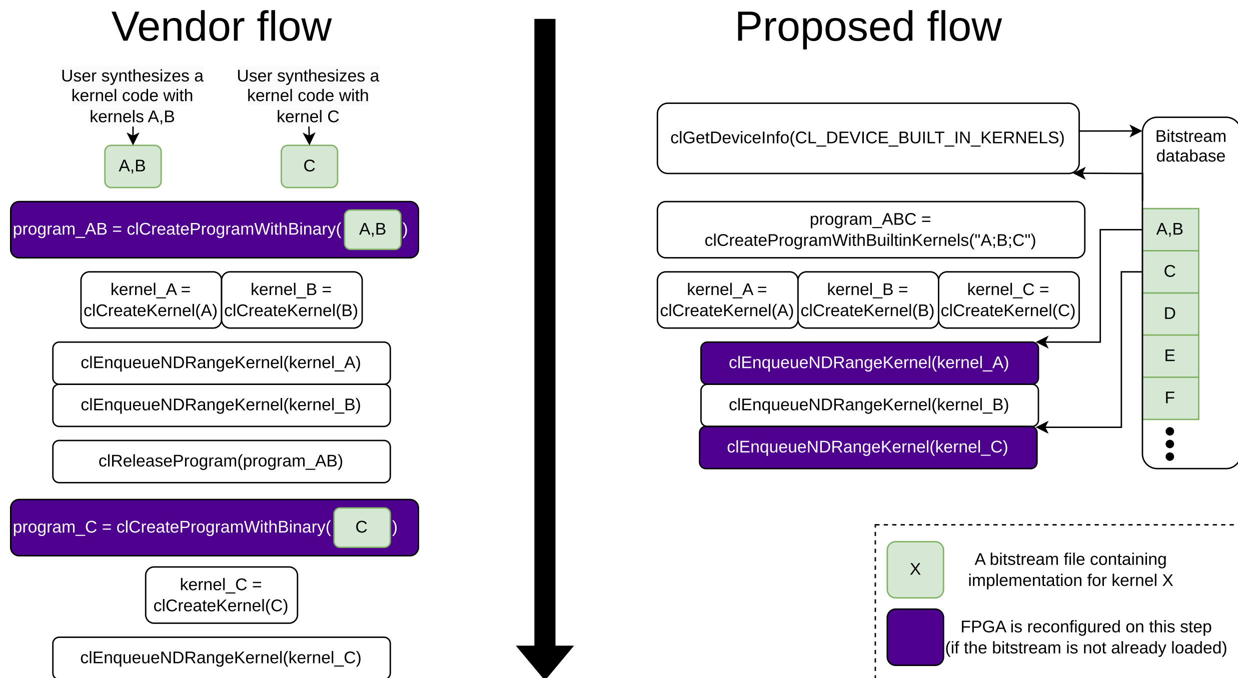

Fig. 3. In FPGA vendor OpenCL implementations, the user directs the FPGA reconfiguration with *clCreateProgramWithBinary*-API calls, tying the FPGA reconfiguration to the OpenCL program object. In the proposed AFOCL flow, the reconfiguration happens automatically behind the scenes based on the contents of the bitstream database and the kernel the user has launched.

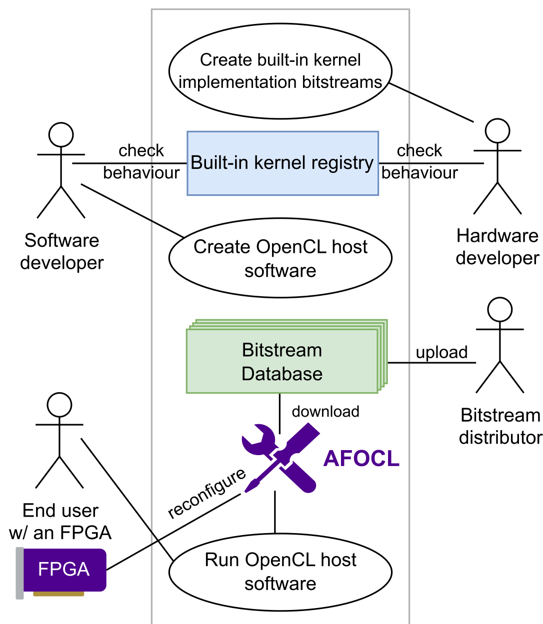

Fig. 4. Key roles involved in the proposed FPGA programming ecosystem. The different users are able to work with clear, standardized abstractions. This is in contrast to the vendor-specific flow shown in Fig. 2.

full OpenCL implementation. The driver fetches and parses the remote database index to know which built-in kernels are available for execution. It reports this back to the OpenCL user via the standard OpenCL device query parameter *CL_DEVICE_BUILT_IN_KERNELS*.

After the user has requested to execute a built-in kernel, and before the implementation actually launches it for execution, the implementation fetches the bitstream from the external database and performs *dynamic partial reconfiguration* to update the reconfigurable region in the shell. This all happens completely behind the scenes with no user involvement. OpenCL memory accesses such as clEnqueueReadBuffer and clEnqueueWriteBuffer are also internally directed to the FPGA using potentially different mechanisms for FPGAs from different vendors, as shown in Fig. 6.

IV. Roles in FPGA Programming

In the following four sections, we discuss the perspectives of end users, software developers, hardware developers, and bitstream distributors. The separation of different roles demonstrates the convenience of the proposed platform in how it distributes the different responsibilities of AFOCL users. The clear division into different roles is highly beneficial, as it enables expertise concentration and convenient integration of efforts from different domain experts.

While more user roles could be identified (e.g., FPGA device vendor or built-in kernel registry maintainer) that also holds interest in the proposed FPGA ecosystem, in this analysis we concentrate on the four most critical roles illustrated in Fig. 4.

