AFOCL: Portable OpenCL Programming of FPGAs via Automated Built-in Kernel Management

Topi Leppänen, Joonas Multanen, Leevi Leppänen, Pekka Jääskeläinen
Faculty of Information Technology and Communication Sciences
Tampere University
Tampere, Finland

***Abstract*—OpenCL provides a consistent programming model across CPUs, GPUs, and FPGAs. However, to get reasonable performance out of FPGAs, OpenCL programs created for other platforms need to be modified. These modifications are often vendor-specific, limiting the portability of OpenCL programs between devices from different vendors.**

In this paper, we propose AFOCL: a cross-vendor portable programming methodology for FPGAs based on standard OpenCL and a database of bitstreams. It is based on the *built-in kernel*-abstraction introduced in OpenCL v1.2. FPGA reconfiguration is handled automatically by the proposed OpenCL runtime and is invisible to the software programmer.

To demonstrate the cross-vendor portability of the method, it is implemented for a PCIe FPGA card from both AMD and Intel. Templates for efficient dataflow-based kernels are created, which can be extended and tailored for each built-in kernel implementation. With a simple evaluation kernel, the runtimes are 85x and 186x faster than an unoptimized OpenCL C kernel implemented with FPGA vendor tooling, AMD and Intel respectively. Against hand-optimized kernel implementations created with vendor tools, the proposed method reaches the same performance as AMD and is only 1.1x slower than Intel, due to limited clock frequency optimization of the template. Thus, the method provides a flexible cross-vendor programming model for FPGAs with competitive performance. The method enables splitting the roles of software developers, who no longer need to concern themselves with FPGA-specific details, and FPGA hardware developers who populate the database. The proposed method is released in open source and integrated into a popular OpenCL implementation PoCL.

***Index Terms*—OpenCL, built-in kernels, FPGA, performance portability, multi-vendor**

I. Introduction

The slowing growth of CPU performance has led to a boom in heterogeneous computing architectures. These devices include GPUs, DSPs, ASICs, and FPGAs. Traditionally, the different device types have had their own programming models, which means that higher-level programs using them need to be modified for each new device type. OpenCL [1] is an open standard aiming to provide a write-once-use-everywhere model of heterogeneous platform programming. It relies on OpenCL implementations to hide the device-specific details from the higher-level programmer.

OpenCL has been found to be a useful programming abstraction for FPGAs [2]–[5], which are very generic devices able to implement most logic circuits, with the trade-off being that they are less efficient than the corresponding logic circuit as a fixed-function ASIC. Still, due to their flexibility, FPGAs have been deployed as part of commercial cloud computing systems [6], [7]. Since FPGAs need a circuit description in order to be programmed, compiling a higher-level program automatically to a circuit description has been found to raise programmer productivity [8]. This automated lowering/compilation-step is called *High-Level Synthesis* (HLS).

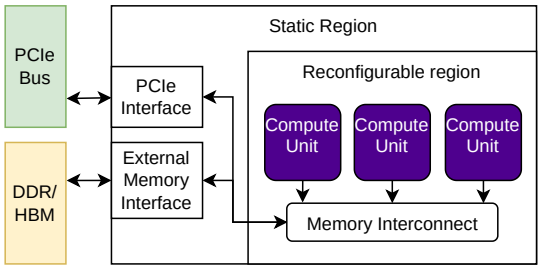

Fig. 1: Generic PCIe FPGA platform for OpenCL

FPGA vendors AMD and Intel both offer OpenCL-based HLS tools for their FPGAs. While these OpenCL implementations have proven to be useful in many cases, they are still quite limited. To get the best performance of a specific kernel, the OpenCL kernel code used for CPUs or GPUs needs to be heavily modified when it is ported for FPGA devices [9]. These optimizations require significant tool-specific knowledge since they can include heavy transformations to make the kernel more dataflow-oriented, vendor-specific extensions, libraries, pragmas, and attributes. The vendor-specific modifications can cause *vendor lock-in*, where the application becomes so tied to specific tools that changing the device vendor is no longer feasible. Additionally, the vendor implementations require the user to trigger FPGA reconfiguration by passing the bitstream to the OpenCL implementation via the host program.

