

Composable Open-Source Toolchain for Synthesizing Hardware Accelerators from OpenCL Command Buffers

TOPI LEPPÄNEN, Tampere University, Finland
LEEVI LEPPÄNEN, Tampere University, Finland
ZAINAB JAMIL, Tampere University, Finland
JAN SOLANTI, Tampere University, Finland
JOONAS MULTANEN, Tampere University, Finland
PEKKA JÄÄSKELÄINEN, Tampere University, Finland

High-level synthesis tools enable developers to use high-level programming languages such as C, C++, or OpenCL to design FPGA accelerators, reducing the complexity of hardware design. While OpenCL provides a portable programming model for heterogeneous systems, the existing FPGA toolchains require non-portable program modifications, pragmas, and pre-generation of bitstreams. To this end, we present a composable open-source toolchain for automated synthesis of hardware accelerators from OpenCL, bridging the gap between high-level parallel programming and reconfigurable hardware design. Our toolchain supports the standard OpenCL input, and the runtime handles bitstream generation, reconfiguration, and kernel execution. In addition to supporting unmodified OpenCL applications, this work is the first to even partially implement the OpenCL command buffer extension to automatically generate specialized FPGA bitstreams. The toolchain is modular and extensible, supporting multiple HLS tools and a vendor-agnostic FPGA integration flow, demonstrated on FPGAs from two different vendors. Compared to AMD's OpenCL FPGA implementation, the proposed toolchain achieves equal or better performance on 70% of PolybenchGPU benchmarks using standard OpenCL and on 80% when using the command buffer extension.

CCS Concepts: • **Hardware** → **Hardware-software codesign**; *Hardware accelerators*; • **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → Just-in-time compilers.

Additional Key Words and Phrases: FPGA, OpenCL, MLIR, command buffer

ACM Reference Format:

Topi Leppänen, Leevi Leppänen, Zainab Jamil, Jan Solanti, Joonas Multanen, and Pekka Jääskeläinen. 2025. Composable Open-Source Toolchain for Synthesizing Hardware Accelerators from OpenCL Command Buffers. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2025), 27 pages. <https://doi.org/10.1145/3786204>

1 Introduction

Although FPGAs can be used to accelerate suitable applications in an energy-efficient way, they still require a high level of expertise to use. Due to being configurable at the level of independent signals, they can represent many digital circuits with primarily *Look-Up Tables* (LUTs). LUTs and hardened implementations of special-purpose logic are connected to each other with a configurable interconnection network. This low-level circuit description enables very efficient execution of

Authors' Contact Information: Topi Leppänen, topi.leppanen@tuni.fi, Tampere University, Tampere, Finland; Leevi Leppänen, Tampere University, Tampere, Finland; Zainab Jamil, Tampere University, Tampere, Finland; Jan Solanti, Tampere University, Tampere, Finland; Joonas Multanen, Tampere University, Tampere, Finland; Pekka Jääskeläinen, pekka.jaaskelainen@tuni.fi, Tampere University, Tampere, Finland.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 1936-7414/2025/1-ART1

<https://doi.org/10.1145/3786204>

certain applications compared to instruction-based computing devices such as CPUs or (general-purpose) GPUs.

Creating these hardware circuit descriptions is difficult, so *high-level synthesis* (HLS) tools have been proposed, which generate the circuit descriptions automatically from a high-level language such as C, C++ or OpenCL. These tools have made FPGAs easier to use by non-experts. However, the lack of native parallelization and offloading features of C/C++ make it difficult to target FPGAs in a portable way. In a typical HLS development flow, the developer is also tasked with integrating the HLS-generated components into the heterogeneous system. In addition, HLS tools often require the use of vendor-specific code optimizations and pragmas, further reducing portability.

Portability is important, as it would be undesirable to have to use a different programming model for each different device type of a heterogeneous platform. The portability of applications and programming models describes the ability to functionally execute the same application on different heterogeneous platforms. Beyond *functional portability*, the goal is typically to achieve satisfactory performance on multiple different platforms. This can be measured as *performance portability* [35].

In an ideal world, the use of FPGAs would be as easy as any other external accelerator. A high-level application developer would develop their application using a performance portable programming model. The programming model implementation would then take the application, automatically apply optimizations to it, tune it for the specific hardware device, and finally launch it for acceleration. Therefore, the adoption of portable accelerator-centric programming models for FPGA programming is essential for aiding FPGA utilization.

In this work, we evaluate how feasible it would be to construct a portable open-source¹ parallel acceleration framework for FPGAs. For the input language to the framework, OpenCL is chosen as it has been adopted by a wide range of hardware vendors and it offers good portability features. Because of this, it is commonly used as a parallel offloading platform by higher-level frameworks that wish to offload generic workloads to heterogeneous devices. The proposed flow automatically generates the accelerator, integrates it into an FPGA system, and runs the application – all with the standard OpenCL API. This moves the proposed tool up the hierarchy from a mere HLS tool, into an *Accelerator-Centric Synthesis* (ACS) tool, as specified by Del Sozzo et al. [40], since it performs the system integration and runtime communication with the FPGA invisibly to the end user.

OpenCL command buffers enable the description of task pipelines consisting of multiple kernels, which can be compiled together at a user-specified point, making it possible to hide the long FPGA compilation latency. Previous FPGA OpenCL implementations have had to rely on non-conformant external tools and program source modifications to allow the user to describe task pipelines [4, 11, 44]. Additionally, command buffer compilation offers a highly convenient stage to perform *specialization*, since the OpenCL launch parameters and arguments are known for all kernels. Specialization can offer significant performance benefits when offloading applications to parallel accelerators [3]. In this work, we also propose a natural way to handle command buffer *mutability* while avoiding FPGA bitstream recompilation.

The proposed toolchain is built on top of the MLIR compiler framework [25]. Several existing and upcoming HLS tools already work with MLIR, making it highly suitable as a base for modular open-source tooling. This makes it possible to evaluate and experiment with multiple different HLS tools within the proposed automated accelerator synthesis framework. MLIR is part of the upstream LLVM project and is under active development, making it a reasonably future-proof platform to build open-source ACS/HLS flows.

The proposed flow is evaluated by running unmodified OpenCL applications from Polybench-GPU [14], and a performance comparable to AMD's Vitis OpenCL implementation is shown.

¹Source code available at: <http://code.portablecl.org>

PolybenchGPU is then modified to use OpenCL command buffers, which allows the proposed toolchain to further increase performance on both AMD and Altera FPGAs. The toolchain is also shown to work as an underlying acceleration platform for an OpenVX implementation. This demonstrates the benefit of choosing the portable OpenCL programming language as the input to the accelerator synthesis toolchain.

In summary, the main contributions of this work are as follows:

- (1) Open-source toolchain for high-level synthesis of accelerators from portable OpenCL input.
- (2) Composable compiler toolchain with interchangeable front- and back-end components for easy integration with existing tooling.
- (3) The first high-level synthesis of OpenCL command buffers to FPGA accelerators.
- (4) Automated specialization and kernel fusion of command buffers.

The work is organized into the following sections. First, Section 2 introduces the relevant concepts, programming standards, and tools that were utilized in the work. Second, Section 3 highlights related works in this domain. Then, Section 4 presents the toolchain developed in this work. In Section 5, the tool is evaluated and compared with the FPGA vendor tooling. Section 6 discusses opportunities for future work to improve the toolchain, and Section 7 concludes the work.

2 Background

The proposed toolchain uses existing, well-researched technologies to build a maintainable accelerator synthesis framework. In this section, the open parallel programming standard OpenCL, and the popular compiler framework LLVM, and specifically its MLIR infrastructure are introduced.

2.1 OpenCL

OpenCL [23] is an open standard from Khronos designed for portable heterogeneous programming. The fundamental mechanism of OpenCL is based on defining and launching acceleration *kernels* using a standardized C API. The openness of the standard allows multiple competing implementations by independent parties. This increases the portability of the programs written in OpenCL, making it also feasible as a middle-level framework between user programs and actual hardware. As a middle-level framework, the end user does not write OpenCL programs directly, but the OpenCL programs are autogenerated, and the OpenCL API calls are made as needed by some higher-level framework, such as SYCL [16], OpenVX [15] or TVM [7], abstracting the quite verbose OpenCL API to increase programmer productivity. OpenCL also supports *performance portability*, by exposing a mechanism for querying performance-related platform details, enabling the creation of automatically tuned applications.

As part of its specification, OpenCL defines a kernel language *OpenCL C* for describing the kernels. *OpenCL C* is based on the C programming language, but includes a few additional restrictions and specific parallel programming features. Most importantly, it is a *Single-Program Multiple-Data*-form of a program (SPMD), where the compiled program is applied over a dynamic grid of work-items. The programmer describes the behavior of a single *work-item*, which is then automatically scaled by the OpenCL implementation. Before launching the kernel, the user can specify the arguments for the kernel. When it is launched, the user defines the range over which the program is applied by specifying *global* and *local sizes* of the computation grid. An example OpenCL implementation could execute the *work-item* as a thread on a single *Single-Instruction Multiple-Threads* (SIMT)-lane of a GPU or on a *Single-Instruction Multiple-Data* (SIMD)-lane of a CPU vector unit.

OpenCL C standard specifies the behaviors for a set of *built-in functions* that the implementation is free to implement as they want. Examples of these functions are math functions (*sqrt*, *sin*, etc.)

and work-item functions (*get_global_id*, *get_local_id*, *barrier*, etc). The work-item functions are needed for the user to control the SPMD-features of the program.

OpenCL includes a Khronos-ratified extension *cl_khr_command_buffer* for recording and replaying OpenCL commands [23]. The command buffer extension introduces new OpenCL calls for constructing the command buffer object, submitting commands into it, finalizing the command buffer, and finally enqueueing it in a command queue.

Command buffers are a natural way to describe applications that wish to enqueue the same workload consisting of multiple kernels multiple times in succession. An example of this could be a video processing pipeline that applies filtering to each frame of a video stream. In this example application, the command buffer would be constructed once at the beginning and then enqueued again and again for each incoming frame of the video. The use of command buffers can decrease the total latency of the system, since the commands do not need to be reconstructed each time they are reused. The effect of this is especially significant when offloading a batch of computation to a remote server [39].

In case the application requires mutating the command buffer after its finalization, the use of *cl_khr_command_buffer_mutable_dispatch* is required. This layered extension to the base command buffer extension gives the user a separate command handle for each command submitted that they can use to mutate the command invocations with the *clUpdateMutableCommandsKHR*-API call. Possible mutations to the command include: changing the *local size*, *global size*, *global offset*, *arguments*, or command execution info. The implementation can opt into these mutations on a per-device basis, and the user can also enable only some of the mutations if their application does not require everything to be mutable for each of the commands.

PoCL [19] is an open-source OpenCL implementation framework that supports multiple different device types and allows them to share a common OpenCL context. This enables it to be used to program heterogeneous systems in a portable fashion using the standard OpenCL API. The device types PoCL supports include CPUs, GPUs, and customized processors. There are previous works that add FPGA support to PoCL [26, 28], but these works are based only on OpenCL *built-in kernels* and do not support a kernel compiler.