A. End User Perspective

The end user does not need to learn any vendor-specific FPGA design tooling or manage FPGA bitstreams and can focus on using the application itself. They need to have access to an FPGA compatible with our database and a software application (likely also in a binary format) that they want to execute. The end user does not need to install FPGA tooling to generate bitstreams. This minimizes the cold-start time from

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

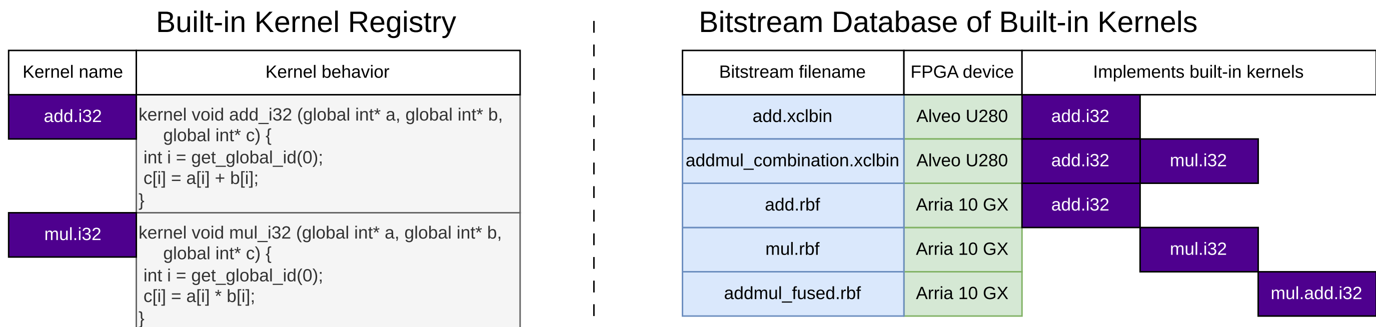

Fig. 5. Correspondence between the built-in kernel registry [4] and the bitstream database. The database contains the implementation bitstreams for kernels in the registry for different FPGAs, in different combinations.

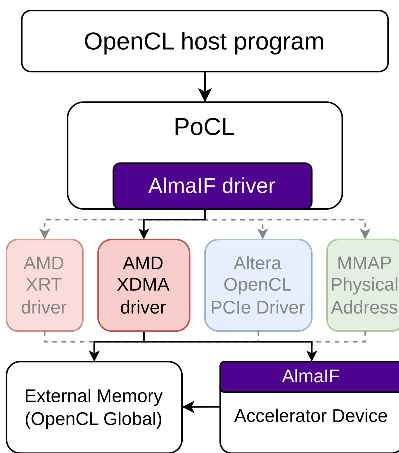

Fig. 6. Diagram showing how the proposed implementation forwards the OpenCL API commands to various FPGA types. The AlmaIF driver supports multiple different FPGA types and redirects the commands depending on which FPGA is used. AMD XRT driver is used with Alveo's Vitis-based platform shell, AMD XDMA driver for custom AMD PCIe shells and on AWS EC2 F1 instances, Altera driver for Altera PCIe cards, and MMAP-based access on SoC FPGAs where the accelerator is accessible in the physical address space.

installing an FPGA to running useful applications with it. To interface with the proposed approach, it is enough that they set up access to the remote bitstream database, after which they can launch the OpenCL application with the provided OpenCL implementation.

B. Software Developer Perspective

The proposed approach simplifies the software for controlling FPGAs significantly. Software developers do not need to manage the low-level FPGA-specific details and can utilize the FPGAs just like any other OpenCL device. They can write standard OpenCL host applications that launch computation tasks as (built-in) kernels and move data between OpenCL devices and the host using the standard OpenCL API. When there are multiple kernels in a single bitstream, the user can decide to execute them one after another using the standard *clSetKernelArg* API to connect the output of one to the input of another.

The use of an open standard strengthens and future-proofs the approach, since it now becomes possible and worthwhile to build more advanced frameworks on top of a well-known OpenCL programming model. It is possible to use FPGAs together with other devices in a multi-device OpenCL implementation. Additionally, existing frameworks with an OpenCL runtime backend could now become easier to port to FPGAs.

To develop the OpenCL host applications, the software developer does not even need to use FPGAs. Since the proposed platform is built on OpenCL compliance, the software developer could use, e.g., their desktop machine as a CPU OpenCL device, as long as their implementation supports the same set of built-in kernels they are using. The compatibility between the different built-in kernel implementations is provided by the DBKs-abstraction [5].

C. Hardware Developer Perspective

As the proposed platform relies on pre-synthesized FPGA bitstreams, somebody has to create the database that contains the kernel implementations. In the proposed flow, this task is left to hardware developers who can develop the kernels against a fixed kernel specification, optimizing and updating their hardware implementation without changing the software users' view of the kernel.

The challenges related to efficient utilization of FPGA still have to be solved, and it is the responsibility of the hardware developer to create the optimized kernel implementations. The hardware developer is free to use any tools available to create the accelerator components. In some cases, it is possible to utilize HLS tools or soft processors, whereas in others, the use of RTL component descriptions or FPGA-specific tooling is mandatory to maximize the performance.

The only requirement by the proposed implementation is the strict adherence to the memory map defined by the AlmaIF interface protocol we specified in [7]. This is needed so that the included software driver can control the accelerator via a consistent interface. AlmaIF defines a number of control registers, a configuration memory, a queueing mechanism to submit kernels and barriers, and a small data memory for the dynamic allocation of small data quantities. In terms of bus interface protocols, the actual interfaces depend on the shell implementation. At a minimum, there has to be a single completer memory interface to access the AlmaIF memory map. Additionally, for efficient use of external memory, the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

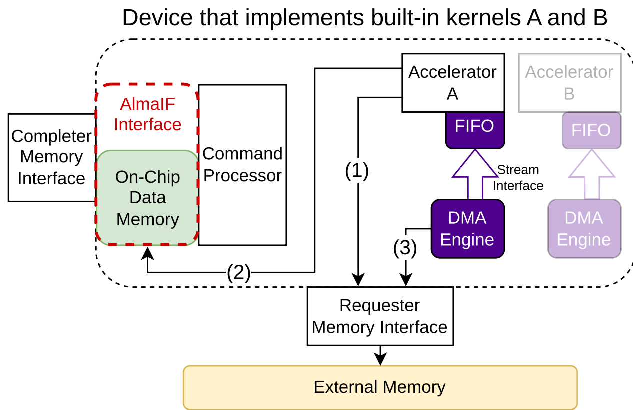

Fig. 7. Template for creating new built-in kernel implementations. The template supports three different methods for moving data to and from the accelerators. 1—Random access to external memory. 2—Random access to on-chip memory. 3—Burst access to external memory using DMA engines.

