The Journal of Supercomputing (2021) 77:13995–14010
https://doi.org/10.1007/s11227-021-03792-7

Efficient heterogeneous programming with FPGAs using the Controller model

Gabriel Rodriguez-Canal[1] · Yuri Torres[2] · Francisco J. Andújar[2] · Arturo Gonzalez-Escribano[2]

Accepted: 1 April 2021 / Published online: 6 May 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract

The Controller model is a heterogeneous parallel programming model implemented as a library. It transparently manages the coordination, communication and kernel launching details on different heterogeneous computing devices. It exploits native or vendor specific programming models and compilers, such as OpenMP, CUDA or OpenCL, thus enabling the potential performance obtained by using them. This work discusses the integration of FPGAs in the Controller model, using high-level synthesis tools and OpenCL. A new Controller backend for FPGAs is presented based on a previous OpenCL backend for GPUs. We discuss new configuration parameters for FPGA kernels and key ideas to adapt the original OpenCL backend while maintaining the portability of the original model. We present an experimental study to compare performance and development effort metrics obtained with the Controller model, Intel oneAPI and reference codes directly programmed with OpenCL. The results show that using the Controller library has advantages and drawbacks compared with Intel oneAPI, while compared with OpenCL it highly reduces the programming effort with negligible performance overhead.

Keywords Parallel programming · FPGA · OpenCL · Heterogeneous computing

This research has been partially supported by the Spanish Ministerio de Economía, Industria y Competitividad and by the ERDF program of the European Union, PCAS Project (TIN2017-88614-R); by Junta de Castilla y Leon - FEDER Grants, PROPHET and PROPHET-2 Projects (VA082P17, VA226P20); MECD (Beca de Colaboración, Spain), ScottishPower Masters Scholarship, and Salvador de Madariaga/Fulbright Scholar Grant (PRX17/00674).

✉ Arturo Gonzalez-Escribano
arturo@infor.uva.es

Extended author information available on the last page of the article

1 Introduction

FPGAs (Field Programmable Gate Arrays) have attracted a wider scope of HPC researchers in the recent years thanks to new programming languages and techniques. Previously, they suffered from heavy and error-prone programming with HDLs (Hardware Description Language) such as Verilog [1] or VHDL [2]. With these tools, the user describes the computation as a high-level procedural program in C-like languages, but needs to deal with many pitfalls associated with hardware design, such as avoiding metastability when crossing clock domains or communicating with memory controllers to access data. Thus, they require deep knowledge of electronics, which hinder their adoption by the HPC community. This has changed in the recent years thanks to the development of HLS (High Level Synthesis) languages and frameworks, such as the Intel FPGA SDK for OpenCL by Intel [3] or SDAccel by Xilinx [4], both based on OpenCL. HLS frameworks internally deal with the pitfalls associated with hardware design. Thus, the user can easily port algorithms implemented with C-like codes, that are seamlessly translated by the tools into HDL code, or working hardware on the FPGA. There are even higher-level approaches built on top of HDL and HLS technologies, also focusing on portability of codes with other types of devices, such as CPUs and GPUs. A promising example is the Intel oneAPI framework [5] based on SYCL [6], an extension to C++ leveraging modern compiler technologies and programming language features. These approaches typically advocate a single source code for both host code and device kernels. In general, the current implementations rely on different and non-compatible compiler back-ends for different types of devices. Thus, the executables generated have limitations to choose the target device at run-time, and to operate with several combinations of devices.

A different approach can be found in the Controller [7, 8] model. It is a heterogeneous parallel programming model that enables performance portability across CPU-core sets (using OpenMP), GPUs of different types and vendors (using CUDA or OpenCL), and Xeon Phi accelerators. It is implemented as a classic C99 library. It integrates the use of different vendor specific technologies through different backends coordinated by the same runtime layer. It can generate executables which select the type of target device at runtime. Apart from generic portable kernels, the programmer can supply different versions of the same kernel optimized for different devices, using the low-level programming models if needed. The runtime chooses the most appropriate one for the selected device.

The use of Controller decreases development effort compared with directly using native or vendor specific technologies, with little to no overhead, by leveraging their vendor specific tools. A first approach to integrate FPGAs in Controller is presented in [9].