2.2 MLIR Compiler Infrastructure

LLVM is a highly successful open-source project for building compiler toolchains. Due to this, there has been a lot of interest in using LLVM infrastructure to build HLS compilers. Especially the quite recent integration of the MLIR project [25] into the upstream LLVM project has energized the field of open-source HLS tooling.

MLIR is based on representing programs in a multilevel *intermediate representation* (IR) constructed using different *dialects*. MLIR dialects can be likened to *domain-specific* compiler intermediate representations. The user can define their own dialects or use *upstream* MLIR dialects included in the LLVM project's source code repository. The strength of the dialect-based approach comes from the ability to mix the dialects in a single program's IR. For example, the loops of the program can use different specialized dialects for their headers and bodies. More complicated loops can use more unstructured control flow dialects, while simpler loops can benefit from a structured control flow representation to make the application of optimization passes easier. The use of structured control-flow dialects like *Structured Control-Flow (SCF)* and *Affine* enable easier development of optimization passes, since each optimization pass does not need to perform as complicated edge-case analysis as might be necessary when trying to decipher the LLVM IR's unstructured (C-like) control flow.

Due to the large size of the LLVM compiler project, the project maintainers have defined an *incubator*-process for upcoming projects that are based on LLVM infrastructure. The eventual goal

of the *incubator*-projects is to be fully integrated a.k.a *upstreamed* to the main LLVM development repository, which would make their maintenance easier, as they would gain significant visibility, and their development would be fully synchronized to LLVM's core development. Examples of current *incubator*-projects which relate to this work are ClangIR [9], Polygeist [32], and CIRCT [12].

Clang is the C/C++ front-end of the LLVM compiler project. In its traditional flow, it converts C/C++ based languages to LLVM's intermediate representation, typically referred as simply *LLVM IR*. The ClangIR project (CIR) proposes an alternative approach to Clang's direct-to-LLVM IR-flow [9]. It includes an MLIR dialect called CIR that is used as an intermediate representation inside Clang. The program in CIR dialect can then be lowered to MLIR's *LLVM dialect*, after which it is translated in a straightforward way to actual LLVM IR-representation. Alternatively, the CIR dialect can be *raised* to upstream MLIR dialects, although this path is currently declared experimental. ClangIR has been approved to be upstreamed to the LLVM development repository, but as of May 2025, the process of transferring its code to the LLVM repository is still in progress.

Polygeist [32] is a completely alternative C/C++ front-end for MLIR. It can emit MLIR's upstream dialects directly including *SCF*, *affine*, *arith* -dialects, which makes it a highly popular pathway for various projects to get access to the MLIR ecosystem. Polygeist performs the difficult task of inferring the higher-level, structured control flow, from possibly highly unstructured C-style control flow, and automatically converting that to the form better suited for further analysis and optimization passes. It also includes support for converting CUDA and OpenMP programs to MLIR dialects, enabling a way to take advantage of MLIR's natively parallel representations to portably support CUDA and OpenMP on various hardware devices.

CIRCT [12] is an actively developed LLVM incubator project for using MLIR for hardware design. It consists of a collection of MLIR dialects specifically for hardware design, optimization passes operating on these dialects, and conversion passes to progressively lower the program into a hardware description (typically to a *Hardware Description Language* (HDL), such as SystemVerilog). By chaining these dialects together with appropriate lowering passes, it is possible to lower a program consisting of generic MLIR dialects into a hardware description of a circuit. The popularity of the CIRCT project motivates the use of MLIR as a part of modern, maintainable HLS toolchains.

3 Related Work

While high-level synthesis tools are able to generate specialized and efficient RTL implementations from a high-level language program description such as C or C++, they do not always consider the system-level integration and runtime control of the generated IP. This task is left to the end user of the tool, which, while increasing the flexibility of the tool and enabling more customized integration of the end product, also adds significant complexity to their use. These types of HLS tools are commonly classified as the *third generation* HLS tools [29]. They can still be quite difficult to use by typical software developers, who might prefer to use FPGAs as generic accelerator devices.

More advanced tools, which also automate system-level integration and provide the end user with an abstraction of an acceleration platform, are classified as Accelerator-Centric Synthesis (ACS) tools by Del Sozzo et al. [40]. FPGA vendors AMD (formerly as Xilinx) and Altera (previously part of Intel) have provided ACS-level tools that allow the end user to use C, C++, OpenCL, and SYCL to program FPGAs. In this work, the choice is made to focus on OpenCL as the input language for ACS tools, as it provides a native acceleration API, as opposed to *ad hoc* vendor-specific APIs required for C or C++.

Numerous previous works have implemented OpenCL for FPGAs, and several works have used MLIR for high-level synthesis. However, to our knowledge, our work is the first one that combines the two. Therefore, in this section, the related OpenCL and MLIR works are introduced separately.

3.1 OpenCL HLS Tools

The most used OpenCL ACS tools are closed source tools provided by the FPGA vendors [4, 17]. This inhibits research into system-level synthesis and to the use of FPGAs as generic acceleration devices. As an example of the limitations of the FPGA vendors' OpenCL implementations, they do not support the OpenCL standard method of constructing the programs with `clCreateProgramWithSource`, but instead require the use of custom external tools to prepare the program binary. Additionally, they may require the user to manage the accelerator configuration state and buffers in a more extensive way than would be required by valid OpenCL programs, such as having to create the program object before being able to create buffer objects [28]. In our work, we address these issues by implementing a more conformant OpenCL runtime and integrating to an open-source standard-conforming OpenCL implementation: PoCL [19].

There are also many existing academic OpenCL HLS tools [10, 20, 31, 34, 36, 38]. However, many of these works have not released their source code, which limits the research into OpenCL-based HLS frameworks. Therefore, we will introduce the existing open-source works in more detail, as looking at their implementation makes them more interesting as a basis for an open-source accelerator synthesis framework.

SOFF [20] is an open source OpenCL high-level synthesis framework. SOFF performs spatial scaling by creating multiple parallel datapaths on the FPGA, and then having a dispatch component that submits work-groups to the datapaths. Each datapath computes a single work-group sequentially, which simplifies their handling of OpenCL barriers and local memory. Compared to our work, they understandably do not support OpenCL command buffers, as the command buffer extension was not released when their work came out. Instead, they handle multi-kernel OpenCL programs by synthesizing separate accelerators for each kernel. Only one of these kernels can be in execution at a time, which limits the available task-level parallelism. Similarly to the FPGA vendors' OpenCL implementation, SOFF requires the use of custom offline tools to pre-generate the FPGA bitstream, which is then loaded in with `clCreateProgramWithBinary`. Their compiler is implemented within LLVM's Clang front-end, in which it preprocesses the OpenCL C source and directly emits Verilog HDL. Our implementation is based on MLIR, and allows the use of different HLS backends, increasing the modularity of our method compared to SOFF. Our MLIR-based implementation uses the MLIR's *parallel*-representation, which allows the backend to later perform spatial scaling and pipelining, without having to rediscover the parallelism. Still, some parts of SOFF's implementation could be useful in ours, as they do achieve greater coverage of benchmarks by supporting the entire polybenchGPU [14] and SpecAccel [22] benchmark sets, while achieving higher performance than Altera's OpenCL implementation.

PCIeHLS [43], ZUCL [36], and FOS [42] are related projects that claim to support OpenCL as a programming model for controlling FPGAs. The latest work, FOS, proposes an abstraction layer similar to operating systems for FPGAs. Their system can multiplex between different acceleration tasks both spatially and temporally. They use a custom dynamic *Partial Reconfiguration* (PR) flow based on their own tooling, which allows them to use the FPGAs in a much more flexible way than would be available if only using the vendor-provided PR flow. The multiple abstraction layers of FOS decouple accelerator development from FPGA platform development. In practice, this means that their HLS tools can target a fixed interface, which is similar in aim to our use of the AlmaIF API [26] to abstract accelerator control. The kernels can be written in C, C++, OpenCL C, or RTL. Looking more carefully at the open-source implementations of ZUCL and FOS, they do not include a proper implementation of the OpenCL API. ZUCL includes a custom, OpenCL-inspired way to launch kernels [41]. The non-conformance makes it difficult to integrate portable OpenCL applications or frameworks, since the runtime API is not implemented. We believe that the implementation of the

standard OpenCL runtime API to control FPGA accelerators would be beneficial for their work and could have potential to expose their impressive low-level FPGA management to more higher-level programming frameworks.

UT-OCL [31] is an open-source OpenCL implementation for embedded Xilinx FPGAs. They propose a framework for investigating different platform-level designs in OpenCL FPGA implementations. The scope of their work is not on high-level synthesis of kernels, but instead they only emulate kernels with soft processors. Integrating kernel synthesis into their framework would be possible in the future. Compared to theirs, in our work we explicitly focus on OpenCL kernel synthesis, while also integrating it into an OpenCL runtime. In our work, we show that kernel synthesis and OpenCL framework implementation are not always separable, as there are connections in both directions between these. As an example, our implementation of OpenCL command buffers relies on the implementation to collect the user data from OpenCL API calls to construct the command buffer which is then synthesized as a whole.

Wang et al. [44] present an OpenCL-based platform for executing multi-kernel task pipelines on FPGAs. They propose a custom OpenCL extension with a new syntax to describe multi-kernel pipelines, instead of using the ratified OpenCL *command buffer* extension as we do, even though it might have been well suited for their use case. Their implementation then schedules and runs the kernels automatically as a kernel pipeline, similarly to our implementation.

3.2 HLS Tools Using the MLIR Framework

There are multiple active open-source projects for generating hardware circuit descriptions using MLIR. All of these works could at least partially be used as the kernel compiler middle- and back-end implementation of the proposed toolchain.

CIRCT [12] is an LLVM incubator project that uses MLIR infrastructure for hardware generation. CIRCT includes multiple dialects at different levels of hardware abstraction, which allows progressive lowering of high-level dialects towards hardware circuit description. CIRCT supports both scheduling HLS operations statically with the *Calyx*-dialect or dynamically using the *Handshake*-dialect. Dynamatic [21] is an HLS compiler based on the same Handshake-dialect included in CIRCT. The dynamic scheduling approach enables it to optimize hardware modules separately and connect them with *streams*, to minimize global control overhead.

HeteroCL [24], later superseded by Allo [6] is a custom MLIR-based programming model for spatial accelerators. Using Allo, the user can generate accelerators by adding customization primitives (similar to pragmas) to the algorithm description in Python.

SODA-OPT [1], ScaleHLS [45], Hida [46] and StreamHLS [5] are MLIR-based projects that apply HLS-specific optimizations as MLIR passes and define HLS-specific MLIR dialects. They perform dataflow optimizations automatically, and then export the program to a back-end HLS tool such as Bambu [13] or Vitis HLS for RTL generation. This is similar in scope to our work, as we also utilize existing tooling for back-end HLS synthesis and bitstream generation. Another limitation in some of these works is the reliance on static memory latency, so they do not attempt to optimize the off-chip memory transfers, which can be a critical factor for performance. Still, at least based on simulated results, they outperform Vitis HLS by multiple orders of magnitude.

None of the previous MLIR HLS works support OpenCL as an input language. The proposed toolchain could therefore be used to bring OpenCL support to these otherwise highly interesting projects, further elevating their reach into the domain of portable parallel programming.