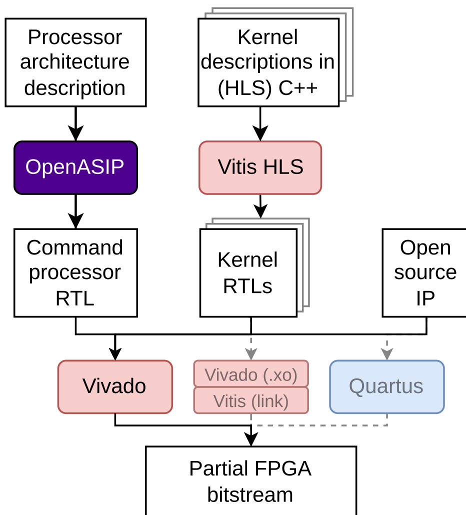

Fig. 8. Toolflow for synthesizing the command processor template to FPGA bitstreams. There are three alternative paths: Vivado for custom shells, SoC FPGAs and AWS F1, Vivado+Vitis for integrating to Viti's platform shells, and Quartus for custom shells on Altera Arria 10 FPGAs.

kernel can have a shell-specific number of requester memory interfaces that perform load and store operations between the FPGA external memory and the accelerator.

As we recognize the potential difficulty in creating these optimized built-in kernel implementations, we have released a template shown in Fig. 7 to serve as a starting point design. The template includes a command processor with accompanying firmware generated with OpenASIP [8] that implements the AlmaIF interface. The hardware developer's main effort is in creating the accelerator IPs and configuring the data movement to and from memories.

The template provides three alternative methods for memory access, each with its own benefits and use cases.

1) *Random Access to External Memory:* A requester interface out of the accelerator. Meant to be used when the access pattern is too random to be easily converted to utilize the DMA units.
2) *Random Access to On-Chip Memory:* A fast access to a small amount of on-chip memory. Suitable for configuration parameters, and small amounts of data.
3) *Streaming Dataflow:* Burst access is generated by DMA units (memory-mapped-to-stream and stream-to-memory-mapped). Most efficient way to move large amounts of data.

A single bitstream can include implementations for multiple built-in kernels, as shown in Fig. 5. If the kernels are *always* used in the same order one after another, it might be possible to *fuse* them to a single kernel, eliminating the intermediate data transfers, as shown in the last row of the figure. Currently, this kernel fusion needs to be performed manually by the hardware developer, by adding both a new DBK to the built-in kernel registry and its implementation to the bitstream database.

The current toolflow supports three different FPGA tool alternatives shown in Fig. 8 for synthesizing RTL to FPGA bitstreams. First, the highlighted Vivado flow can be used to create a block design in Vivado, which is then synthesized either as a region of a custom PCIe shell or to fill an entire SoC FPGA. The custom Vivado flow is important for increased flexibility, as it allows the creation of custom shells, instead of relying on the AMD-provided Vitis platform shell architecture. Second, the Vitis-based flow first wraps the RTL sources in a block design and generates a .xo-file out of it. This .xo-file is then *linked* to an Alveo platform shell using Vitis. The Vitis-flow enables convenient simulation of the entire platform using Vitis's hardware emulation feature, into which the OpenCL implementation can connect. Third, the Quartus flow is meant for Altera Arria 10 FPGAs, and it synthesizes the RTL sources to a reconfigurable region of a custom shell.

After the hardware developer has created the partial bitstream file, they upload it to the database together with accompanying metadata, which includes a list of the DBKs the bitstream implements, and any configuration firmware needed by the bitstream. The driver uploads the firmware to the AlmaIF configuration memory after the bitstream is loaded.

It is clear that the generation and synthesis of built-in kernel implementations requires significant effort and is likely to require FPGA-specific optimizations. However, since the same hardware design is reused from the database multiple times by different users, the high hardware development cost is amortized over all the database users.

D. Distributor Perspective

A hindrance in the wide usage of FPGAs for acceleration tasks is the reliance on the users to develop their own accelerators. Even the compilation and FPGA synthesis of an existing kernel can be an inconvenient task since it requires the installation and use of large and slow FPGA tools. The world of software has long been distributing application binaries, and it is rarely required for the end users to build large applications from source. Similarly, to widen the adoption of FPGAs for generic acceleration tasks, good bitstream distribution mechanisms are needed.

The proposed flow opens up interesting possibilities for FPGA bitstream distributors, who are able to set up repositories offering optimized bitstreams. With external

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

distributors, the end user may connect to a free or paid repository, and get access to FPGA acceleration kernels, which work directly in the provided open-source OpenCL implementation.

Fully open-source bitstream distribution servers have a significant potential to increase the community adoption of FPGAs. With open-source distribution, the expensive hardware developer effort can benefit users globally, with no need for each user to be an FPGA (tooling) expert. Naturally, the license terms of all the technologies used need to be reviewed to make sure that distributing the tool outputs (bitstreams) is legal. The legality is not estimated to be a critical issue, since the proposed method is FPGA-vendor agnostic, it should always be possible to use FPGAs with license terms permitting the distribution.