In this work, we present the following contributions: (1) a description of a new Controller backend for FPGAs, adapting a previous OpenCL backend designed for GPUs. It integrates the Intel technology and compiler for FPGAs using OpenCL, implementing all the Controller features. (2) A fully operational implementation

of host-tasks as another type of kernel asynchronously executed in the host. They are automatically synchronized by data dependencies with FPGA kernels or data transfers, and they can use system resources such as the input/output system, a feature needed in many streaming applications. (3) An experimental study that validates our proposal. This study compares programs developed with the Controller model with equivalents programmed with Intel oneAPI, and with OpenCL directly. It includes comparisons in terms of performance, and in terms of four well-known development effort metrics. The rest of the paper is organized as follows: Sect. 2 outlines the related work. Section 3 describes the solution developed to integrate FPGAs in our model. Section 4 shows the experimentation results that validate our proposal. Finally, Sect. 5 presents the conclusion.

2 Related work

There are several projects that target the development of a high-level heterogeneous programming model, that is portable and efficient. EngineCL [10] is an OpenCL-based runtime system that simplifies the co-execution of a single massive data-parallel kernel on all the devices of a heterogeneous system. The University of Thessaly's programming model [11] supports seamless execution on hybrid architectures with approximation semantics. Both models support the use of platforms with CPUs, GPUs and FPGAs, using OpenMP-like pragmas to annotate the program. They use OpenCL under the hood for every supported architecture. This can lead to higher overheads than using the vendor specific tools such as CUDA for Nvidia GPUs or OpenMP for CPUs.

EngineCL and SYCL [6] (including the DPC++ extension used in Intel oneAPI [5]) exploit modern C++ features such as variadic templates, initializer lists or rvalue references, leveraging sophisticated compiling techniques. The portability of the common kernel code may compromise the efficiency, and some device specific configurations, optimizations, or algorithm changes in the kernel, may be needed for specific devices such as FPGAs. Focusing on interoperability, there are implementations of SYCL for different low-level or vendor technologies. In general, they are in separated projects or the compilers should be recompiled to generate code for different devices. Thus, programs compiled for FPGAs do not support launching kernels for other type of devices. For example, the DPC++ compiler in the main Intel oneAPI project only supports Intel devices. Programs can be compiled to support one of two options: Only Intel FPGAs kernels; or kernels for both CPUs and integrated Intel GPUs. A different implementation of the DPC++ compiler implements a CUDA backend, but it generates executables that only support kernels for Nvidia GPUs. Other SYCL projects, such as HipSYCL [12], present a similar problem. The programmer should choose at compile time to support either CPUs, or Nvidia GPUs using CUDA, or AMD GPUs using ROCm.

There are other programming models or frameworks extending OpenMP or implementing OpenACC for FPGAs: OmpSs @FPGA [13], FPM [14], AnyHLS [15] or the OpenARC compiler [16]. They rely on non-standard directives to express parallelism and dependencies, and use a source-to-source compiler that translates

them into API calls to specific runtime systems. This complicates its usage in conjunction with other vendor or device specific compilers. Thus, interoperability is also difficult.

The current implementations of most of these programming frameworks do not yet include a fully operative model of host-tasks that are automatically synchronized by data dependencies, have access to the input/output system, and are interoperable with FPGA kernels. For example, in current Intel oneAPI beta 10, CPU kernels do not support I/O operations, kernels cannot be queued in both *cpu_selector* and *fpga_selector* in the same program, and possible workarounds such as using *host_accessor* also fail at runtime according to our tests. Until a fully operative implementation is provided, extra synchronizations are sometimes needed in the host to ensure correctness, losing opportunities to overlap host and device computations.