4 Modular Accelerator-Centric Synthesis Framework

In this work, we propose and implement an OpenCL-based ACS-tool. It can utilize open-source high-level synthesis tools listed in Fig. 1 as parts of its internal implementation. The toolchain

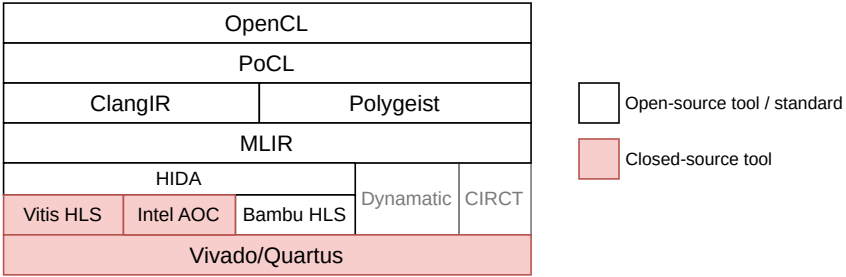


Fig. 1. The components used in the proposed accelerator synthesis toolchain. The goal is to gradually replace closed-source components with open-source ones.

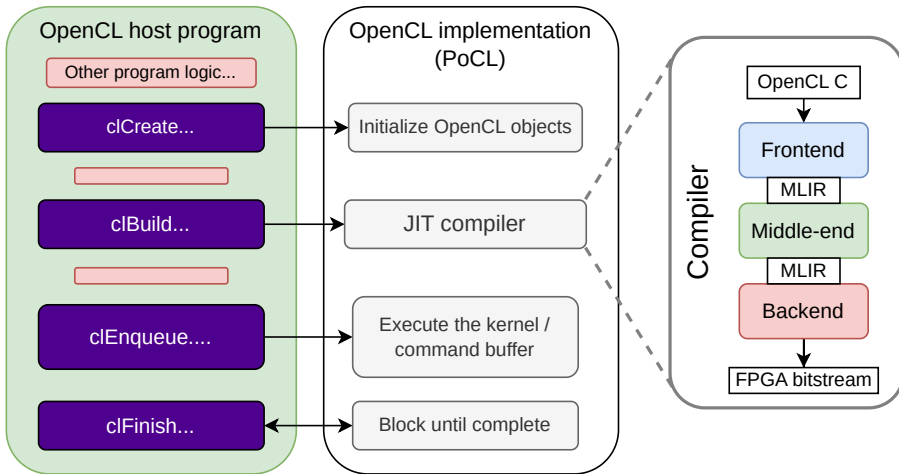


Fig. 2. The proposed OpenCL implementation prototype includes a JIT compiler that is used in proper coordination with the implementation of the other OpenCL API calls. On the right side of the figure is the high-level structure of the proposed MLIR-based compiler.

consists of two major parts, an OpenCL runtime based on PoCL [19], and an OpenCL C JIT compiler. As shown in Fig. 2, the OpenCL runtime communicates with the user program and calls the JIT compiler as needed to generate FPGA bitstreams.

4.1 OpenCL Runtime

To implement the OpenCL standard, it is not enough to only create an *OpenCL C* compiler. The OpenCL implementation must also implement other coordination features for managing the heterogeneous computing system, such as buffer allocation, command queues, event handling, etc. In this work, previous works by Leppänen et al. [26, 28] are utilized to leave the management of most of these other aspects to a generic OpenCL implementation framework PoCL [19].

OpenCL follows a *dual-source code* model in which the kernel source code is compiled at the application runtime. The typical place for compilation is the *clBuildProgram* call, which allows the user to pass additional compilation parameters. Alternatively, kernel compilation may be deferred

to *enqueueing* time, which allows the implementation to compile a version of the kernel that is specialized by the launch parameters. This is the current default behavior of PoCL [19].

However, this is not directly usable for the proposed FPGA-based flow since it would require a faster way to switch the program on the FPGA. On a CPU or a GPU, changing the program is fast enough to not present any issues, as they are instruction-based compute devices, which can change their program by reloading a different program binary to their general-purpose memory. Since the FPGA fabric is reconfigured at a much finer granularity, making the whole-FPGA reprogramming take up to multiple seconds, it is no longer feasible to switch out the program binary for each kernel. This restriction forces the proposed implementation to generate the FPGA bitstream at the *clBuildProgram*-call, which limits the level of specialization available, as the launch parameters (e.g. OpenCL *local size*) cannot be specialized into the FPGA bitstream.

The proposed implementation supports *clCreateProgramWithSource*, which means that the program can be built using the OpenCL standard API without requiring the user to first call custom tools to generate prebuilt FPGA binaries, as the AMD and Altera tools do. The proposed tool still supports offline compilation, via the *CL_PROGRAM_BINARIES* parameter to the *clGetProgramInfo* which can be queried after building the program to extract the program binary. This enables explicit management of the build artifacts (FPGA bitstreams). Even without this, the implementation caches the FPGA bitstreams to an internal directory, to avoid needless regeneration of the FPGA bitstream every time the application is rerun. Additionally, the tool does not rely on having an existing OpenCL program object constructed before OpenCL buffer allocations, or disallow the change of OpenCL program while having active buffer allocations, unlike AMD and Altera OpenCL tools, making the proposed tool more OpenCL-compliant in that aspect. This makes it easier to integrate into existing OpenCL applications and frameworks, as there are less OpenCL host code modifications needed.

The Khronos command buffer extension gives the OpenCL implementation an additional user-defined spot for kernel compilation at *clFinalizeCommandBufferKHR*-call. Therefore, the use of command buffers could be a novel method to solve the problem of non-specialized kernel instances on FPGAs. The focus in this work will be in this *finalization*-stage of the OpenCL command buffer lifetime, since it offers a highly interesting point to perform multi-kernel and specialization optimizations for FPGA devices.

At this stage, all buffers have been created, kernel arguments are set, and kernels have been submitted to the command buffer with their launch parameters. Therefore, the complete task pipeline is known to the compiler, including the kernel arguments, local and global sizes, and the inter-kernel connectivity. The specialization available at the command buffer finalization time can help eliminate many of the FPGA-specific modifications often necessary in FPGA OpenCL tools related to e.g. conversion to single-work item kernels and description of multi-kernel spatial task pipelines.

In the proposed implementation, the command buffer is built into an FPGA bitstream at the *clFinalizeCommandBufferKHR*-call and cached into an implementation-managed directory, so that future calls to the same application will not needlessly recompile the command buffer. The command buffer extension does not define an explicit way of caching the built command buffer, like it does for built programs, so the current implementation relies on the implementation's internal caching to prevent recompilation at the application deployment.

While it is valid to submit a mixture of command buffers and regular OpenCL kernels, this work only evaluates the case when the user does not do that, but instead always constructs a single OpenCL command buffer, and then enqueues that multiple times. In the proposed implementation, since the FPGA is reconfigured at the *clFinalizeCommandBufferKHR*-call, any subsequent enqueues of the finalized command buffer are fast and do not require reconfiguration. If the user would

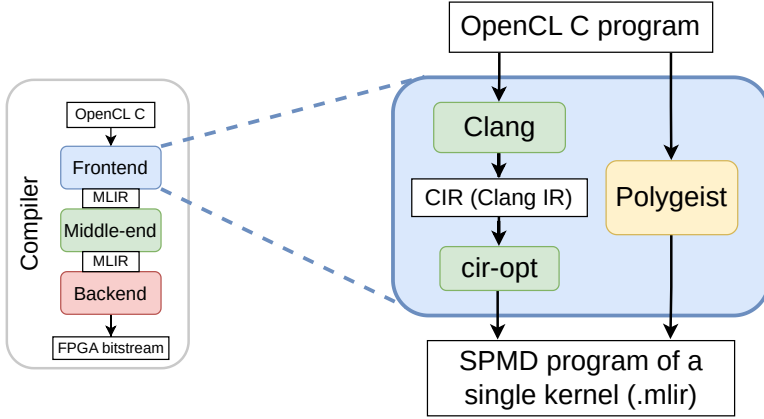


Fig. 3. The frontend of the proposed OpenCL C JIT compiler. By using the MLIR as an intermediate representation, the proposed toolchain is able to take advantage of two separate MLIR frontend projects. The *cir-opt* component that converts the CIR dialect to MLIR dialects is highly experimental.

submit a different command buffer, or a different, independent kernel, the proposed implementation would be forced to reconfigure the FPGA, which can be a costly operation, taking up to multiple seconds.

In the following three sections, the structure of the proposed OpenCL C compiler is described. To simplify the explanation, the description is split along the typical way of structuring a compiler by describing the front-end, middle-end, and back-end separately.

4.2 Kernel Compiler Front-end

The front-end of the proposed compiler shown in Fig. 3 takes the OpenCL C kernel program and converts it into MLIR representation, which uses the upstream dialects of the LLVM-project. There are at least two potential components that could be used to perform this task: ClangIR [9] and Polygeist [32]. Since the proposed compiler is built to be as modular as possible, both of these are integrated into the toolchain. To evaluate the usability of these alternative front-ends, benchmark sets are compiled with them, and the generated MLIR is evaluated manually to see whether the front-end is able to generate valid programs.

Based on our evaluation, Polygeist is more suitable for this task, as it can compile the entire PolybenchGPU [14] and many applications from SpecAccel [22]. Additionally, Polygeist is able to *raise* the abstraction level of IR up to the MLIR *affine* dialect, which enables more optimizations than what is available when using only the lower-level dialects as ClangIR does. For these reasons, for the rest of this paper, Polygeist is used as the OpenCL C compiler front-end to raise the IR abstraction as high as possible for the sake of further optimizations.

Based on our experimentation, ClangIR seems to be a more in-progress work, as many internal modifications to its source code are needed in order to generate programs that use upstream MLIR dialects. Even then, it is only able to compile one of the SpecAccel-benchmarks. Especially ClangIR's internal *cir-opt* tool that converts from the *CIR* dialect to regular upstream MLIR dialects is very experimental work and would require much more development effort to be able to compile typical programs to upstream MLIR dialects.

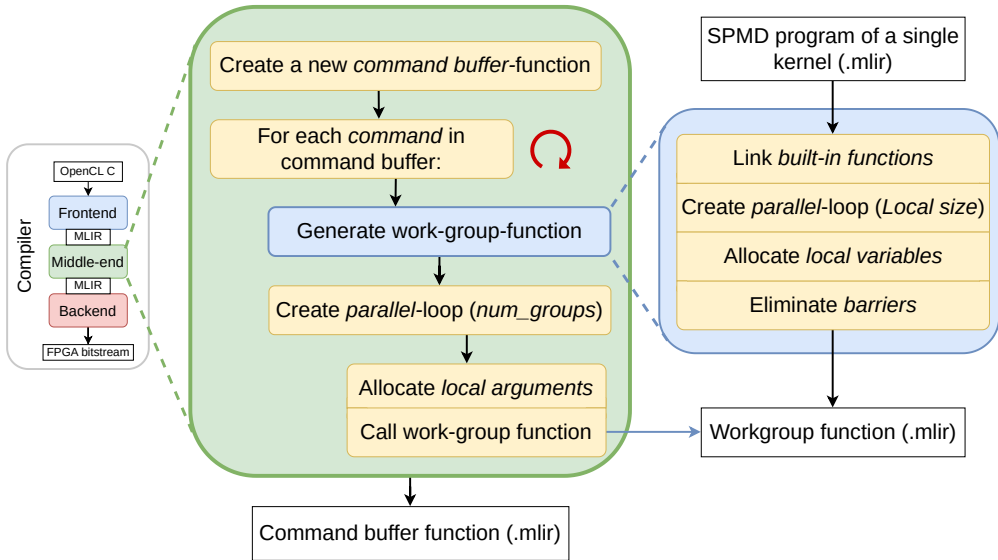


Fig. 4. The middle-end of the proposed OpenCL C JIT compiler. Most of the transformations are done in generic upstream MLIR dialects. The right side of the figure converts the SPMD kernels programs into work-group functions. When compiling a command buffer, the middle part of the figure generates the *command buffer* function that sequentially calls all the commands invoked in the OpenCL command buffer. The use of upstream MLIR dialects enables the proposed compiler to use pre-existing MLIR transformation passes, e.g. *barrier elimination*-pass from Polygeist [33].