In this work, we propose a more portable and flexible OpenCL programming model called AFOCL. Instead of asking the user to first compile the bitstream and then program it to the FPGA in an explicit call, we propose a method in which the OpenCL implementation is connected to a database of precompiled bitstreams and where the implementation manages the FPGA reconfiguration in the ways it sees fit. The user

979-8-3503-3757-0/23/$31.00 ©2023 IEEE

utilizes the standard OpenCL built-in kernel API (available since version 1.2 of the standard) to get the functionality they want and does not need to handle the bitstream binaries, or anything FPGA vendor-specific.

A separate database of OpenCL built-in kernel implementations decouples the kernel development from host software development. The kernel implementations are created by hardware experts and synthesized only once. The OpenCL implementation completely manages the FPGA reconfiguration behind-the-scenes by fetching the bitstreams from the database and reconfiguring the FPGA when the kernels are enqueued. The database contains the same built-in kernel implemented for different FPGAs to enable cross-vendor portability.

The OpenCL software developer can keep their host application completely standard-conformant, which means that they can develop their applications even with a different OpenCL implementation and device (e.g. desktop CPU), as long as that OpenCL implementation contains the implementations for the same set of built-in kernels.

Thus, the proposed method will help in clarifying and isolating the roles of software and hardware developers. The hardware developers will be responsible for adding new kernels to the database. The software developers can focus on writing host applications without any FPGA vendor-specific code.

To assist the hardware developers in creating new built-in kernel implementations to the database, we provide a *command processor template* which is distributed alongside the other sources. The command processor template includes a small controller to make the communication to the provided OpenCL driver consistent, a configurable number of DMA engines for efficient data movement, and the acceleration logic itself in separate IPs.

To evaluate the proposed methodology, the runtime is implemented on two PCIe FPGAs, one from Intel and one from AMD. As a demonstration of the cross-vendor portability, the host program used to control the different FPGAs can be kept exactly the same. The overhead measurements show that the proposed method can give significant performance improvement over unoptimized OpenCL C kernels, and adds only a small overhead compared to hand-optimized kernel implementations created with corresponding vendor tooling.

Summarizing, this paper provides the following contributions:

- AFOCL: a cross-vendor portable programming model for FPGAs based on standard OpenCL.
- Abstraction of all FPGA-specific details from software developers via automated FPGA management.
- Methodology for reusing hardware development effort via shared bitstream database.

The source code of the proposed framework including the command processor template and the OpenCL implementation is released in open-source in order to progress the field of vendor-independent FPGA programming.[1]

[1]Source code available at: https://github.com/cpc/AFOCL.

II. Related Work

SYLVA [10] uses a library of *macros*, which are pre-synthesized function implementations which they tie together to generate complete specialized systems for ASIC, FPGA and CGRA backends. They show how the use of macros can save expensive hardware development effort and offer an easier target for high-level synthesis tools compared to lower-level RTL.

ZUCL [11] is a framework for deploying OpenCL applications for the Xilinx UltraScale+ MPSoC platform. They support multiple reconfigurable slots on the FPGA which can be dynamically reconfigured with pre-synthesized *relocatable* OpenCL kernels. The support for relocatable bitstreams allows them to reuse the once synthesized bitstream in different, identical in structure, reconfigurable slots. They recognize the importance of decoupling the accelerator compilation and the static platform shell, which allows independent development and updating of each component. In their later work FOS (*FPGA Operating System*) [12], they manage FPGA resources *spatially* in an analogous manner to generic *operating system's* (OS) temporal sharing to distribute the resources for OpenCL kernels at runtime based on application characteristics.

PLD [13] is an FPGA compilation framework that supports different compilation paths based on the optimization level set by the user. They support compiling the design for a set of reconfigurable regions connected by an NoC similar to HiPR [14]. Park et al. [15] extend PLD to support hierarchical partial reconfiguration where reconfigurable slots can be recombined to form double- and quad-sized slots.

Rodriquez-Canal et al. [16] support dynamic partial reconfiguration of OpenCL kernels in their Controller framework. They support platforms with multiple dynamically reconfigurable slots and have support for reconfiguring them separately based on the application's needs.

UT-OCL [17] is an OpenCL platform implemented with hardware and software components to execute OpenCL kernels on FPGAs. It includes a small MicroBlaze core for each kernel implementation to handle communication with the OpenCL driver, which is quite similar to the proposed *command processor template*.