3 FPGA support in Controller

The original Controller model introduced the *Controller* entity. Each Controller object transparently manages the coordination and communication of the host code with one device. See white shaded elements in Fig. 1. Data structures on the host are managed with *HitTile* objects, which are a kind of fat-pointer that stores several meta-data including data sizes, data pointers in host and device, etc. The HitTile variables can be associated with a Controller and allocated in one side (internal variables), or in both host and device sides (tied variables). The host code launches requests to the Controller object queue. These requests include execution of device kernels, or data movements for tied variables from host to device or device to host. An execution policy module coordinates the execution of the requests, calling the internal API of the runtime backend that implements the proper technology for that

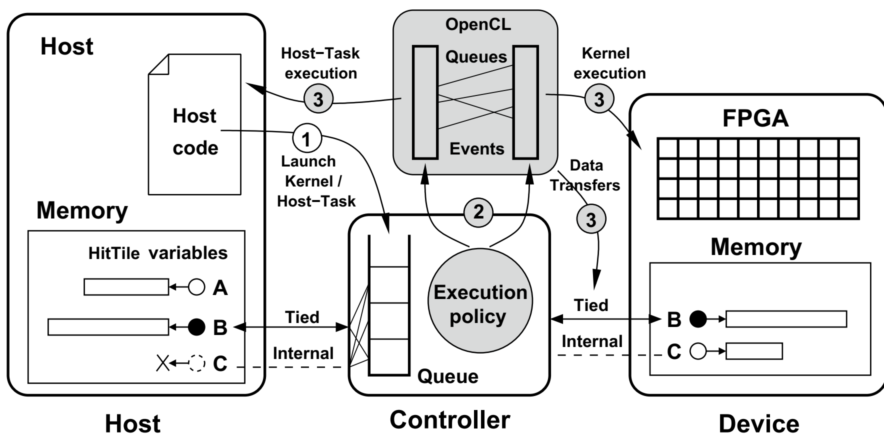


Fig. 1 Controller model for FPGA kernel and Host-Task synchronization. New or modified elements are shaded in grey color

device. Previous runtime backends were developed for CPU cores (using OpenMP), Xeon Phi coprocessors (using Intel LEO/COI libraries), Nvidia GPUs (using CUDA), and AMD/Nvidia GPUs (using OpenCL).

3.1 Design of the new backend

A new runtime backend is proposed in this work for FPGA devices. Developing a new runtime backend for Controller implies implementing its functionalities with the new technology. Basic features of the model should be preserved to keep the functional and performance portability of the compiled programs when using different types of devices: (a) Support of existing synchronous and asynchronous execution policies. (b) Uniform syntax and user interface for host codes, to allow porting Controller applications to the new device type by simply integrating the specific kernels. (c) Independence of the compiler. Controller is implemented as a library in a compiler agnostic way. (d) Performance. The resulting programs exploit the specific compiler and runtime technology for the device. Thus, the Controller programs for FPGAs should obtain the same performance as OpenCL programs specifically tuned for them.

The new FPGAs backend design is shown in Fig. 1. The circled numbers indicate the order in which operations are executed: (1) host-tasks or device-kernels execution requests are launched to the Controller queue. Both host-tasks and device kernels use the same launching interface. Data movement requests can be skipped in the host code, as they can be derived from data dependencies. (2) The execution-policy module calls the new backend to translate the requests into OpenCL commands. Host-tasks and kernels are queued in their respective OpenCL queues. The parameters used in the call are analyzed to determine data-dependencies with previous requests, based on the declared input/output role of the formal parameters. OpenCL data movement requests are queued if needed. OpenCL events are registered or updated, and synchronizations that depend on them are also added to the OpenCL queues. (3) The OpenCL driver takes care of the real execution of both FPGA kernels and host-tasks, and the synchronization of them with data movements.

For this purpose, the HitTile structure is extended with new information. First, it includes an enumerated type field to trace the memory status. This field records if after the execution of all the previous requests, the data of the HitTile is consistent on both memory hierarchies, or if the latest updated image is in the host or in the device. Second, the new HitTile includes a set of OpenCL events that can be used to register the finalization of previous OpenCL queued tasks, and to synchronize future OpenCL tasks with them. Events to register the end of read and write data operations in the host side, and in the device side are included. Each request for a kernel or host-task execution activates a simple check for each real parameter one by one. The memory status and the input/output role determine if the real parameter can be used safely in the host or device, or if a data movement from the other memory hierarchy is needed to get the most updated data. The events allow to introduce in the queues wait conditions to ensure that previous kernels/host-tasks have finished

to update data before starting to move it, and to ensure that required data movements are finished before the computation starts.

3.2 FPGAs integration in the Controller framework

FPGAs differ from the architectures previously supported in Controller in several aspects that affect their integration.