In the future, as the ClangIR project develops, it might make more sense to shift the front-end of the proposed toolchain to use ClangIR. ClangIR seems to have a more promising future, as it will eventually be integrated to the upstream LLVM-project and will therefore continue to be under active development. This is different from Polygeist, which is still an LLVM incubator project and has a much lower level of development activity than ClangIR as of May 2025.

4.3 Kernel Compiler Middle-end

Shown in Fig. 4 is the middle-end of the kernel compiler. It works mostly in upstream MLIR dialects, making it generic and easily extensible to other MLIR-based projects. When the command buffers are not used, only the parts shown on the right side of the figure are used to convert the SPMD program to a workgroup-function representation in a *de-SPMD*-pass. In this part of the flow, the kernel function library is linked in, barriers are eliminated, and local memory is allocated. When finalizing a command buffer, the kernel command enqueues are generated for as many work groups as specified by the user in the OpenCL *global size*-parameter. Not shown in the figure are multiple generic MLIR optimization passes that are inserted into various points. These include for example *canonicalization*, *common subexpression elimination*, *memory-to-register* transformations, generic *affine loop optimizations*, etc.

4.3.1 Single Kernel Transformations. In the proposed implementation, the OpenCL *built-in functions* are linked to the user kernel program as MLIR representations. The kernel function implementations can be described either in C, in which case they are converted to MLIR by Polygeist when the toolchain is built, or directly as MLIR representations. Many of the OpenCL built-in functions

have a direct correspondent in MLIR's math-dialect, which makes them easy to implement in the proposed MLIR based flow, as can be seen in Listing 1.

```

module {
  func.func private @_Z8_cl_sqrtf(%a: f32) -> f32 {
    %res = math.sqrt %a : f32
    return %res : f32
  }
  func.func private @_Z8_cl_sqrtf(%a: f64) -> f64 {
    %res = math.sqrt %a : f64
    return %res : f64
  }
}

```

Listing 1. OpenCL *sqrt*-built-in function implementations for 32-bit and 64-bit precision using MLIR's math dialect. The names are mangled since OpenCL C overloads the function names so that the same function name can be used for functions that differ only in their parameter types.

The built-in work-item functions require more careful implementation, as they depend on the de-SPMD-transformation described below. Therefore, at the linking stage, the built-in work-item functions are only replaced by MLIR's GPU dialect's *thread_id*, *block_id*, *block_dim* and *barrier*-operations (CUDA terminology), and then the upcoming de-SPMD-pass will replace these with appropriate implementations. Implementations for the built-in work-item functions are shown in Listings 2 and 3.

```

func.func private @_Z12get_local_idj(%arg0: i32) -> i64 {
  %c0_i64 = arith.constant 0 : i64
  %0 = arith.index_cast %arg0 : i32 to index
  %1 = scf.index_switch %0 -> i64
  case 0 {
    %thread_id_x = gpu.thread_id x
    %2 = arith.index_cast %thread_id_x : index to i64
    scf.yield %2 : i64 }
  case 1 {
    %thread_id_y = gpu.thread_id y
    %2 = arith.index_cast %thread_id_y : index to i64
    scf.yield %2 : i64 }
  case 2 {
    %thread_id_z = gpu.thread_id z
    %2 = arith.index_cast %thread_id_z : index to i64
    scf.yield %2 : i64 }
  default {
    scf.yield %c0_i64 : i64 }
  return %1 : i64
}

```

Listing 2. OpenCL *get_local_id*-built-in function implementation written directly in MLIR using SCF and arith dialects. In typical usage with a constant input parameter the entire function is inlined and optimized to be just a single *gpu.thread_id*-op.

```

size_t _Z13get_global_idj (unsigned int dim) {
  return _Z12get_group_idj(dim) * _Z14get_local_sizej(dim)
    + _Z12get_local_idj(dim) + _Z17get_global_offsetj(dim);
}

```

```
}

```

Listing 3. OpenCL *get_global_id*-built-in function implementation written in C. The implementation calls into other built-in functions (using their mangled names) written in MLIR. These calls are inlined and reduced to only their fundamental *gpu*-dialect ops.

After the built-in functions are linked, the SPMD program is converted to a version that executes an entire OpenCL workgroup. The SPMD version of the program is wrapped inside of an *affine.parallel*-operation, where the bounds of the *affine.parallel*-operation correspond to the OpenCL *local size* given at the kernel invocation. At this point of the compiler pipeline, the local size bounds can also be left as dynamic values, in case the local size is not yet known. Simultaneously, the GPU dialect's *block_dim* operations are lowered to the bounds of the *affine.parallel*-operation. The *gpu.thread_id*-operations (a.k.a OpenCL *local id*) are lowered to the *affine.parallel*-operation's iterator variables. The *gpu.block_id* operation (a.k.a OpenCL *group id*) is constant within a workgroup, but different for each invocation of a workgroup function, so it is extracted as a hidden function parameter of the work-group function.

At this stage, the IR can be temporarily invalid, as the potentially existing *gpu.barrier*-operations in the work-group function prevent the free parallelization of the *affine.parallel*-op. To fix this, the barriers must be eliminated. Polygeist's barrier removal pass [33] is used to achieve this. It works with *polygeist.barrier*-operations, which are otherwise identical to *gpu.barrier*, but take the *parallel*-operation's iterators as inputs (in the proposed implementation these correspond directly to OpenCL *local ids*). Therefore, to use the Polygeist's barrier removal pass, it is enough to replace *gpu.barrier*-operations with the *polygeist.barrier*-operations, passing the *affine.parallel*-operation iterators to it, and calling the removal pass. The current implementation of the barrier-elimination pass requires the *local size* to be known, since the pass allocates function-internal memory. Since the current implementation does not support dynamic memory allocation inside the kernel, it therefore does not support OpenCL barriers in non-command buffer programs.

OpenCL local memory declared in the scope of the kernel function is a shared memory region between the work-items of a same work-group. This memory is not visible outside of the kernel, so it is possible to implement it completely inside the newly created work-group function. Implementing this feature is as simple as moving the *memref.alloca*-operations allocated in the OpenCL local address space outside of the *affine.parallel*-operation, so that the local memory is not re-allocated separately for each work group function loop iteration. After this transformation, all work items of the *affine.parallel* operation will now use a common *memref.alloca* handle to access the variable in local memory.

4.3.2 Command Buffer Compilation. At the OpenCL *clFinalizeCommandBufferKHR*-call, a specialized command buffer implementation is generated. First, the single-kernel transformations described above are called for each command in the command buffer to generate specialized single-kernel implementations for each kernel invocation. Then, the commands are fused together to form a new hidden *command buffer kernel*. This "super"-kernel includes as its arguments all the buffer arguments of the kernel commands, and invokes the other kernels as function calls.

At the finalization stage, all the kernels and their arguments are known in (non-mutable) command buffers. This enables some optimizations that would not be possible to perform at typical OpenCL program build-time. The current implementation internally de-duplicates all the global buffer arguments with the same value. This is done to simplify later alias analysis, since after de-duplication it can be assumed that no global buffer arguments ever alias with each other. This is currently a required step in the toolchain since one of the used back-end compilers assumes non-aliasing pointers. The current implementation does not support OpenCL sub-buffers.

Anything that is known to be constant by the command buffer finalization time is replaced by an actual constant in the compiler IR to maximize optimization opportunities. All the arguments passed in by value are replaced by their value. The whole NDRange, both global and local sizes, are known for all the kernels, which means that many parallel-operation boundaries can be replaced by constant values, simplifying many loop optimizations. At this time, also the size of local buffer arguments is known, which allows statically allocating the local memory in the *command buffer*-function before the kernel that uses the local buffer arguments is invoked.

To simplify the current implementation, it is limited to only in-order command queues and does not allow including other than *clCommandNDRangeKernelKHR*-calls in the command buffer. Fixing these limitations would not drastically change the underlying implementation method. For example, if the user would like to submit the *clEnqueueCopyBuffer*-command in the middle of two kernel enqueue commands, a new custom copying routine could be automatically inserted in between the kernels.

4.3.3 Mutable Command Buffers. Since many applications require the use of mutable commands in command buffers, the proposed implementation partially supports these. There are two critical properties to consider when supporting mutable command buffers on FPGAs. First, updating the mutable commands of a command buffer shouldn't trigger a recompilation, as it is such an expensive operation. Second, the command buffer should be only as mutable as required and no more. This means that combining nonmutable and mutable commands in the same command buffer and only retaining the mutable property that the user specified with the *CL_MUTABLE_DISPATCH_UPDATABLE_FIELDS_KHR*-parameter should be supported.

The proposed implementation checks the property and leaves either the global size or all the arguments of a given command mutable, while specializing everything that the user did not require to be non-mutable. An example of this is shown in Fig. 5. In this example, kernel1 is fully specialized, kernel2 is specialized by its arguments while leaving the global size mutable, and kernel3 specifies both the arguments and the global size to be mutable. The *affine.parallel* operations of the different kernels follow each other immediately inside the single *command_buffer_function*. One potential way to specialize this even further would be to allow the user to specify single arguments as mutable, but this is not currently possible with the Khronos mutable command buffer extension.

The mutable global size is implemented by creating three hidden arguments (x,y,z) to the command buffer function for each command that has mutable global size. This enables passing the number of work groups to the command buffer kernel before it is launched. Any updates to the mutable global size can then be made by updating the values of these hidden arguments.

The current implementation does not fully implement the mutable command buffers as specified in the Khronos command buffer extension. Only *CL_MUTABLE_DISPATCH_GLOBAL_SIZE_KHR* is supported and *CL_MUTABLE_DISPATCH_ARGUMENTS_KHR* is partially supported. The user can query the OpenCL device for the information on which parameters it exposes for mutability and only use those in the application.