The distributor is working with the same DBK-abstraction as the software and hardware developers. Therefore, the distributor can transparently update the built-in kernel implementations in their database, since the kernel interface (AlmaIF in the proposed approach) is fixed. This enables bug fixes and performance increases to existing kernels to happen invisibly to the end user. Still, it's likely a simple bitstream versioning scheme would be needed to allow users to apply updates when it is convenient to them.

Currently, two different bitstream distribution methods are supported by the implementation.

1) HTTP: The HTTP server is constructed as a simple directory-based structure with a JSON file at a directory root acting as an index and metadata storage. The bitstream files are automatically downloaded to a local cache when they are used for the first time.

2) AWS: This method utilizes AWS EC2 F1 instances. The bitstream is wrapped as Amazon FPGA Image (AFI). Only an AFI identifier is stored in the database, while the actual bitstream is stored in AWS servers, as a service provided by AWS with no extra charge. When the kernel is launched, the implementation calls AWS API to load the bitstream, and the actual bitstream is transferred to the FPGA via AWS sideband channels without being visible to the user instance.

V. Evaluation

To evaluate the programming flow and its portability, AFOCL is implemented to support four FPGAs: AMD Alveo U280 (for both Vitis platform shell and a custom Vivado shell), AWS EC2 F1 FPGAs (AMD Ultrascale+ VU9P), AMD PYNQ-Z1 (XC7Z020-1CLG400C) and Altera Arria 10 GX development kit (10AX115S2). The list of currently supported platforms is shown in Table I. The necessary drivers are created to communicate between the OpenCL implementation and accelerators implemented on the FPGA fabric (shown in Fig. 6). The host program can be kept completely vendor-agnostic and OpenCL standard-compliant and the FPGA reconfiguration happens automatically when kernels are launched.

A. Overhead Measurement

The kernels are created slightly differently for AFOCL shells than for the vendors' own OpenCL FPGA flows. Therefore there can be slight performance differences between AFOCL and the vendors' OpenCL flows. To measure only the overhead, a minimal vector addition program is used as a benchmark. The data starts from the host memory, from which it is moved to the device with *clEnqueueWriteBuffer*. Then, the kernel has to load in the data from the external memory and write the results back. To simplify the initial measurements, only a single external memory bank is used. Finally, the results are copied back to the host memory with *clEnqueueReadBuffer*. The results are shown in Table II.

TABLE I
Current FPGA Platform Support for AFOCL

FPGA	Comments
Alveo U280	AMD Vitis xilinx_u280_gen3x16_xdma_1_202211_1-platform
Alveo U280	Custom shell based on reconfigurable *Block Design Containers*
Altera Arria 10	Custom shell modified from a10_ref reference platform
AWS EC2 F1	XDMA shell F1.X.1.4 (v04261818)
Pynq Z1	Block design covering the entire FPGA

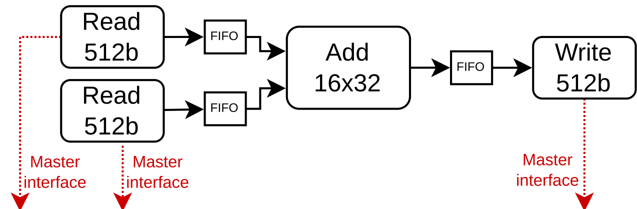

Fig. 9. Streaming dataflow implementation of vector addition where the computation has been decoupled from the external communication by the concurrent read and write tasks.

First, the comparison against the unoptimized OpenCL C kernel demonstrates the dire need for performance portability in FPGA OpenCL implementations. In this test, the simplest possible OpenCL C vector addition kernel is synthesized with the vendor FPGA flows. The proposed method shows 85× and 186× throughput improvement on Alveo U280 and Arria 10, respectively. This improvement comes purely from adopting a wider, burst-based access pattern to external memory. By default, the vendor flows synthesize hardware that performs single 32-bit accesses, whereas the provided template directs the hardware designer to develop a kernel using DMA-units and wide external memory interface present in the template.

The second comparison is made against a vectorized addition kernel. This modification to the OpenCL C sources is easy to make, since it is enough to use OpenCL C vector datatypes, after which the vendor HLS-tool generates the appropriately wider datapaths. Still, the proposed method is 3.7× and 12× faster, because the vendors' methods are still performing one wide memory access at a time.

For maximum performance, the vendors recommend using a streaming pattern in which the communication to the external memory is handled in a different concurrent task than the main computation. As shown in Fig. 9, the tasks are connected to each other with on-chip memory storage, which decouples the external communication from computation. In this design

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE II

RESULTS OF 32-BIT UNSIGNED ADDITION-KERNEL SYNTHESIZED WITH FPGA VENDOR TOOLS AND THE PROPOSED METHOD. THE RUNTIME RESULTS ARE CALCULATED WITH A PROBLEM SIZE OF 8 MILLION ELEMENTS

Altera Arria 10 GX	**Kernel execution time**	**Buffer transfer time**	**Clock frequency**	**Area (out of 394 420 ALMs)**
Unoptimized OpenCL C kernel	1468.8 ms	18.1 ms	324 MHz	5 290 (1.3%)
SIMD OpenCL C kernel	92.6 ms	18.3 ms	303 MHz	4 867 (1.2%)
Streaming dataflow kernel	7.3 ms	18.3 ms	341 MHz	8 222 (2.1%)
AFOCL	7.9 ms	31.0 ms	285 MHz	8 594 (2.2%)
AMD Alveo U280				**Area (out of 1 303 680 LUTs)**
Unoptimized OpenCL C kernel	681.8 ms	24.8 ms	554 MHz	2 310 (0.18%)
SIMD OpenCL C kernel	29.6 ms	24.7 ms	437 MHz	5 569 (0.43%)
Streaming dataflow kernel	8.0 ms	25.0 ms	523 MHz	6 602 (0.51%)
AFOCL (w/ Vitis platform shell)	8.0 ms	21.6 ms	266 MHz	16 396 (1.3%)
AWS F1 (AMD Ultrascale+ VU9P)				**Area (out of 1 180 984 LUTs)**
AFOCL	8.7 ms	45.9 ms	250 MHz	169 154 (14%)
AMD XC7Z020				**Area (out of 53200 LUTs)**
AFOCL	68.7 ms	302.4 ms	100 MHz	5 559 (10%)