Limitations of offline compilation: The original OpenCL backend for GPUs leveraged the online compilation feature of OpenCL by conveying kernel code as strings. The mandatory offline compilation for FPGAs removes this possibility. Slight modifications of the original Controller API for kernel declarations have been introduced. The kernel code is declared in a block enclosed in curly brackets after the Controller prototype specification. In the new API, the input/output roles of parameters are specified on the kernel prototype declaration. Offline compilation requires separate files for host and kernel codes. Thus, the backend is designed to expect two different files. The code conversions needed in the kernels file to generate correct C code for the AOC compiler are directly done by the preprocessor.

Pipeline configuration: The pipeline configuration is a decision with great impact on performance and that is specific for each particular version of the kernel. For that reason, the kernel prototype declaration has been extended with a new optional clause called *PIPELINE*. It can be used to set the number of SIMD lanes and/or compute units, and choose between the NDrange or Task model of execution. This new clause must always appear in kernel signatures for FPGAs. An example of a prototype or signature declaration for a matrix multiplication kernel is shown in Listing 1, along with a kernel launching call. Listing 2 shows an equivalent code in DPC++. The Controller *PIPELINE* declaration is a compact equivalent to the DPC++ declarations *nd_range, nd_item, intel::num_simd_work_items,* and *intel::reqd_work_group_size*. The Controller *PARAMS* declaration substitutes the DPC++ creation of accessors when the kernel is submitted to a queue. The kernel parameters declaration in Controller is skipped in the DPC++ example by implementing the kernel as a lambda function that captures by reference all external variables, assuming that the kernel code is correct and does not introduce undesired side effects. In Controller, the possible parameters that *PIPELINE* can receive are: (1) TASK: Execute as a SWI (Single Work-Item or single-task) kernel; (2) NDRANGE: Execute as a NDrange kernel; (3) SIMD(M, wg_x, wg_y, wg_z): Execute in a pipeline with M lanes and workgroup dimensions (wg_x, wg_y, wg_z); and (4) CU(N): Execute in a pipeline replicated N times. Due to the restrictions in the lower level programming model,

TASK and *NDRANGE* cannot be used together in the same signature.

Listing 1 Example of Controller code for a matrix multiplication kernel

```
CTRL_KERNEL(Mult, FPGA, PIPELINE( NDRANGE,
    SIMD( simd_level , block_size , block_size , 1) ),
    PARAMS( OUT, IN , IN , INVAL, INVAL ),
    K_float C, K_float A, K_float B, int awidth , int bwidth) {
        ... kernel code ...
    }
Ctrl_launch(fpga1 , mat_range, Mult, c, b, a, size, size );
```

Listing 2 Example of DPC++ code to introduce the same kernel

```
fpga1_queue.submit([&](handler &h) {
    auto a_mat = a.get_access<access::mode::read>(h);
    auto b_mat = b.get_access<access::mode::read>(h));
    auto c_mat = c.get_access<access::mode::discard_write>(h);
    h.parallel_for<class mult> (nd_range<2>(mat_range, local_range),
        [=](nd_item<2> ij)
        [[intel::num_simd_work_items( simd_level ),
          cl::reqd_work_group_size( 1, block_size , block_size ),
          intel::kernel_args_restrict]]
          {
              ... kernel code ...
          });
    });
```

Support for the three execution modes: In addition to the ordinary execution mode, FPGAs support the profiling of kernels by instrumenting the pipeline with performance counters and the emulation on a CPU. Different object codes are generated for each mode using a different compiler flag. In Controller, different versions of the same kernel can coexist and be used from the same binary. Each time that an FPGA kernel is compiled in a specific mode, it is saved with an internal name indicating the mode. Thus, a kernel can be invoked in the desired mode and the runtime chooses the appropriate object code transparently. The same simple solution allows choosing at runtime the execution of instances of the same kernel with different compile-time parameterizations.

Divergences and support for incongruent grid sizes: A computation can be issued for a grid size that is not multiple of any of the block dimensions. As a result, the last block in that dimension will have some idle work-items to avoid, for example, accessing unallocated memory positions. The original OpenCL backend of Controller deals with this issue by conditionally executing the code only if the work-item index is inside the grid limits. This leads to a branch divergence. Although this solution works for GPUs, FPGAs cannot cope with branch divergence in SIMD pipelines. They are not synthesizable. The AOC compiler issues a warning and the target is compiled with a single SIMD lane.