There are two limitations to the mutable arguments, which are not compliant with the specification. First, the size of the local buffer arguments is not mutable. According to the specification, the user should be able to update the size of the local memory arguments, just like they would update any other OpenCL argument. The *values* of the local memory arguments are not user-configurable anyway, as they are internally allocated by the implementation and not accessible to the host or other kernels. In the current implementation, the local memory arguments are statically allocated at the command buffer finalization-time "just outside" the kernel, but still inside the internal *command_buffer*-function and the FPGA bitstream. The inability to mutate local buffer arguments is therefore in place to avoid recompilation. An alternative way to achieve compliance would be to allocate

OpenCL host code:

```

clCreateCommandBuffer(...)
clCommandNDRangeKernel(kernel1, global_size = {1024, 1024, 1024}, local_size = {16, 16, 16},
    mutable_fields = 0 )
clCommandNDRangeKernel(kernel2, global_size = {8, 1, 1}, local_size = {4, 1, 1},
    mutable_fields = CL_MUTABLE_DISPATCH_GLOBAL_SIZE_KHR )
clCommandNDRangeKernel(kernel3, global_size = {32, 32, 32}, local_size = {16, 16, 16},
    mutable_fields = CL_MUTABLE_DISPATCH_ARGUMENTS_KHR |
    CL_MUTABLE_DISPATCH_GLOBAL_SIZE_KHR )
clFinalizeCommandBuffer(...)

```

Generated MLIR command buffer function:

```

func.func command_buffer_function(..kernel2_num_groups_x.., ..kernel3_args.., ..kernel3_num_groups_x/y/z..)
    affine.parallel (%0, %1, %2) = (0, 0, 0) to (64, 64, 64)
    affine.parallel (%3, %4, %5) = (0, 0, 0) to (16, 16, 16)
    kernel1 computation
    affine.parallel (%0) = (0) to (%kernel2_num_groups_x)
    affine.parallel (%1) = (0) to (4)
    kernel2 computation
    affine.parallel (%0, %1, %2) = (0, 0, 0) to (%kernel3_num_groups_x, .._y, .._z)
    affine.parallel (%3, %4, %5) = (0, 0, 0) to (16, 16, 16)
    kernel3 computation

```

Fig. 5. An example command buffer showcasing the partial mutability support in the proposed implementation. The shown OpenCL host code syntax is simplified for brevity. The actual implementation uses the syntax as defined in the OpenCL standard [23].

the local memory dynamically, using either a block of pre-created on-chip memory or memory external to the FPGA chip.

The second limitation relates to the mutation of global buffer arguments. The current implementation requires that any mutations to global memory pointer arguments must preserve the same aliasing as was the case in the original arguments. This means that the user must always reset all global buffer arguments that pointed to the same underlying buffer in the original command buffer to still point to a common buffer in the modified command buffer. This is possible to do in a specification-defined way using multiple *cl_mutable_dispatch_config_khr*-configs in the same *clUpdateMutableCommandsKHR*-call. If the user does not update all the aliasing arguments in the same *clUpdateMutableCommandsKHR*-call, the implementation will return an error. This makes this part of the implementation non-compliant, but at least the user has a proper way to notice the issue. The limitation is caused by the proposed implementation's de-duplication of aliasing buffer arguments described in Section 4.3.2. In practice, the restriction on the use of mutable command buffers might not be as limiting as it seems, since in at least all of the evaluated benchmarks, all the aliasing buffer arguments are always mutated to the same value at the same time. If strict compliance would be required, a compliant way to implement this would be to re-generate the bitstream at the mutation step, when the implementation notices a change in the buffer aliasing.

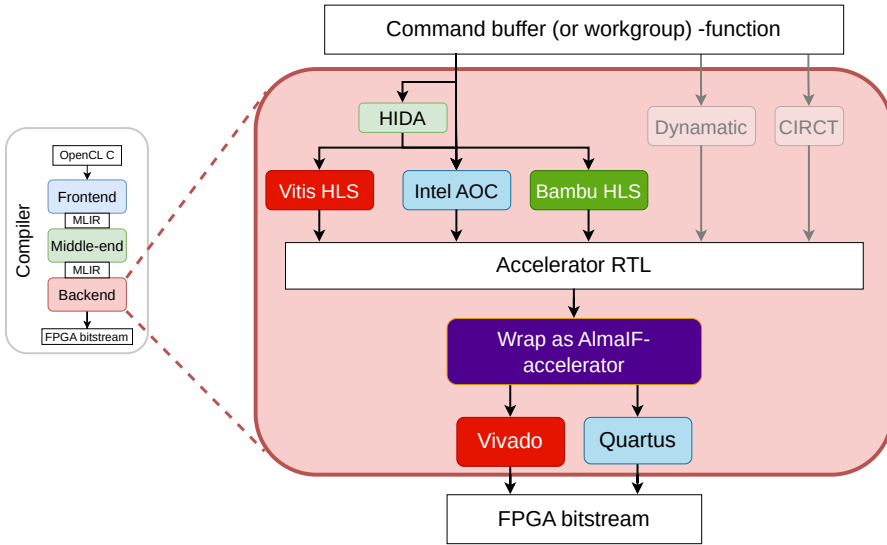


Fig. 6. The backend of the proposed OpenCL C JIT compiler. The incoming program is in generic MLIR format, which enables the proposed toolchain to use alternative HLS tools for backend HLS synthesis. The backend converts the program into a circuit description, generates an AlmalF accelerator out of it, and calls the necessary FPGA vendor tooling to generate the bitstream. The bitstream can either contain multiple kernels as independent accelerators, or a single command buffer kernel. CIRCT and Dynamic are greyed out due to them being not yet fully functional flows.

4.4 Kernel Compiler Backend

The steps in the kernel compiler so far have been agnostic to the actual acceleration device. In the stage of the toolchain shown in Fig. 6, the generic MLIR functions are converted into a hardware circuit that executes the kernel. As a debugging feature, at this stage the MLIR function can also be lowered via LLVM to a host kernel, and directly executed. This is equivalent to the C-simulation feature available in commercial ACS tools.

Making the back-end compilation modular is the key to building a portable and modular compilation pipeline. The backend compilation consists of two major steps, first generating the actual computation circuit with a substitutable HLS tool, and then wrapping the computation circuit in a well-specified memory-mapped IP, which is then easy to integrate into the actual FPGA device so that it can communicate with the provided OpenCL driver. The use of cross-vendor portable hardware integration methodology [26] provides a way for the proposed toolchain to utilize FPGAs from different vendors.

4.4.1 Generating Kernel RTL. The generation of the computing circuit is left for pre-existing HLS tools. These tools work with the generic MLIR input and export an RTL circuit with potentially custom interfacing to the external memory. Initially, Hida [46] is chosen as an HLS tool to perform this task. It applies HLS optimizations as MLIR passes, and then emits C++ code suitable to be passed into Vitis HLS. Hida is chosen because it produces highly competitive results [46] and because it is easy to integrate with the proposed MLIR-based flow. It still relies on Vitis HLS to generate the actual RTL, which unfortunately introduces a closed-source component to the toolchain. However, the use of reliable Vitis HLS for RTL generation means that even if Hida is

unable to apply optimizations to some difficult-to-compile kernel, it will still emit valid C++ for Vitis HLS, which is then able to produce a valid RTL implementation, even if it is unoptimized. As part of this work, minor modifications to Hida are made to make it emit single work-item OpenCL C programs suitable for an OpenCL-based HLS back-end compiler.

Bambu [13] is an open-source HLS tool that can generate RTL circuit descriptions starting from C/C++, or from compiler intermediate representations (GCC or Clang). It is a standalone tool and can be used in a similar manner as e.g. Vitis HLS. Since it is an open-source tool, using it would benefit the proposed toolchain, since it enables deeper research into HLS optimizations. To use Bambu in the proposed toolchain, the processed program is emitted as a C++-kernel using the emission code of Hida, and then Bambu is invoked to convert the C++ description into an RTL circuit.

On Altera FPGAs, it seems natural to integrate Altera's HLS compiler to the proposed toolchain. The vendor tool might be able to take advantage of device-specific optimizations to generate efficient circuits. However, Altera has deprecated their C/C++ and OpenCL compilers in favor of the oneAPI programming model [18]. The future of Altera's FPGA tooling and programming language support appears unclear as of May 2025, since Intel recently sold majority ownership of Altera to an external party [30]. Therefore, it is not guaranteed whether Altera will in the future support only the previously Intel-led oneAPI programming model. In any case, this uncertainty serves as a further motivation for the proposed open-source toolchain, as it would liberate any specific vendor's FPGA devices to be programmed with other than the vendor-provided programming framework. As a proof-of-concept, the *Intel FPGA SDK for OpenCL* (AOC) compiler is still for now integrated as one of the back-end compilers for the proposed toolchain to generate accelerator RTL.

CIRCT [12] would be another attractive option for the HLS back-end, since it is an open-source tool and actively developed by multiple parties. It includes a tool called *hlstool* which is supposed to be able to take in high-level MLIR dialects, and generate RTL circuits with two alternative paths, one using Calyx IR, and another using a Handshake dialect. However, to integrate it with the proposed tool, much more work would be needed, since neither of the flows can currently compile even very basic programs that access external memory.

Dynamatic [21] is another MLIR-based tool, which uses the same Handshake dialect available in CIRCT to generate elastic circuits without having to use a static scheduler to organize the computation. Dynamatic can take advantage of the *affine.parallel*-operation's semantics to generate more parallel circuits than if only using *affine.for*-loops. The current issue for integrating Dynamatic to the proposed flow relates to its inability to generate elastic memory interfaces, which are required for reading and writing data to external memory. Implementing this feature to Dynamatic might make it possible to integrate to the proposed toolchain, since Dynamatic is able to generate RTL circuits (although with static latency memory interfaces) for many of the benchmarks.

4.4.2 Wrapping the RTL as an AlmaIF Accelerator. Wrapping of the RTL circuit to an FPGA bitstream can be a highly vendor-specific task. To increase the cross-vendor portability of the proposed toolchain, the AlmaIF integration methodology [26] is used to carry this out. The kernel compiler back-end needs to take into account the method of runtime communication with the OpenCL driver. The AlmaIF-driver abstracts the communication between the accelerator device and the OpenCL driver to work in a similar, memory-mapped manner on PCIe and SoC FPGAs, for AMD and Altera FPGAs. This provides a clear target for always generating similarly structured memory-mapped accelerators that only need to conform to the AlmaIF control protocol.

The hardware integration methodology is explained more in the AlmaIF paper [26] and only briefly summarized here. First, a wrapper (block design) as shown in Fig. 7 is created for the IP. In the block design, a small soft processor core is added to control the accelerator IP and to communicate

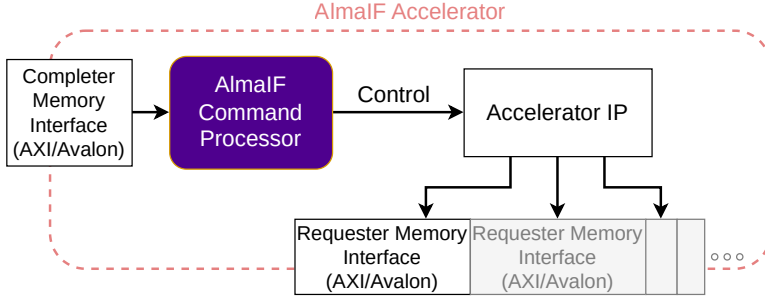


Fig. 7. Structure of a typical AlmalF accelerator. The Accelerator IP is generated by the proposed OpenCL C compiler toolchain. The AlmalF command processor implements the AlmalF interface, in order to facilitate communication with the OpenCL driver.

with the OpenCL driver. The OpenCL driver produces *packets* that contain the launch information of the kernel enqueue command. The soft processor reads the *packet*, launches the accelerator IP to compute the kernel, waits for its completion, and then marks the *packet* as completed.