method, the external memory accessing task can perform multiple memory accesses in rapid succession, as long as there is enough space in the intermediate storage. Compared to the proposed implementation, this type of kernel reaches the same performance on Alveo U280 and is 1.1× faster on Arria 10. We believe this convergence in performance is due all the methods being close to reaching the maximum memory bandwidth available on a single bank while mixing read and write-requests. This result makes sense in principle, since there is nothing fundamental provided by the proposed flow that would lead to different achievable performance. After all, the kernel optimization on FPGA is similar to hardware design in general, and it is possible to design similar hardware with the proposed method as the vendor OpenCL FPGA tools.

Comparison against the highest performing streaming pattern can be used to estimate the area and clock frequency overheads of the template. On Arria 10, AFOCL has only a small area overhead, but on Alveo U280, the overhead is more noticeable. AFOCL achieves slightly lower clock frequency on Arria 10 and significantly lower clock frequency on Alveo U280. The differences are due to the instantiated template which includes fixed, more generic components (DMA engines, command processor, and interconnects between these) compared to the streaming pattern-design. The differences are expected to diminish as the template is optimized further and when a more complex kernel is used, which is likely to pull down the clock frequency of the comparison target and to reduce the relative area usage of the fixed template components.

The proposed method implementing the vector addition is also evaluated on AWS F1 PCIe and XC7Z020 SoC FPGAs. The vendor methodologies are not compared against these devices, as from the previous comparisons with Arria 10 and Alveo U280 it can be reasonably extrapolated that the equivalent performance would be achievable for also these devices. The performance on AWS F1 matches closely the performance of a comparatively-sized Alveo U280. The area utilization seems much larger, but it is explained by the differences in the split to the static shell and reconfigurable region between the shells. For the smaller SoC FPGA, there are less FPGA resources available, which caused the vectorization factor of the kernel implementation to be reduced to 2, as opposed to a factor of 16 used in other kernel implementations. As a consequence, the performance is lower in proportion.

B. Scalability

In this test, we show how AFOCL could scale to enable the utilization of more of the FPGA resources and external memory ports. The previous benchmark utilized only a single external memory port. However, it is typical that modern FPGAs have multiple chips of DDR memory or stacks of HBM memory with multiple access ports. In this section we evaluate how the proposed method can further accelerate the original vector addition example by utilizing more HBM ports. The test is performed with Alveo U280 which has 8GB of HBM. The Vitis platform shell for Alveo U280 exposes 32 pseudo channels each with the size 256 MB.

The goal in utilizing the parallel memory banks of the Alveo's HBM memory is to keep it invisible to the host programmer. Therefore, the OpenCL host program can be kept identical to the one used in the original vector addition evaluation in Section V-A. The data is partitioned internally to the different banks by the driver, and collected back when the user reads the data back from the device.

By utilizing 30 HBM pseudo channels and ten replicated vector addition accelerators from Section V-A, we are able to improve the maximum throughput of the vector addition benchmark by a factor of 15×. If the amount of data is increased beyond the 8 million elements used in the original benchmark to take advantage of the larger capacity offered by the parallel HBM banks, the maximum throughput compared to the single-channel throughput is increased by a total factor of 24× up to 300 GB/s, which is already 65% of the total HBM bandwidth available.

While this scalability test demonstrates how the proposed method is able to scale up to utilize more of the FPGA and memory interface resources, it is not yet very generic, as it only works for workloads that are easily decomposed both in terms of input- and output-data, a.k.a embarassingly parallel workloads. Additionally, the split to different banks in the driver is inconvenient to combine to other kernel implementations that do not work with the split data, requiring additional copies in and out of device when changing to those

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE III

RESULTS OF 1-D 11-TAP CONVOLUTION KERNEL SYNTHESIZED BASED ON A VENDOR-PROVIDED EXAMPLE DESIGN AND AFOCL FOR ALVEO U280. THE RUNTIME RESULTS ARE GENERATED WITH THE PROBLEM SIZE OF 8 MILLION 32-BIT SIGNED INTEGER ELEMENTS

1D Convolution (FIR)	Kernel execution time	PCIe transfer time	Clock Frequency	Area (out of 1 303 680 LUTs)	DSPs (9024 total)
Unoptimized OpenCL C kernel	3752.0 ms	5.8 ms	500 MHz	4353 (0.33%)	3 (0.033%)
Shift register kernel	954.7 ms	5.8 ms	525 MHz	4799 (0.37%)	33 (0.37%)
Streaming dataflow kernel [9]	94.8 ms	5.9 ms	357.3 MHz	3338 (0.26%)	22 (0.24%)
AFOCL	5.5 ms	43.4 ms	211 MHz	18051 (1.4%)	8489 (94%)

TABLE IV

RESULTS OF 2-D 3 × 3 CONVOLUTION KERNEL FROM POLYBENCH [10] SYNTHESIZED WITH VENDOR TOOLS AND AFOCL FOR ALVEO U280. THE RUNTIME RESULTS ARE GENERATED WITH THE INPUT SIZE OF 2048 × 2048 32-BIT FLOATING POINT ELEMENTS

2D Convolution (polybench-conv2d)	Kernel execution time	Clock Frequency	Area (out of 1 303 680 LUTs)	DSPs (9024 total)
Unoptimized OpenCL C (Vitis HLS)	662.6 ms	500 MHz	5141 (0.39%)	29 (0.32%)
AFOCL	19.1 ms	272 MHz	19168 (1.5%)	50 (0.55%)

kernel types. A proper implementation of scalable kernels utilizing multiple HBM banks will require a hardware address translator component that automatically interleaves the traffic from PCIe and accelerators to different banks.