While many of the earlier works described above have implemented automated FPGA reconfiguration based on the OpenCL kernel which the user wants to launch, according to the authors' best knowledge, the proposed method is the first work suggesting a *database of OpenCL built-in kernels* combined together with the automated FPGA management. Additionally, the previous works' discussed here have focused on only AMD FPGAs, whereas the proposed method suggests a consistent vendor-independent programming model validated on devices from both AMD and Intel.

While especially SYLVA [10] and FOS [12] seem very close in intention to the proposed method, they do not propose the pre-compilation and distribution of bitstreams. Additionally, the proposed method explicitly stays within standard OpenCL-compliant host applications, which keeps them completely

portable and vendor-independent, which was not explicitly addressed in the related works described above.

III. Automated Built-in Kernel Management

The proposed method uses a similar overlay structure to FPGA vendor OpenCL implementations seen in Fig 1. It consists of a static platform region for handling the connectivity to external interfaces, which is loaded onto the FPGA device at power-on time, and a reconfigurable region which is reprogrammed with the built-in kernel implementations.

AFOCL consists of two major components: A database of pre-synthesized bitstreams that implement the built-in kernels and a runtime driver which fetches bitstreams from the database and automatically reconfigures the FPGA.

A. Bitstream Database

At the core of the proposed method is a pre-synthesized database of built-in kernel implementations. Every bitstream contains an implementation for one or more built-in kernels. According to the OpenCL specification [1], built-in kernels are kernels with the behavior defined by the OpenCL implementation and only identified by their name.

In order to make sure that the functionality of the pre-synthesized built-in kernel matches what the software developer expects, the use of a *built-in kernel registry* such as the one proposed in [18] is recommended. The built-in kernel registry contains the mapping between the built-in kernel name and its functionality but does not define or contain anything about where or how the built-in kernel will be implemented (CPU, GPU, FPGA, etc.). When the kernel implementation in the bitstream database follows the specification set out in the built-in kernel registry, there is a guarantee that the built-in kernel implementation will have *exactly the same* behavior as the registry entry. This also provides a consistent program-lowering target for higher-level tools and frameworks. The difference between the built-in kernel registry and the proposed database is further illustrated in Fig. 2.

Supporting multiple different FPGA devices from different vendors in the same database is possible, as long as every FPGA device has its own bitstream entry. Therefore, a single built-in kernel has multiple implementations in the database for different FPGA devices.

One built-in kernel can be a *fusion* of two lower-level built-in kernels. For example, *multiply-accumulate*-operation is a fusion of *multiply* and *addition* operations. Alternatively, these two kernels can be implemented as a *combination* of accelerators on the same bitstream as independently launchable built-in kernels. This can be seen in Fig. 3 which includes accelerators A and B both controlled by the same command processor. Currently, there is no automated support to fuse or combine built-in kernels, so these kinds of optimizations need to be performed manually.

Different FPGA device types, kernel fusions, combinations, and possible kernel specializations can make the total database size large, thus the effort in expanding it for new FPGAs could become significant. However, since the centralized database allows many users to reuse and share common bitstreams, the method could actually reduce the *total* number of bitstream generation runs globally. Additionally, having open and easily available bitstream databases could increase the use of FPGA devices, which are commonly known to be hard to utilize.

Another interesting possibility enabled by the shared database is that updating it with more performant built-in kernel implementations can improve the application runtimes of potentially thousands of users who have opted-in for updates. This makes it worth it to spend significant time and effort to optimize the most popular built-in kernels.

B. Adding New Built-in Kernel Implementations

Adding new built-in kernel implementations is an important part of the proposed method. This is a step that requires hardware development expertise since the kernel implementations are meant to be highly optimized.

In order to swap kernels in the reconfigurable region, the kernels need to have a commonly understood interface. For this, we use AlmaIF [19] accelerator interface. It is a memory-mapped interface with configuration and control registers, a region for a ring buffer of command packets, and a region for dynamically allocatable data. Every kernel implementation will need to implement the interface in order for the runtime driver to be able to communicate with the kernels.

The built-in kernel developer can use whatever technique they like to implement the built-in kernel including RTL, HLS, and soft processors. The only requirement is compliance with the AlmaIF memory-mapped interface specification.

One of the ways to implement the AlmaIF interface is to have a separate *command processor* which handles the interfacing to the driver and a separate accelerator(s) that performs the core of the computation. We provide a *command processor template* shown in Fig. 3, which can be modified by the built-in kernel developer.