Once the kernel has been wrapped as an AlmalF accelerator, it is passed to FPGA vendor tools to be integrated as part of their partially reconfigurable *shell*. For AMD FPGAs, *Vitis* is used with *V++*-command to link the custom accelerator to the FPGA platform shell. For Altera FPGAs, *Quartus* has to be used to perform this part of the flow. Since this part of the toolchain is still dependent on vendor tooling, using an open source shell, such as Coyote [37] or FOS [42] could reduce reliance on closed source tools.

5 Evaluation

The proposed toolchain is evaluated on OpenCL benchmarks to find out how it compares against commercial implementations and to measure the effect of command buffer specialization. Additionally, a demonstration in which the proposed toolchain is integrated into a higher-level framework, OpenVX, is included to showcase the benefits of hierarchical platform design based on open standards. All evaluations are performed on real FPGA hardware in order to extract the most realistic runtime numbers, and to demonstrate the functionality of the proposed toolchain. Before measurements are taken, the benchmarks are executed once in order for the FPGA bitstreams to be generated in the cache.

5.1 Command Buffer Specialization

To distinguish whether the performance of the proposed toolchain is due to additional MLIR optimizations performed or due to the specialization available for command buffer synthesis, an evaluation is performed in which the specialization is reduced but the flow is otherwise kept the same. The proposed toolchain with the Hida backend is compared to AMD's Vitis OpenCL implementation. The comparisons are made by running the PolybenchGPU benchmark on the AMD Alveo U280 FPGA using Vivado 2022.1 as the FPGA synthesis tool.

For the non-specialized results, the entire OpenCL program is synthesized at OpenCL *clBuild-Program*-call. For these results seen in the column *Proposed toolchain, no cmd buffers*, the original polybenchGPU benchmarks are used. Every kernel is synthesized as an independent accelerator IP (see Fig.7), and a single command processor controls all of them. In principle, the same amount of

Table 1. PolybenchGPU runtime results in seconds for the *standard* dataset on AMD Alveo U280. For each benchmark, it is listed whether their command buffer consists of one single kernel or multiple kernels, and whether they require mutable command buffers. 3DCONV produces wrong results with Vitis OpenCL. ADI benchmark does not have a command buffer implementation due to its complexity, and is not included in the geometric mean in order to make geometric means comparable.

	Vitis OpenCL no cmd buffers	Proposed toolchain			Mutable global size	Mut. args.	Multiple kernels
		no cmd buffers	no kernel fusion	w/ kernel fusion			
2DCONV	0.75	0.68 (0.90)	0.67 (0.89)	0.67 (0.89)			
2MM	567.04	473.09 (0.83)	463.09 (0.82)	462.79 (0.82)			x
3DCONV	6.01 (x)	5.6 (0.93)	5.47 (0.91)	5.48 (0.91)		x	
3MM	13.04	9.42 (0.72)	9.2 (0.71)	9.2 (0.71)			x
ADI	1.97	2.87 (1.46)	-	-			x
ATAX	2.55	1.23 (0.48)	1.23 (0.48)	1.23 (0.48)			x
BICG	2.55	1.23 (0.48)	1.23 (0.48)	1.23 (0.48)			x
CORR	128.14	131.27 (1.02)	130.35 (1.02)	128.15 (1.00)			x
COVAR	127.01	127.54 (1.00)	129.77 (1.02)	127.49 (1.00)			x
FDTD-2D	1186.87	3379.89 (2.85)	3349.59 (2.82)	3348.87 (2.82)		x	x
GEMM	4.41	3.21 (0.73)	3.07 (0.70)	3.07 (0.70)			
GEMVER	6.51	11.14 (1.71)	10.86 (1.67)	11.06 (1.70)			x
GESUMMV	2.52	0.34 (0.14)	0.34 (0.13)	0.34 (0.13)			
GRAMSCHM	2296.33	2299.21 (1.00)	2298.02 (1.00)	2296.86 (1.00)	x	x	x
JACOBI1D	13.59	14.88 (1.09)	10.92 (0.80)	10.66 (0.78)			x
JACOBI2D	111.52	118.83 (1.07)	115.63 (1.04)	115.62 (1.04)			x
LU	471.41	1629.51 (3.46)	1616.11 (3.43)	1616.14 (3.43)	x	x	x
MVT	2.55	1.23 (0.48)	1.23 (0.48)	1.23 (0.48)			x
SYR2K	134.48	75.17 (0.56)	28.62 (0.21)	28.59 (0.21)			
SYRK	47.41	46.52 (0.98)	23.77 (0.50)	23.77 (0.50)			
Geo. mean (19/20)	29.44	25.44	22.72	22.67			

information is available to the compiler than for the comparison results collected with AMD's Vitis OpenCL implementation.

From Table 1, it can be seen that even when not using command buffers, the proposed toolchain produces a better geometric mean across the benchmarks than Vitis OpenCL, being faster in 10/20 benchmarks, slower in 6/20, and within 5% of runtime in the rest. This shows that the generic MLIR optimizations used have a beneficial effect on performance.

Then, the PolybenchGPU benchmarks (except *ADI*) are converted to use the OpenCL command buffer extension. The modified source code for the PolybenchGPU benchmark is also published². There is no concise way to represent the *ADI* benchmark using a command buffer, other than pushing every single command invocation to a command buffer, which would lead to having thousands of command invocations, which the current version of the toolchain is not able to handle. Supporting very large command buffers would require the implementation to synthesize either non-specialized kernels, or to make the specialization more intelligent such that it could allow hardware sharing between different command invocations of the same kernel.

The results *with command buffer but without kernel fusion* are collected by turning off the *kernel fusion*, but still allowing the proposed compiler to apply the command buffer-specific specializations for kernels' scalar arguments, global, and local sizes. If the benchmark program requires mutable command buffers, it may not benefit from this specialization. The specialization gives a performance improvement across most of the benchmarks, since especially the statically known loop bounds enable the compiler to make better optimizations related to e.g. unrolling and vectorization. The

²Source code for the PolybenchGPU with OpenCL command buffers: <https://github.com/cpc/polybench>

geometric mean is clearly improved, and the proposed method is faster on 12/20 benchmarks, slower, or not able to implement on 4/20 and approximately equal on the rest.

Finally, *kernel fusion* is enabled as well, in which all the kernel invocations of the command buffer are flattened inside a single kernel, which is then passed for optimization. This is only relevant for benchmarks that have multiple kernels as signified by the *Multiple kernels*-column. Perhaps a bit surprisingly, the geometric mean does not improve significantly. This shows that the proposed toolchain is not able to take full advantage of the now available inter-kernel optimizations. To help explain this, the multi-kernel benchmark programs have multiple kernels for a valid reason, which is to achieve inter-kernel synchronization. Based on the OpenCL command queue synchronization, the previous kernel must be fully executed until the next one is launched. Therefore, any optimizations would need to take into account the potentially complex memory dependencies to determine whether some logic could be moved between the kernels, or whether some loops can be fused, etc. Further research into inter-kernel optimization would be needed, since we estimate there would be potential for some optimizations that are not currently found by the proposed flow.

5.2 Backend Evaluation

Since the proposed flow supports multiple backends, it provides a great opportunity to benchmark different FPGA devices and HLS backend tools to see how well they perform the task of generating accelerators from specialized command buffers.

First, Vitis HLS [4], Hida [46], and Bambu [13] are compared against each other on the AMD Alveo U280 FPGA using Vivado 2022.1 as the backend HLS tool. From Table 2 it can be seen that applying Hida optimizations before passing the program to Vitis HLS does not provide any significant performance increase.

The Hida publication [46] demonstrated impressive speedup compared to Vitis HLS using the C-version of the Polybench benchmark set. In this work, PolybenchGPU is used, which has slightly different kernels and much larger dataset sizes than the CPU version. The inability for Hida to reach significant improvement is likely due to the larger datasizes since now the data must be fetched from external memory, whereas in the original publication, all the data could fit into the FPGA's on-chip memory. Additionally, Hida had to be modified in order to get all the benchmarks to synthesize and produce correct results on the FPGA.

A custom pipeline of Hida passes based on Hida's *C++ Pipeline* was used to generate the Hida results shown. Hida also includes a potentially more effective *design-space exploration*-engine, but it is not able to synthesize most of the benchmarks due to the DSE process taking too long or running out of memory. This perhaps demonstrates an inability of Hida to scale up to larger dataset sizes, or at least that further research into it is needed to take advantage of its promised dataflow optimizations.

Initially with Hida, a significant performance regression was found in the CORR and COVAR benchmarks. This was determined to be caused by an overly eager *loop perfection* pass of Hida. Shown in Fig. 8, the loop perfection pass perfectly nests the two loops, but as a consequence, this leads to a major regression in Vitis HLS's ability to schedule the loops. This is likely due to the innermost loop now having to conditionally write to a second memory location in the *symmat*-array, which causes a complicated memory dependency constraint. To deal with this, Vitis HLS increased the initiation interval to 148 cycles, which led to a major performance regression. To work around this, in the reported results, Hida's loop perfection pass is disabled. This allows Vitis HLS to schedule the inner loop with an initiation interval of 6. Naturally, disabling the optimization can hinder further optimization passes. But it is the simplest workaround, until a better method is found for determining whether loop perfection should be performed, perhaps based on the Hida's DSE engine.

Table 2. PolybenchGPU runtime results in seconds for the command buffer implementation using different HLS back-ends. The best runtime for each device is highlighted in green. On the Altera’s FPGA and with Bambu HLS some benchmarks either do not synthesize, run indefinitely, or produce incorrect results (marked with an ‘x’). The geometric mean is computed using only the benchmarks that all the back-ends are able to correctly execute (rows with no x’s) in order to make them comparable. The same dataset is used for both devices, making the results comparable across device vendors.