The alternative approach used in our new backend is the introduction of padding to the images of the data structures on the device. This solution might lead to minor performance improvements in general, and can be extended to every architecture supported by Controller with minor effort. Thus, in our prototype all dimensions of a data structure are aligned to 256 elements. 256 work-items are the default dimensions assumed by AOC for a SIMD kernel with a synchronization point [3]. This

allows the use of SIMD with work-groups of up to 256 work-items in every dimension safely and transparently. The user does not need to change how the data elements are accessed in the kernel code.

4 Experimentation

This section discusses the experimentation conducted to verify that (1) our prototype implementation has a minimal effect on performance when compared to using the current OpenCL HLS technology for Intel FPGAs; and (2) the solution proposed offers a code implementation with much less programming effort than directly using OpenCL. We also conduct the same experimental work with DPC++ codes compiled with the Intel oneAPI framework, to show the advantages and drawbacks of our implementation. The full prototype implementation of Controller, along with all the application codes used for the experimentation are publicly available at https://trasgo.infor.uva.es/controller/.

4.1 Case studies

The experiments are conducted with three applications and different parameters in order to cover different scenarios related to load balance between data transfers and host/device computations, computation overlapping, etc. Two of them feature SWI kernels, and the other one an NDrange kernel.

Hotspot: This program is an adaptation of the Hotspot code in the Rodinia suite of benchmarks [17, 18]. It computes the stability point of the Poisson's partial differential equation (PDE) for heat diffusion in 2 dimensions. It is a 4-point stencil program that executes a fixed number of time iterations. The FPGA kernel selected exploits the shift register pattern for both spatial and temporal locality, and it is implemented as a SWI kernel. The program is tested with 400 iterations. The result matrix is transferred to the host after each device kernel operation, saving it in a different host buffer using a host-task. Thus, the results could be used to check partial results or to create an animation of the computation evolution. Matrices with input sizes from 1024×1024 to 4096×4096 are considered to test scenarios that range from more costly data transfers than computations, to more costly computations than data transfers.

Matrix power: This program is an evolution of the 2 and 3 mm programs in the PolyBench Benchmarks [19]. It generates a chain of matrix multiplications of arbitrary length k, using the following iterative process: $C^k = C^{k-1} \times A : k \in [1 : n]$ where $C^0 = A$. The kernel to multiply matrices is obtained from the Intel FPGA Support Resources. This optimized kernel uses local memory, loop unrolling and SIMD to take advantage of the FPGA resources. Each partial result C^i is transferred to the host. A host-task computes the normalization of the matrix and saves it in another buffer.

The program is tested with $k = 30$ iterations and input sizes from 640×640 to 2560×2560 to generate different computation ratios between host and device.

Sobel operator: The Sobel Operator is a common image processing filter. For this experimental study, we select an implementation that iteratively processes frames from a video in YUV format. The kernel was adapted from the one available at Intel FPGA Support Resources, also implemented as a SWI kernel. The program reads an input YUV video stream from a file, frame by frame. Each frame has three components that are communicated to the device. The Sobel filter is applied to each component. The resulting image is transferred back to the host to store it in an output video file. The computation of the Sobel filter is a very fast operation. Thus, this study case is very demanding in terms of concurrency exploitation and asynchronous data-transfer executions. The program is tested with 120, 240, 300 and 360 frames of a high-definition video (Full HD images of 1920×1080 pixels).

4.2 Experimental environment

For each case study, we have two reference OpenCL codes: a synchronous program that submits kernel and data-transfer operations to a single queue, and an asynchronous version that uses more advanced synchronization mechanisms to keep the correctness while overlapping operations. Controller and oneAPI codes are developed for each case study. In Controller codes, the choice between synchronous and asynchronous policy can be done at runtime. In oneAPI, the compiler analyzes the code and introduces automatically asynchronous operations. Due to the lack of support of a fully functional host-task implementation or interoperable CPU kernels while compiling for FPGAs, the programmer sometimes needs to include explicit synchronizations to coordinate host-code execution with data transfers.