C. Convolution Kernels

Since the map-pattern of the vector addition benchmark is straightforward to scale, we want to evaluate the proposed method with slightly more complicated stencil-patterns to see whether there are issues with the proposed method that would only show up with more complicated kernels. For simplicity, we only use a single HBM port in these experiments. The results for an 11-tap 1-D convolution kernel are shown in Table III and for 3 × 3 2-D convolution kernel in Table IV. These comparisons are only implemented for the Alveo U280 FPGA. The AMD OpenCL results for 1-D convolution are based directly on a vendor-provided example streaming design [9]. The comparison target for 2-D convolution is Vitis HLS with the non-modified OpenCL C kernel *conv2d* from PolyBench [10]. The comparison targets demonstrate the performance one could reasonably expect porting OpenCL C kernels to vendor FPGA implementations with limited regard for hardware-specific optimizations.

As can be seen from the results, the proposed flow is able to deliver competitive performance on more complex kernels. The AFOCL kernels are manually vectorized in order to compute multiple convolution iterations in parallel, which explains the higher performance achieved by the proposed method. It is likely that similar optimization could be made with the vendor tooling, but it was not done for now, as in this case the comparison is made against the example design released by AMD and against Vitis HLS compiling unoptimized portable OpenCL C kernels. The reason for the noticeable slowdown in PCIe transfer time in the 1-D convolution benchmark is still unknown, but should be unrelated to the computation happening in the programmable region.

VI. DISCUSSION

The proposed approach is naturally not without some limitations and drawbacks. In this section we discuss some of the most apparent key concerns to assure that they do not prevent the viability of the proposed approach.

A. Managing the Database Size

In the future, some DBKs will have multiple implementations for even the same FPGA, with different specialization parameters (for example, a fixed work-group size), or in different combinations of other kernels. If the same DBK is implemented multiple times in different bitstreams, the driver's choice of the bitstream could be made using metadata related to DBK performance, and which other kernels the user has launched for execution (which would require deferred execution and a scheduling decision).

Since there are likely to be thousands of different built-in kernels, dozens of different FPGAs, and when counting different kernel fusions, combinations and specializations, the total size of the database might grow to be enormous. This could become a problem in the future. However, we do not yet have a realistic view on how many total bitstreams will be needed to build an effective and generally usable complete system within an application domain. In the future, a full domain-level test of the platform needs to be created to evaluate this issue.

The remote servers help with the rapidly growing size of the total storage required. The actual storage can be inexpensive and globally distributed, from which bitstreams are fetched only as they are needed. This reduces the storage requirements of the FPGA users, who only hold a cached copy of the bitstreams they have used. While the size of the bitstreams (10–100 s of megabytes) is larger than some software binaries, the storage and distribution of large amounts of data can nowadays be done cost-effectively.

The current implementation of the proposed flow only uses FPGA shells with a single reconfigurable region. If more regions are to be used, it will likely become important to make the partial bitstreams *relocatable*, to prevent having to resynthesize the same kernel implementations for every region.

Bitstream compression was not yet utilized in this implementation. Even a simple compression scheme could

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

provide a significant benefit on the storage and bandwidth requirements. And, with on-FPGA decompression, it could even reduce the reconfiguration latency by requiring less off-chip bandwith.

B. Porting the Bitstream Database

One major disadvantage of using bitstream as the distribution method is that bitstreams are not compatible between different FPGAs. This is different to software binary distribution, in which just a few instruction-set architectures are enough to cover a vast majority of the CPUs used in practice (x86, ARM, RISC-V). While porting to a completely new FPGA device requires moderate effort, once the porting has been done once, it then becomes easier to port new built-in kernel implementations to already supported platforms.

As described in Section IV-C, the hardware development of built-in kernel implementations requires expertise and effort. It would be desirable to reuse that effort even when porting the database to different devices. To accomplish this, automated scripts could be created to port the RTL-level kernel implementations to different FPGA types, as long as the FPGAs are close enough to each other in features.

At some point, the kernel implementations need to be scaled up and down if the FPGA resources differ significantly. For example, the vectorization level of the kernels and the number of accelerator instances can be made adjustable, after which the RTL and FPGA bitstream are regenerated.

VII. Related Work

There are a number of previous approaches to make the programming of FPGAs easier. The method we propose combines ideas from two major directions.

1) Exposing a reconfigurable region for OpenCL kernel-based programming.
2) Pre-synthesizing bitstreams to be used for a library-based acceleration.

While the FPGA has been abstracted as a generic OpenCL device before, to our best knowledge, it has not been combined with the approach of pre-synthesizing the OpenCL kernels beforehand, and then distributing and reconfiguring the FPGA completely behind-the-scenes. Finally, the proposed approach uses the standard OpenCL built-in kernel API to launch tasks, which is not something that has been done by any of the following works listed. These aspects enable the clear separation of responsibilities between different roles shown in Fig. 4, which is a novel approach to programming FPGAs with OpenCL.

A. OpenCL on FPGAs

In addition to the FPGA vendor OpenCL implementations described in Section II-B, other FPGA OpenCL implementations have been created.

Rodriques-Canal's Controller framework [11] supports the dynamic partial reconfiguration of FPGAs as part of a full heterogeneous parallel programming model. They support pre-emption [12] to dynamically schedule tasks with different priorities. Their work is close in intention to the proposed approach, since they also recognize the importance of easy integration of FPGAs to heterogeneous systems. However, they do not use a standardized API for the end user to control the system, but instead have developed their own Controller API, which supports CPU, GPU and FPGA devices.