Efficient data movement between the accelerator and the external memory is highly important for good runtime performance. The command processor template has support for three different memory access types:

1) *Random access to external memory*. The most generic one which will work with most applications. Recommended for large data amounts which require random access.
2) *Random access to on-chip memory*. Similarly very generic and easy to implement, but limited by the amount of on-chip RAM, and the transfer speed of the slave memory interface. Recommended for small data amounts.
3) *Streaming dataflow*. Burst access between external memory and on-chip FIFOs. Most efficient utilization of external bandwidth, but is possibly difficult to implement for all types of kernels. Recommended for large data amounts where the access patterns are straightforward.

The memory access types can be freely combined even in the same built-in kernel implementation. An example template implementing all three memory access types is released in

Built-in Kernel Registry

Kernel name	Kernel behavior

add.i32	kernel void add_i32 (global int* a, global int* b, global int* c) {
	int i = get_global_id(0);
	c[i] = a[i] + b[i];
	}

mul.i32	kernel void mul_i32 (global int* a, global int* b, global int* c) {
	int i = get_global_id(0);
	c[i] = a[i] * b[i];
	}

Bitstream Database of Built-in Kernels

Bitstream filename	FPGA device	Implements built-in kernels		
add.xclbin	Alveo U280	add.i32		
addmul_combination.xclbin	Alveo U280	add.i32	mul.i32	
add.rbf	Arria 10 GX	add.i32		
mul.rbf	Arria 10 GX		mul.i32	
addmul_fused.rbf	Arria 10 GX			mul.add.i32

Fig. 2: Simplified illustrations of the built-in kernel registry from [18] and the built-in kernel database (proposed). The built-in kernel names used in the database are exactly the same as in the registry to enforce the link between the kernel behavior and implementation.

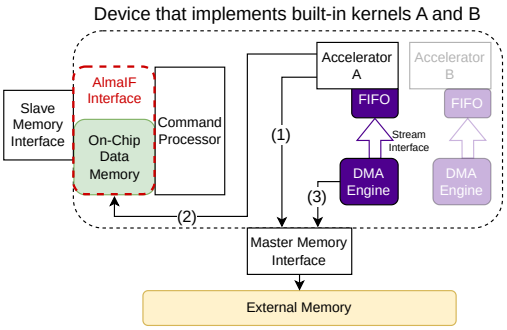

Fig. 3: Command processor template with the three different memory access types. (1) *Random access to external memory* (2) *Random access to on-chip memory* (3) *Streaming dataflow*.

open-source to facilitate the easy addition of new built-in kernel implementations. The command processor template is FPGA vendor and device-specific and therefore may require moderate one-time porting efforts when changing the device type or especially the FPGA vendor. At the time of writing, the template has been ported for Intel Arria 10 GX and Xilinx Alveo U280 FPGAs.

The provided command processor template likely needs small modifications for each new built-in kernel. Typically only the interfacing between the command processor, DMA engines, and the acceleration IP needs customization since it depends on the number and type of kernel arguments.

C. FPGA Reconfiguration

To reconfigure the FPGA, the proposed method includes a runtime driver component inside the OpenCL implementation that connects to the bitstream database. At the initialization time, the runtime driver will parse the database index. After this, the driver knows which built-in kernels it can implement, and will report this list in a standard OpenCL *CL_DEVICE_BUILT_IN_KERNELS* device info query as defined in OpenCL specification [1].

The runtime driver will reconfigure the FPGA when the kernel is enqueued. This enables more flexible use of FPGA compared to vendor (AMD, Intel) OpenCL implementations as shown in Fig. 4. The vendor OpenCL implementations tie the FPGA reconfiguration to the program object construction with *clCreateProgramWithBinary*-call. This can complicate the host programs somewhat since the program objects cannot be constructed beforehand for kernels in different bitstreams. Alternatively, the user must know beforehand which kernels are used in the same application and synthesize all those into the same bitstream (if they fit). The proposed method will automatically reconfigure the FPGA when the user launches a kernel that is not already configured on the FPGA.

IV. Evaluation

To demonstrate the cross-vendor portability of the proposed method and to assess the inevitable overheads that come with a more generic method, we implemented AFOCL for two FPGAs from two different vendors. The FPGAs used for the evaluation are AMD/Xilinx Alveo U280 and Intel Arria 10 GX development kit with 10AX115S2 FPGA.

In all of the evaluations, the host program used for the proposed method is kept the same when switching between FPGAs from different vendors. This already demonstrates the ability of the proposed method to hide vendor-specific details from the software developer.