	Alveo U280 (AMD)				BittWare IA-420f (Altera)	
	Vendor OpenCL (no cmd buffers)	Vitis HLS	Hida + Vitis HLS	Bambu HLS	Vendor OpenCL (no cmd buffers)	Hida + Intel AOC
2DCONV	0.75	0.67	0.67	10.47	0.03	0.20
2MM	567.04	462.73	462.79	5009.71	307.87	270.03
3DCONV	6.01 (x)	5.48	5.48	54.91	0.19	x
3MM	13.04	9.20	9.20	112.36	5.33	6.77
ATAx	2.55	1.23	1.23	10.17	0.36	0.29
BICG	2.55	1.23	1.23	10.18	0.39	0.29
CORR	128.14	128.16	128.15	x	59.37	124.42
COVAR	127.01	127.44	127.49	2396.08	80.36	91.56
FDTD-2D	1186.87	3349.34	3348.87	3461.36	x	x
GEMM	4.41	3.07	3.07	37.54	1.60	2.66
GEMVER	6.51	11.35	11.06	21.40	0.39	15.47
GESUMMV	2.52	0.34	0.34	6.54	0.76	0.51
GRAMSCHM	2296.33	2293.14	2296.86	x	x	4.22
JACOBI1D	13.59	10.69	10.66	64.32	0.51	2.58
JACOBI2D	111.52	115.59	115.62	686.38	1.76	1.19
LU	471.41	1615.89	1616.14	x	5.41	4.11
MVT	2.55	1.23	1.23	10.76	0.36	0.29
SYR2K	134.48	28.62	28.59	623.76	64.23	14.13
SYRK	47.41	23.78	23.77	532.50	28.08	17.77
Geo. mean (14/19)	13.70	8.25	8.23	86.01	2.35	3.23

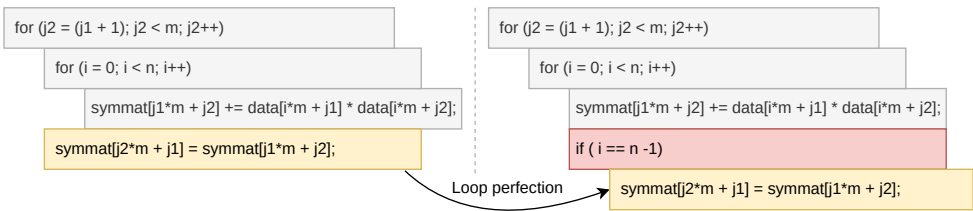


Fig. 8. The loop perfection pass of Hida that degrades the performance of CORR and COVAR benchmarks by increasing the initiation interval of the loop from 6 to 148 cycles.

Another Hida optimization that had to be disabled was the additional duplication of the kernel arguments. Hida duplicates arguments that are used multiple times in a loop and then calls the function with the same pointer value in all of its duplicated arguments. This is done in order to eliminate contention of the memory interface, which in turn helps Vitis HLS’s loop pipelining, since the accesses are split to independent interfaces. However, according to Vitis HLS documentation, Vitis HLS assumes that pointer function arguments never alias, which puts Hida in violation of this constraint, and this optimization had to be disabled.

Table 3. Arithmetic means of resource usage and maximum clock frequency of the proposed toolchain compared to Vitis OpenCL implementation on polybenchGPU benchmarks on Alveo U280. The utilization ratio out of the total available resources is in brackets. Vitis’s default target clock frequency of 300 MHz is used in all the runs.

	LUT	BRAM	DSP	Max. clock freq.
Vitis OpenCL	7257.1 (0.61%)	2.6 (0.14%)	33.4 (0.37%)	300 MHz
Proposed w/ Hida+Vitis HLS	10631.4 (0.90%)	24.3 (1.33%)	17.2 (0.19%)	300 MHz
Proposed w/ Bambu HLS	5102.2 (0.43%)	10.0 (0.55%)	10.0 (0.11%)	126 MHz

Table 4. Arithmetic means (\pm standard deviation) of PolybenchGPU compilation times of the proposed toolchain compared to Vitis OpenCL implementation targeting Alveo U280 FPGA. The first row of values is from Vitis OpenCL implementation. The runtimes for the proposed toolflow are broken down by the high-level structure of the MLIR-based compiler. The compiler runtimes for the Hida- and Bambu-based flows diverge after the middle-end passes. The ‘wrapper’-column refers to the creation of a block design using Vivado (the blue ‘Wrap.’-box in Fig. 6).

		Compiler runtime from OpenCL C to RTL					RTL to bitstream
Vitis OpenCL		69.4 \pm 12.7 s					
Proposed toolflow	Polygeist frontend	MLIR middle-end passes	Hida	Vitis HLS	Wrapper	Total	~2 hours
	4.8 \pm 0.2 s	0.5 \pm 0.1 s	3.2 \pm 1.5 s	54.9 \pm 3.9 s	25.8 \pm 0.3 s	89.1 \pm 4.8 s	
			Bambu HLS		Wrapper	Total	
			7.9 \pm 1.1 s		28.4 \pm 0.4 s	41.6 \pm 1.4 s	

Comparison to Bambu HLS [13] shows that the academic HLS tool is not yet at the level of commercial Vitis HLS. The runtime results are consistently slower than Vitis HLS. This is partially explained by the consistently lower maximum clock frequency and the lower resource usage visible in Table 3. However, the source code of the tool is open-source, which makes it possible to build future research on ways to improve the performance. This would not be possible with closed-source vendor-provided HLS tools.

The FPGA resource usage is an internal implementation detail that is not highly relevant to a higher-level programmer. It is enough that the automatically generated circuit fits on the FPGA. However, resource usage can offer insight into the compiler’s ability to utilize available FPGA resources. From Table 3 it can be seen that the proposed toolchain uses noticeably more LUT resources than Vitis OpenCL, which may help explain the slightly higher performance. However, from the utilization ratio of the FPGA, it can be seen that there is still significantly more logic available to be used. This motivates future research to focus on automatic spatial scaling to better take advantage of the parallelism available in the OpenCL program representation.

The compilation times for AMD Alveo U280 are measured and shown in Table 4. From the table it is clear that the bitstream generation dominates the total tool runtime for both the proposed approach and the AMD’s OpenCL implementation. The Hida-based flow is the slowest, which can be explained by the fact that in addition to applying its own optimizations, it uses Vitis HLS for the RTL generation. The Bambu HLS-based flow is the fastest as it does not rely on Vitis HLS. The back-end phase of the proposed toolflow in both Hida- and Bambu-based flows also includes the creation of a wrapper block design using Vivado, which consumes slightly less than half a minute.

The same benchmarks are then executed on the Intel Agilex 7-based Bittware IA-420f accelerator card using *Intel FPGA SDK for OpenCL (AOC) v.2024.1.0* and *Quartus v.21.4* as the back-end compiler. The first observation is that Altera’s IA-420f FPGA executes the non-modified OpenCL

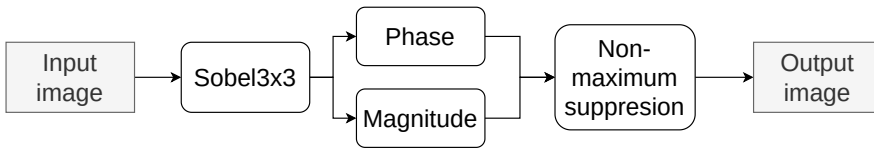


Fig. 9. The sample OpenVX program used to demonstrate the integration with higher-level frameworks. It performs part of the computation of a typical Canny computer vision algorithm.

Table 5. Execution runtime of the OpenVX program shown in Fig. 9 with 8-bit greyscale 3840x2160 image. The Vitis OpenCL runtime is from [27].

Vitis OpenCL	Proposed (non-specialized)	Proposed (specialized)
5.34s	5.40s	4.76s

benchmarks clearly faster than AMD’s Alveo U280. Furthermore, based on the geometric mean of the benchmarks, the proposed method using command buffers is slower than directly using *Intel FPGA SDK for OpenCL* with OpenCL C kernels. However, the proposed method is faster on 56% of the benchmarks, which makes it still competitive to the vendor’s OpenCL implementation.

5.3 OpenVX Demonstrator

The proposed OpenCL-based accelerator synthesis toolchain integrates well with higher-level frameworks. This is demonstrated by accelerating a computer vision application written in OpenVX [15]. The OpenVX application is executed with a custom OpenVX implementation that offloads the computation to the proposed OpenCL implementation prototype with the Hida backend and Alveo U280 FPGA. The task graph shown in Fig. 9 is built using the standard OpenVX API. Then the entire graph is verified with *vxVerifyGraph*-call. After that, the graph can be executed as many times as desired with the *vxProcessGraph*-call.

Our OpenVX implementation prototype supports OpenCL command buffers by mapping the *vxVerifyGraph* to OpenCL *clFinalizeCommandBufferKHR*, and the *vxProcessGraph* to *clEnqueueCommandBufferKHR*. In this way, the entire OpenVX graph can be optimized as a whole by the OpenCL implementation prototype.

The Vitis OpenCL numbers for comparison are collected by synthesizing an equivalent OpenCL program. The results using the proposed method use the OpenVX API to construct and execute the application. Similarly to the earlier specialization experiment, the non-specialized results are generated by synthesizing the OpenCL program at *clBuildProgram*-time, which gives the compiler the same amount of information as was used to generate the Vitis HLS comparison numbers. From Table 5 it can be seen that the runtime without the specialization is very close to the Vitis HLS results. The command buffer specialization and kernel fusion then help the proposed method to achieve noticeably higher performance compared to Vitis OpenCL.

6 Future Work

There are yet more improvements and features that could be built on top of the proposed open-source toolchain. In this section, some ideas for future research in accelerator-centric synthesis are listed.

Khronos has released an official set of conformance tests to ensure that the different OpenCL implementations are working as specified. Therefore, in order to prove the true portability of the

proposed toolchain, the OpenCL conformance tests are a clear target to pass for any aspiring OpenCL implementation. Since most of the conformance test suite programs do not use command buffers, this would limit the toolchain to not support OpenCL barriers on those programs without triggering FPGA reconfiguration between kernels. Therefore, the proposed toolchain could be extended to automatically infer command buffers from non-command buffer programs by deferring the compilation and enqueueing all the way until the control must be returned to the host program in *clFinish* or *clWaitForEvent*.

To fully liberate the heterogeneous FPGA programming from FPGA vendor tooling, the remaining closed-source components shown in Fig. 1 should eventually be replaced with open-source alternatives. The use of an open-source FPGA platform shell such as Coyote [37] would help with this goal. There are a few open-source FPGA synthesis works such as Platypus [47] and F4FPGA [8] that make it possible to generate also the FPGA bitstreams with open-source tools. However, since their work is mostly focused on very small FPGAs, it is unlikely that one is able to program larger FPGA accelerator cards such as Alveo U280 with completely open-source flows in the near future.

In the proposed implementation the task pipelines were attempted to be constructed by the way of naive kernel fusion. The inter-kernel optimizations were left for the HLS compiler to discover. However, the OpenCL standard specifies an explicit way for describing task pipelines with OpenCL pipes. Supporting these in the proposed toolchain could be a way to unlock higher performance without relying on the compiler to discover the dataflow parallelism. As the kernel arguments are known at the command buffer finalization time, the full OpenCL pipe connectivity is also available. This could allow the use of statically sized FIFOs as pipes, which is the most efficient way to implement OpenCL pipes on FPGA [27].

In addition, support for efficient off-chip data movement needs to be improved to achieve more competitive results. The current implementation does not perform any data transfer optimizations as Hida does not attempt to optimize that aspect. In typical computing systems, this issue is often solved by explicit or implicit data caching. Analyzing the data access patterns could enable specialized pre-fetching, coalescing, and caching of data. An interesting approach to this is in AXI4MLIR [2]. It attempts to optimize external memory accesses and is capable of generating AXI interfaces in MLIR representation directly. Integrating AXI4MLIR into the proposed tool could enable higher performance by more efficient use of external memory bandwidth.