PCIeHLS [13], ZUCL [14] and FOS [15] are based on a highly advanced shell-based platform for AMD UltraScale+ FPGAs. Their shell supports multiple reconfigurable regions with relocatable partial bitstreams, enabling them to use the same partial bitstream in different regions. Their flow allows a single kernel to utilize multiple regions to scale the area-performance tradeoff based on the total device utilization. While they recognize the benefit of making the system easily controllable by non-experienced users, they do not implement the standard OpenCL runtime API to control the platform, but instead use their own simpler task-based API to launch kernels. Compared to our flow, their method is meant for only a single FPGA type, while ours is implemented on FPGAs from multiple different vendors and FPGA types. Because of the relocatable partial bitstreams, their work could potentially be used as a basis for a bitstream distribution framework, however in their work, they did not yet suggest or implement a mechanism for bitstream distribution.

SOFF [16] is an OpenCL C HLS compiler that can achieve competitive performance compared to FPGA vendor OpenCL implementations. They target Altera FPGAs, and use SnuCL as the OpenCL runtime system. As their work is more focused on the efficient HLS of OpenCL kernels, whereas our work focuses on the integration of kernels to OpenCL runtime invisibly to the end user, their work does not overlap with our work in a significant way. Their HLS compiler is a highly interesting tool that could potentially be integrated to our platform to enable the generation of built-in kernel implementations out of software kernels for easier population of the bitstream database.

Owaida et al. [17] introduce a tool for automatically converting OpenCL programs to FPGA accelerators. They perform optimizations to the OpenCL input, and target a structured hardware accelerator template to provide a consistent optimization target. They convert memory-mapped accesses to streaming-style accesses automatically as part of their framework, similar to the *streaming dataflow* access-type of our proposed hardware template. Similar to SOFF, they concentrate on the automatic synthesis of OpenCL kernels.

UT-OCL [18] is a research platform for OpenCL on SoC FPGA platforms. They utilize small MicroBlaze soft processors to implement the compute kernels and system components, making their FPGA platform highly flexible. They also suggest an implementation for the communication with the OpenCL driver to be based on a soft processor interface, similar to the command processor template proposed by us. However, they do not utilize partial reconfiguration and bitstream distribution to execute the kernels, but instead execute the kernels by compiling them for scalar RISC soft processors, which limits the achievable performance.

B. FPGA Overlays

There is a large amount of research on FPGA overlays [19], [20], [21], [22], [23], [24] which circumvent the long FPGA

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

synthesis times by offering a coarser compilation target, which the application is then compiled/mapped to. In our work, we use a slightly different, shell-based, approach. We do not build our accelerators on any intermediate fabric, but instead reconfigure the entire region with a custom pre-optimized kernel implementation. This avoids the performance and area overhead inherent to any overlay-based approaches. With our platform, the kernel implementation can utilize the fine granularity of the FPGA fabric, increasing the level of specialization available.

C. Presynthesized Hard Macros

The following works take advantage of pre-synthesizing parts of the FPGA configuration, but do not operate at the level where partial bitstreams correspond to full OpenCL kernels.

SYLVA [25] is a synthesis framework based on pre-synthesized blocks that are stitched together to form complete computation pipelines. It supports ASIC, FPGA and CGRA backends.

HMFlow [26] is a similar macro-based approach as SYLVA targeting AMD FPGAs. Their goal is in speeding the FPGA tool runtime by attempting to use pre-optimized *hard macros* which are stitched together.

Our proposed work operates at a much coarser granularity than the two works introduced here, since we work at a level of pre-generated bitstreams, which are downloaded and reconfigured with no mapping step. The works introduced here are aiming to simplify the synthesis process by mapping to coarser-grain primitives rather than FPGA resources directly.

D. Presynthesized Bitstreams

Blaze [27] is an open-source platform for integrating pre-synthesized FPGA bitstreams into Apache Spark and Hadoop YARN. In principle, their approach is similar to ours, since they do not focus on kernel compilation or FPGA details, but instead figure out a way to abstract FPGAs in a higher-level framework with pre-generated bitstreams.

ViTAL [28] is an FPGA virtualization framework based on fine-grained pre-synthesized partial bitstreams. The framework allows automatic split of the application to multiple FPGAs each with a number of identical reconfigurable regions. By a careful construction of the overlay, they perform the partial FPGA bitstream generation ahead of time, but are able to defer the resource allocation to the regions to runtime, which gives them a large amount of flexibility in supporting multiple applications and multiple FPGAs without the fragmentation issues inherent to overlays with multiple reconfigurable regions.

AmorphOS [29] is a quite similar multiple-region-based shell approach with a number of pre-generated partial bitstreams to virtualize the FPGA resources spatially and temporally between multiple applications. To maximize throughput they support an automated toolflow to then synthesize all of the reconfigurable regions into a single bitstream, which eliminates the fragmentation. Conceptually, this is somewhat similar to our database having multiple built-in kernel implementations in the same FPGA bitstream.

The major difference compared to our work in all three introduced works is our use of standard OpenCL API to control the platform. The focus of our work has been to see how far the standard OpenCL can go in utilizing FPGAs in a dynamic, cross-vendor, way. We believe adherence to an open standard aids in the portability and longevity of our approach. However, a possible integration of our work to some of the related work would be a highly interesting experiment, since we could take advantage of the previous, advanced research on dynamic FPGA platforms.

VIII. Conclusion

In this work, we proposed an automated FPGA bitstream distribution and management platform based on the cross-vendor standard OpenCL API. The proposed approach creates a clean abstraction between the hardware developers creating the acceleration IPs for the database, and the software developers who can call computation kernels as they would for any OpenCL device. This makes it easier to integrate FPGAs to higher-level frameworks via OpenCL. The framework facilitates a reuse of hardware development effort by remote bitstream distribution to the end users which we demonstrated with an on-premise HTTP server installation and by utilizing the AWS.