A. Command Processor Overhead

The overhead is measured with a vector addition benchmark. A simple benchmark is chosen to separate the overhead measurement from the kernel implementation technique. Since the application is so simple, it is clear that the runtime performance of the design depends on mostly two factors: efficient utilization of external bandwidth, and the clock frequency of the design.

The accelerator created with the proposed method consists of the command processor template (Fig. 3) with three DMA engines, two for loading the data and one for writing. The computation kernel is instantiated as a separate accelerator with streaming-type interfaces. The DMA engines convert between the memory-mapped and the streaming interfaces, which means that externally the accelerator looks to be simply doing memory-mapped burst accesses to external memory.

Table I contains runtime, clock frequency, and area comparison of the proposed method against three different variations generated with the vendor methodologies. The input data starts

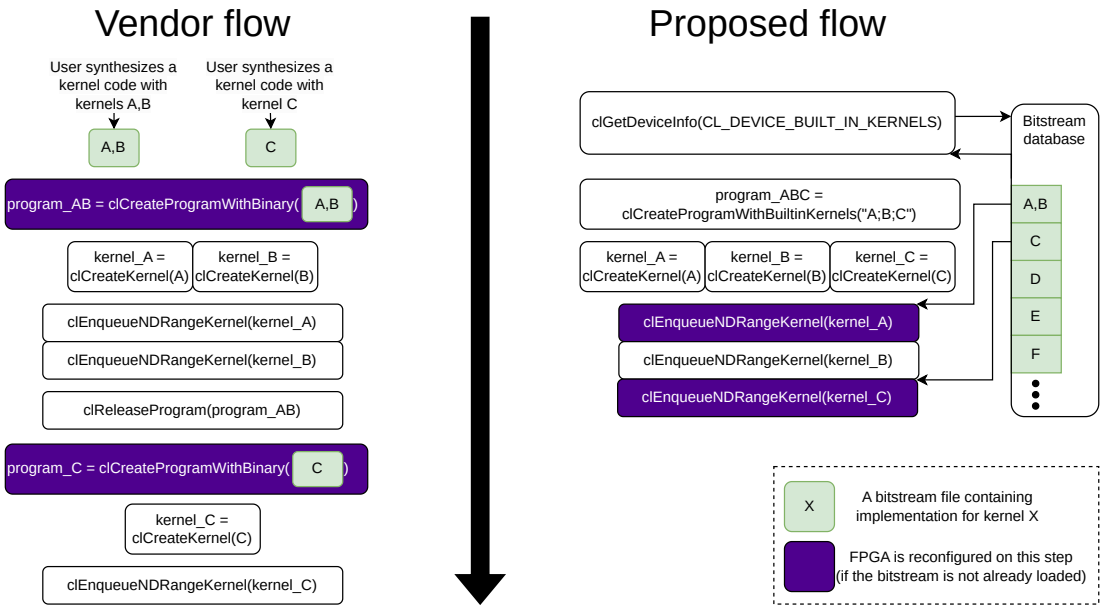

Fig. 4: The method for launching kernels from different bitstreams with the vendor flow (left) and the enqueue-time reconfiguration of the proposed method (right). The vendor methods allow only one program object to be active simultaneously, since the *clCreateProgramWithBinary*-call reconfigures the FPGA. The proposed method automatically reconfigures the FPGA only after the commands have been enqueued.

TABLE I: Results of 32-bit unsigned addition-kernel synthesized with FPGA vendor methodologies and the proposed method. The runtime results are calculated with the problem size of 8 million elements.

Intel Arria 10 GX	Kernel execution time	PCIe 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	4 866.5 (1.2%)
Streaming dataflow kernel	7.3 ms	18.3 ms	341 MHz	8 222 (2.1%)
Proposed Method (AFOCL)	7.9 ms	31.0 ms	285 MHz	8 593.5 (2.2%)
Xilinx 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%)
Proposed Method (AFOCL)	8.0 ms	21.6 ms	266 MHz	16 396 (1.3%)

TABLE II: Kernel launch latencies and the dynamic reconfiguration times. Streaming dataflow kernel from Table I was used as a baseline for vendor methodologies.