7 Conclusions

We introduced an open-source MLIR-based toolchain for generating FPGA accelerators from portable OpenCL input. The toolchain offers an end-to-end accelerator synthesis flow, in which the software programmer is not required to manage the FPGA in-detail. The OpenCL implementation prototype supports many advanced OpenCL features, such as kernel functions, barriers, and local memory. Additionally, to the authors' best knowledge, this work is the first to utilize OpenCL command buffers to automatically generate complete FPGA bitstreams. In our work, we showed how this enables specialization and kernel fusion without requiring the use of tool-specific pragmas, as is typically the case with HLS tooling. We also proposed a way to manage the command buffer mutability in an efficient manner without having to reconfigure the FPGA.

The toolchain is built on top of a cross-vendor portable hardware integration methodology. This can help to reduce *vendor lock-in*, since user applications remain portable to FPGAs from different vendors. The use of standard OpenCL input allows it to be easily integrated into higher-level parallel programming frameworks. This was demonstrated by the use of the proposed toolchain within an OpenVX implementation. With the PolybenchGPU benchmarks, we showed competitive results on both AMD and Altera FPGAs compared to the OpenCL implementations from the vendors. We

believe that due to the modular nature of the toolchain, it can serve as a collaborative project to further the development of open-source tooling in the domain of automated accelerator synthesis.

References

- [1] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Kaeli, and Antonino Tumeo. 2022. An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (San Diego, California) (ICCAD '22)*. ACM, New York, NY, USA, Article 6, 9 pages. doi:10.1145/3508352.3549424
- [2] Nicolas Bohm Agostini, Jude Haris, Perry Gibson, Malith Jayaweera, Norm Rubin, Antonino Tumeo, José L. Abellán, José Cano, and David Kaeli. 2024. AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (Edinburgh, United Kingdom) (CGO '24)*. IEEE, 143–157. doi:10.1109/CGO57630.2024.10444801
- [3] Aksel Alpay and Vincent Heuveline. 2025. Adaptivity in AdaptiveCpp: Optimizing Performance by Leveraging Runtime Info During JIT-Compilation. In *Proceedings of the 13th International Workshop on OpenCL and SYCL (Heidelberg, Germany) (IWOC '25)*. ACM, New York, NY, USA, Article 2, 12 pages. doi:10.1145/3731125.3731127
- [4] AMD. 2022. Vitis HLS version 2022.1. Retrieved 28 April 2025 from <https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html>
- [5] Suhail Basalama and Jason Cong. 2025. Stream-HLS: Towards Automatic Dataflow Acceleration. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA, USA) (FPGA '25)*. ACM, New York, NY, USA, 103–114. doi:10.1145/3706628.3708878
- [6] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (June 2024), 28 pages. doi:10.1145/3656401
- [7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18) (Carlsbad, CA, USA)*. USENIX Association, Berkeley, CA, USA, 578–594.
- [8] F4FPGA contributors. 2022. F4FPGA. Retrieved 22 May 2025 from <https://f4pga.org/>
- [9] LLVM contributors. 2022. ClangIR (CIR). Retrieved 28 April 2025 from <https://llvm.github.io/clangir/>
- [10] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. 2012. From OpenCL to high-performance hardware on FPGAS. In *22nd International Conference on Field Programmable Logic and Applications (FPL) (Oslo, Norway)*. IEEE, 531–534. doi:10.1109/FPL.2012.6339272
- [11] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2021. Transformations of High-Level Synthesis Codes for High-Performance Computing. *IEEE Trans. Parallel Distrib. Syst.* 32, 5 (may 2021), 1014–1029. doi:10.1109/TPDS.2020.3039409
- [12] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. 2021. MLIR as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*, Vol. 3.
- [13] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC) (San Francisco, CA, USA)*. IEEE, 1327–1330. doi:10.1109/DAC18074.2021.9586110
- [14] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar) (San Jose, CA, USA)*. IEEE, 1–10. doi:10.1109/InPar.2012.6339595
- [15] Khronos® OpenVX Working Group. 2022. The OpenVX Specification 1.3.1. https://www.khronos.org/registry/OpenVX/specs/1.3.1/html/OpenVX_Specification_1_3_1.html.
- [16] Khronos® SYCL Working Group. 2025. SYCL™ 2020 Specification (revision 10). <https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>.
- [17] Intel. 2022. *Intel FPGA SDK for OpenCL Software Technology*. Intel. Retrieved 21 May 2025 from <https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html>
- [18] Intel. 2024. *oneAPI Specification 1.4*. Intel. Retrieved 19 May 2025 from <https://oneapi-spec.uxlfoundation.org/specifications/oneapi/v1.4-rev-1/oneAPI-spec.pdf>
- [19] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raikila, Jarmo Takala, and Heikki Berg. 2015. pocl: A Performance-Portable OpenCL Implementation. *Int. Journal of Parallel Programming* 43, 5 (2015), 752–785.

doi:10.1007/s10766-014-0320-y

- [20] Gangwon Jo, Heehoon Kim, Jeesoo Lee, and Jaejin Lee. 2020. SOFF: An OpenCL High-Level Synthesis Framework for FPGAs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (Valencia, Spain). IEEE, 295–308. doi:10.1109/ISCA45697.2020.00034
- [21] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) (*FPGA '18*). ACM, New York, NY, USA, 127–136. doi:10.1145/3174243.3174264
- [22] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2015. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). Springer International Publishing, Cham, 46–67.
- [23] Khronos® OpenCL Working Group. 2025. *The OpenCL™ Specification v3.0.18*. Khronos. Retrieved 9 April 2025 from https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf
- [24] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) (*FPGA '19*). ACM, New York, NY, USA, 242–251. doi:10.1145/3289602.3293910
- [25] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore’s Law. arXiv:2002.11054 [cs.PL] <https://arxiv.org/abs/2002.11054>
- [26] Topi Leppänen, Atro Lotvonen, Panagiotis Mousoulitotis, Joonas Multanen, Georgios Keramidas, and Pekka Jääskeläinen. 2023. Efficient OpenCL System Integration of Non-Blocking FPGA Accelerators. *Microprocess. Microsyst.* 97, C (mar 2023), 11 pages. doi:10.1016/j.micpro.2023.104772
- [27] Topi Leppänen, Joonas Multanen, Leevi Leppänen, and Pekka Jääskeläinen. 2024. Towards Efficient OpenCL Pipe Specification for Hardware Accelerators. In *Proceedings of the 12th International Workshop on OpenCL and SYCL* (Chicago, IL, USA) (*IWOCL '24*). ACM, New York, NY, USA, Article 2, 8 pages. doi:10.1145/3648115.3648128
- [28] Topi Leppänen, Leevi Leppänen, Joonas Multanen, and Pekka Jääskeläinen. 2024. Bitstream Database-Driven FPGA Programming Flow Based on Standard OpenCL. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 32, 12 (2024), 2257–2268. doi:10.1109/TVLSI.2024.3458062
- [29] Grant Martin and Gary Smith. 2009. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers* 26, 4 (2009), 18–25. doi:10.1109/MDT.2009.83
- [30] Sophie Won Metzger, Stephen Gabriel, and Matt Benson. 2025. Intel Announces Strategic Investment by Silver Lake in Altera. Retrieved 21 May 2025 from <https://www.intc.com/news-events/press-releases/detail/1736/intel-announces-strategic-investment-by-silver-lake-in>
- [31] Vincent Mirian and Paul Chow. 2015. 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, 1–6. doi:10.1109/ReConFig.2015.7393366
- [32] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) (*PACT '21*). ACM, New York, NY, USA, 12 pages.
- [33] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) (*PPoPP '23*). ACM, New York, NY, USA, 119–134. doi:10.1145/3572848.3577475
- [34] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D. Antonopoulos. 2011. Synthesis of Platform Architectures from OpenCL Programs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines* (Salt Lake City, UT, USA). IEEE, 186–193. doi:10.1109/FCCM.2011.19
- [35] S. John Pennycook and Jason D. Sewall. 2021. Revisiting a Metric for Performance Portability. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (St. Louis, MO, USA). IEEE, 1–9. doi:10.1109/P3HPC54578.2021.00004
- [36] Khoa Dang Pham, Anuj Vaishnav, Malte Vesper, and Dirk Koch. 2018. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers* (Dublin, Ireland). VDE-Verlag, Berlin, Germany, 1–9.

- [37] Benjamin Ramhorst, Dario Korolija, Maximilian Jakob Heer, Jonas Dann, Luhao Liu, and Gustavo Alonso. 2025. Coyote v2: Raising the Level of Abstraction for Data Center FPGAs. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) (*SOSP '25*). ACM, New York, NY, USA, 639–654. doi:10.1145/3731569.3764845
- [38] Kavya Shagrithaya, Krzysztof Kepa, and Peter Athanas. 2013. Enabling development of OpenCL applications on FPGA platforms. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors* (Washington, DC, USA). IEEE, 26–30. doi:10.1109/ASAP.2013.6567546
- [39] Jan Solanti and Pekka Jääskeläinen. 2025. Latency Reduction Potential of Server-Side Command Buffers in OpenCL-Based Edge Offloading. In *Proceedings of the 13th International Workshop on OpenCL and SYCL* (Heidelberg, Germany) (*IWOCL '25*). ACM, New York, NY, USA, Article 4, 4 pages. doi:10.1145/3731125.3731129
- [40] Emanuele Del Sozzo, Davide Conficconi, Alberto Zeni, Mirko Salaris, Donatella Sciuto, and Marco D. Santambrogio. 2022. Pushing the Level of Abstraction of Digital System Design: A Survey on How to Program FPGAs. *ACM Comput. Surv.* 55, 5, Article 106 (Dec. 2022), 48 pages. doi:10.1145/3532989
- [41] Anuj Vaishnav. 2018. Source code of the OpenCL driver for ZUCL. Retrieved 22 May 2025 from https://github.com/zucflpl/zucfl_fsp/blob/166e9ec45c98306197cedc0b2edacbbe6293789b/driver-scheduler/OpenCL_driver.c
- [42] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. 2020. FOS: A Modular FPGA Operating System for Dynamic Workloads. *ACM Trans. Reconfigurable Technol. Syst.* 13, 4, Article 20 (sep 2020), 28 pages. doi:10.1145/3405794
- [43] Malte Vesper, Dirk Koch, and Khoa Pham. 2017. PCIeHLS: an OpenCL HLS framework. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers* (Ghent, Belgium). VDE-Verlag, Berlin, Germany, 1–6.
- [44] Yuefei Wang, Wendong Mao, Lang Feng, Jin Sha, and Zhongfeng Wang. 2025. A CPU+FPGA OpenCL Heterogeneous Computing Platform for Multi-Kernel Pipeline. *ACM Trans. Des. Autom. Electron. Syst.* 30, 4, Article 67 (July 2025), 23 pages. doi:10.1145/3744922
- [45] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Seoul, Republic of Korea). IEEE, 741–755. doi:10.1109/HPCA53966.2022.00060
- [46] Hanchen Ye, Hyegang Jun, and Deming Chen. 2024. HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) (*ASPLOS '24*). ACM, New York, NY, USA, 215–230. doi:10.1145/3617232.3624850
- [47] ZeroASIC. 2025. Platypus Embedded FPGAs. Retrieved 22 May 2025 from <https://www.zeroasic.com/platypus>

Received 23 May 2025; revised 17 October 2025; accepted 15 December 2025