We demonstrated that similar performance to the FPGA vendor OpenCL implementations is achievable, showing that the overhead of the proposed, more portable method is negligible. We showed that the proposed method is able to support different memory access patterns and multi-ported external memories efficiently. Cross-vendor and cross-device type -features of the framework were shown by implementing the framework on Altera PCIe FPGA, AMD PCIe and SoC FPGAs.

References

[1] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2018, pp. 578–594.

[2] T. Leppänen, J. Multanen, L. Leppänen, and P. Jääskeläinen, "AFOCL: Portable OpenCL programming of FPGAs via automated built-in kernel management," in *Proc. IEEE Nordic Circuits Syst. Conf. (NorCAS)*, Oct. 2023, pp. 1–7.

[3] Khronos® OpenCL Working Group. (2024). *The OpenCL™ Specification V3.0.16*. Accessed: 5, Apr. 2024. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

[4] T. Leppänen, A. Lotvonen, and P. Jääskeläinen, "Cross-vendor programming abstraction for diverse heterogeneous platforms," *Frontiers Comput. Sci.*, vol. 4, Oct. 2022, Art. no. 945652. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fcomp.2022.945652

[5] H. Linjamäki. (2023). *CL_KHR_Defined_Builtin_Kernels*. Accessed: Mar. 19, 2024. [Online]. Available: https://github.com/KhronosGroup/OpenCL-Docs/pull/1007

[6] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1014–1029, May 2021, doi: 10.1109/TPDS.2020.3039409.

[7] T. Leppänen, A. Lotvonen, P. Mousouliotis, J. Multanen, G. Keramidas, and P. Jääskeläinen, "Efficient OpenCL system integration of non-blocking FPGA accelerators," *Microprocess. Microsyst.*, vol. 97, Mar. 2023, Art. no. 104772, doi: 10.1016/j.micpro.2023.104772.

[8] K. Hepola, J. Multanen, and P. Jääskeläinen, "OpenASIP 2.0: Co-design toolset for RISC-V application-specific instruction-set processors," in *Proc. IEEE 33rd Int. Conf. Application-Specific Syst., Architectures Processors (ASAP)*, Jul. 2022, pp. 161–165.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[9] Xilinx. (2021). *Streaming Lab*. Accessed: Apr. 5, 2024. [Online]. Available: https://xilinx.github.io/xup_compute_acceleration/streaming_lab.html

[10] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. Innov. Parallel Comput. (InPar)*, May 2012, pp. 1–10.

[11] G. Rodriguez-Canal, Y. Torres, F. J. Andújar, and A. Gonzalez-Escribano, "Efficient heterogeneous programming with FPGAs using the controller model," *J. Supercomputing*, vol. 77, no. 12, pp. 13995–14010, Dec. 2021. Accessed: 10.1007/s11227-021-03792-7.

[12] G. Rodriguez-Canal, N. Brown, Y. Torres, and A. Gonzalez-Escribano, "Task-based preemptive scheduling on FPGAs leveraging partial reconfiguration," *Concurrency Comput., Pract. Exper.*, vol. 35, no. 25, p. e7867, Nov. 2023, doi: 10.1002/cpe.7867.

[13] M. Vesper, D. Kocha, and K. Phama, "PCIeHLS: An OpenCL HLS framework," in *Proc. 4th Int. Workshop FPGAs Softw. Programmers (FSP)*, Sep. 2017, pp. 1–6.

[14] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch, "ZUCL: A Zynq UltraScale+ framework for OpenCL HLS applications," in *Proc. FSP Workshop ; 5th Int. Workshop FPGAs Softw. Programmers*, Aug. 2018, pp. 1–9.

[15] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "FOS: A modular FPGA operating system for dynamic workloads," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 4, pp. 1–28, Sep. 2020, doi: 10.1145/3405794.

[16] G. Jo, H. Kim, J. Lee, and J. Lee, "SOFF: An OpenCL high-level synthesis framework for FPGAs," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 295–308.

[17] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *Proc. IEEE 19th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, May 2011, pp. 186–193.

[18] V. Mirian and P. Chow, "UT-OCL: An OpenCL framework for embedded systems using Xilinx FPGAs," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Riviera Maya, Mexico, Dec. 2015, pp. 1–6.

[19] A. Brant and G. G. F. Lemieux, "ZUMA: An open FPGA overlay architecture," in *Proc. IEEE 20th Int. Symp. Field-Programmable Custom Comput. Mach.*, Apr. 2012, pp. 93–96.

[20] J. Coole and G. Stitt, "Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts," *IEEE Micro*, vol. 34, no. 1, pp. 42–53, Jan. 2014.

[21] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Coarse grained FPGA overlay for rapid just-in-time accelerator compilation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 6, pp. 1478–1490, Jun. 2022.

[22] D. Capalija and T. S. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–8.

[23] S. Ma, Z. Aklah, and D. Andrews, "Just in time assembly of accelerators," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2016, pp. 173–178, doi: 10.1145/2847263.2847341.

[24] D. Capalija and T. S. Abdelrahman, "Towards synthesis-free JIT compilation to commodity FPGAs," in *Proc. IEEE 19th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, May 2011, pp. 202–205.

[25] S. Li, N. Farahini, A. Hemani, K. Rosvall, and I. Sander, "System level synthesis of hardware for DSP applications using pre-characterized function implementations," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Sep. 2013, pp. 1–10.

[26] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping," in *Proc. IEEE 19th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, May 2011, pp. 117–124.

[27] M. Huang et al., "Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale," in *Proc. 7th ACM Symp. Cloud Comput.*, vol. 99. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 456–469, doi: 10.1145/2987550.2987569.

[28] Y. Zha and J. Li, "Virtualizing FPGAs in the cloud," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, 2020, pp. 845–858, doi: 10.1145/3373376.3378491.

[29] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS," in *Proc. 13th USENIX Symp. Operating Syst. Design Implement.*, 2018, pp. 107–127.