Intel Arria 10 GX	Kernel launch time	Reconfiguration time
Vendor OpenCL flow	127 µs	3.7 s
Proposed Method	240 µs	3.3 s
Xilinx Alveo U280		
Vendor OpenCL flow	48 µs	3.7 s
Proposed Method	148 µs	3.7 s

from and ends up to an external DDR or HBM memory of the PCIe FPGA card. Transfers over the PCIe bus from host CPU to PCIe card DDR/HBM are not included in the kernel execution time. Only a single memory port between FPGA and external memory was used to simplify the comparison.

First, an unoptimized implementation of the kernel consists of a bare-minimum description of the OpenCL C kernel performing 32-bit element-wise addition of two arrays and storing the result to an output array. The comparison against the unoptimized OpenCL C kernel is interesting since it is the level of abstraction that we expect the software developer to work on. The software developer is mostly only interested in what computation the kernel functionally performs. Therefore, a novice developer using our method will get 85x and 186x faster vector addition compared to the unoptimized OpenCL C kernel synthesized with the vendor methods on Alveo U280 and Arria 10 GX, respectively.

Explicit vectorization with the use of OpenCL vector datatypes is easy to implement for this simple kernel. For this evaluation, a 16-element vector datatype was chosen. This is already significantly faster than the unoptimized kernel due to the wide datapath of the accelerator and wider external memory accesses. Even still, the proposed method is shown

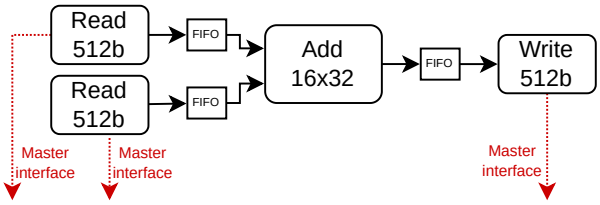

Fig. 5: Streaming dataflow implementation of vector addition.

to be 3.7x and 12x faster, respectively.

Finally, a streaming dataflow kernel was constructed that uses vendor-specific API on AMD FPGA and OpenCL pipe API on Intel FPGA to describe a task pipeline with four different stages visualized in Fig 5. It also uses the same 16-element vectors between the tasks. This is again significantly faster than merely vectorizing the kernel since it decouples the data transfer from the computation. This interleaves the data loading, computation and result writeback with each other. FIFO buffers in between tasks also help absorb small variations in external memory accesses.

The proposed method reaches the same performance on Alveo U280 as the most optimized *streaming dataflow kernel*. Since the kernel is so simple, it is likely that the runtime becomes constrained by the external memory bandwidth. With Intel Arria 10 GX, the proposed method is 1.1x slower. The small performance slowdown is likely because of the slower clock frequency. The lower clock frequency is due to the command processor template not yet being optimized for clock frequency. In the future, optimizing this factor should bring the performance to approximate equality, since there is nothing fundamental preventing it.

From the results of Table I we can say that the proposed method does not add significant runtime overhead when compared to the optimized vendor methodology. Additionally, this benchmark demonstrates how much performance is left on the table if non-optimized OpenCL C kernels are used.

In this example, the most optimized versions are not portable between the vendors. The Intel version uses OpenCL pipes, which the AMD implementation does not support, and the AMD version uses a vendor-specific hls_stream-library. In the proposed methodology, these types of optimizations are completely isolated to the creation of the database, where they aren't visible to the end user. Additionally, using OpenCL pipes as was done with Intel's version of the dataflow kernel requires having 4 separate OpenCL kernels, which all need to be instantiated and managed by the host application.

A noticeable FPGA area usage increase of the proposed method can be seen in Table I. This is due to the inclusion of the command processor which implements a more generic AlmaIF-interface for each kernel. This area increase is expected to stay constant when the size of the kernel is scaled up, so it is not estimated to be a significant issue.

The command processing of the proposed method is expected to add a small overhead to all kernel launches since it is more flexible than the vendor methodologies. We measured the effect of this with very small data sizes. From Table II we can see that the proposed method adds only approximately 100 µs latency to each kernel call.

B. Dynamic Reconfiguration

After the user launches the built-in kernel for execution, the driver will reconfigure the FPGA which takes significant time. To evaluate the FPGA reconfiguration time, two sample kernels are created, *add.i32* and *add.i16*. These two kernels are added to a same local bitstream database. The host program initializes an OpenCL device with access to this database. Then, these two kernels are enqueued one after another. The reconfiguration times are measured and reported in Table II. The times are shown to be very similar to the vendor tooling which is unsurprising, since the underlying reconfiguration technology is exactly the same.

V. Future Work

The current implementation reconfigures the FPGA immediately when the user *enqueues* the kernel which can be non-optimal. This can be solved by implementing *deferred reconfiguration*, in which the OpenCL implementation defers the reconfiguration until the host application directly or indirectly waits for the enqueue command to finish.

More options should be explored related to external memory connectivity. It should be possible to have multiple master interfaces connected to different memory banks of e.g. HBM. This would additionally require intelligent allocation and partitioning logic from the OpenCL implementation.

VI. Acknowledgements

The authors are grateful for the funding from Academy of Finland (decisions #331344 and #353199), which made this research possible.

VII. Conclusion

This paper presented a novel, cross-vendor portable method AFOCL, for controlling FPGAs using OpenCL built-in kernels. AFOCL consists of a database of presynthesized bitstreams of built-in kernel implementations and a runtime driver which automatically reconfigures the FPGA device when the user enqueues a kernel.

The method was shown to provide a significant performance improvement (85x and 186x on AMD and Intel, respectively) compared to generic usage of FPGA vendor OpenCL implementations while delivering competitive performance (1.0x and 0.90x, respectively) against hand-optimized vendor-specific kernel implementations. AFOCL demonstrates significant potential to ease the application development for FPGAs since the method decouples the difficult-to-create kernel implementations from the application development. This helps to separate the roles of hardware and software developers by allowing them to work across a common abstraction.

AFOCL source code was made open to enable further development of vendor-independent FPGA programming with open-source OpenCL implementations.

REFERENCES

[1] Khronos® OpenCL Working Group, "The OpenCL™ specification v3.0.14," 2023, accessed 19 May 2023. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

[2] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proceedings of the International Workshop on OpenCL 2013 and 2014*, ser. IWOCL '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2664666.2664670

[3] N. Brown, "Exploring the acceleration of Nekbone on reconfigurable architectures," in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020, pp. 19–28.

[4] ——, "Porting incompressible flow matrix assembly to FPGAs for accelerating HPC engineering simulations," in *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2021, pp. 9–20.

[5] J. Oppermann, M. B. Mickaliger, and O. Sinnen, "Pulsar search acceleration using FPGAs and OpenCL templates," *Experimental Astronomy*, Jan. 2023. [Online]. Available: https://doi.org/10.1007/s10686-022-09888-z

[6] D. Chiou, "System aspects of deploying FPGAs for cloud infrastructure," in *2023 IEEE Custom Integrated Circuits Conference (CICC)*, 2023, pp. 1–2.

[7] "Amazon EC2 F1 instances," Amazon, accessed 19 May 2023. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[8] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, May 2019.

[9] 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, p. 1014–1029, may 2021. [Online]. Available: https://doi.org/10.1109/TPDS.2020.3039409

[10] 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 *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 1–10.

[11] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch, "ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications," in *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, Aug. 2018, pp. 1–9.

[12] 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, sep 2020. [Online]. Available: https://doi.org/10.1145/3405794

[13] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne Switzerland: ACM, Feb. 2022, pp. 933–945. [Online]. Available: https://dl.acm.org/doi/10.1145/3503222.3507740

[14] Y. Xiao, A. Hota, D. Park, and A. DeHon, "HiPR: High-level partial reconfiguration for fast incremental FPGA compilation," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 70–78.

[15] D. Park, Y. Xiao, and A. DeHon, "Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2022, pp. 1–10.

[16] G. Rodriguez-Canal, N. Brown, Y. Torres, and A. Gonzalez-Escribano, "Task-based preemptive scheduling on FPGAs leveraging partial reconfiguration," *Concurrency and Computation: Practice and Experience*, p. e7867, 2023. [Online]. Available: https://doi.org/10.1002/cpe.7867

[17] V. Mirian and P. Chow, "UT-OCL: an OpenCL framework for embedded systems using Xilinx FPGAs," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Riviera Maya, Mexico: IEEE, Dec. 2015, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7393366/

[18] T. Leppänen, A. Lotvonen, and P. Jääskeläinen, "Cross-vendor programming abstraction for diverse heterogeneous platforms," *Frontiers in Computer Science*, vol. 4, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fcomp.2022.945652

[19] 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, no. C, mar 2023. [Online]. Available: https://doi.org/10.1016/j.micpro.2023.104772