For a fair comparison, both Controller and oneAPI programs use the same optimized kernel codes extracted from the references. Compilation and optimization parameters are the same, and the order of operations in the host code is also preserved. Table 1 presents the bitstream metrics for the kernels reported by the Intel AOC compiler. All kernels are compiled with a single compute unit. Hotspot and Sobel are implemented as SWI kernels, reporting initiation interval 1. Matrix multiplication kernel is implemented as NDrange, and we present results for two cases with different optimization/compilation parameters. Case (1) is tested for all programming models. It uses SIMD=8 and blocksize = 16×16. This is the best combination experimentally found that is supported by oneAPI. Case (2) uses SIMD=4 and blocksize = 64×64. This is the best combination experimentally found for the reference and Controller programs, but it fails to compile with oneAPI beta 10 compiler. Small differences in the values between reference and Controller versions are mainly derived from the different parameter encapsulation using HitTiles.

The experimentation was conducted on nodes of the Paderborn Center for Parallel Computing (PC^2). Each node features 2 Intel(R) Xeon(R) Gold 6148 CPU @ 2.40 GHz (40 cores), with 192 GB of RAM, and two Intel Stratix 10 GX 2800 FPGA with 32 GB of memory (Nittware 520 N cards). The O.S. is a CentOS 7 Linux distribution. OpenCL and Controller programs are compiled with GCC v8.1.0 using the Intel FPGA SDK for OpenCL, AOC v20.3. The oneAPI programs are compiled with the oneAPI Beta Update 10. The performance experiments measure

Table 1 Bitstream metrics for the kernels reported by the Intel AOC compiler

Version	ALUTs	Regs.	Logic util. (%)	I/O pins (%)	DSP Blk. (%)	Mem. bits (%)	RAM Blk. (%)	Freq. (MHz)	Fanout
Hotspot									
Ref	108,882	247,152	17	69	1	4	6	397.61	1572
oneAPI	110,053	248,964	17	69	1	4	6	397.61	1572
Ctrl	108,836	247,388	17	69	1	4	6	397.61	1572
Matrix pow									
Ref (1)	121,402	278,239	19	69	2	4	7	335.68	31776
oneAPI (1)	122,470	284,907	19	69	2	9	14	327.01	12321
Ctrl (1)	121,415	280,173	19	69	2	4	7	341.18	31776
Ref (2)	117,159	290,458	18	69	5	7	12	347.22	65568
Ctrl (2)	117,173	291,637	19	69	5	7	12	341.29	65568
Sobel filter									
Ref	98,122	210,622	15	69	0	3	5	397.61	1572
oneAPI	98,199	211,231	15	69	0	3	5	397.61	1572
Ctrl	97,351	208,675	15	69	0	3	5	397.61	1572

the clock time from the start of the first data transfer to the end of the last host-task. This includes data transfers, computation times, and system overheads.

4.3 Performance results

Each experiment is repeated 10 times, and average execution times are obtained. Figure 2 shows the experimental results for: (1) different input sizes (Sobel program is skipped as it works with images of fixed resolution); and (2) different number of iterations. Table 2 summarizes the overheads and percentage of overlap of the asynchronous version of each code in comparison with the corresponding synchronous one using the same programming model.

The plots and table show very similar performance results for the Controller programs in comparison with their respective OpenCL counterparts. For some programs and input sizes, the Controller programs show a slight positive overhead, while for some others, and specially in the Sobel case, the overhead is negative, indicating a better performance of the Controller version.

The results for oneAPI in the case of Hotspot show a better performance than the OpenCL reference and Controller when comparing synchronous codes. In the case of asynchronous codes, it depends on the input size. For some sizes, oneAPI performs much better, and for others (3000 and 3500), the oneAPI program is slightly outperformed by the reference and Controller version. The reason is that the oneAPI compiler is automatically transforming and optimizing the kernel, generating a different pipeline as reported by the Intel AOC compiler. This indicates that the reference code is not using the best possible kernel. The compiler technology in oneAPI is an advantage in case of kernels easily analyzable and optimizable at compile time. Controller needs to be provided with a better kernel to improve the oneAPI results. Kernels up to three times faster than the one in the reference program have been reported for this application [20].

The performance of the oneAPI matrix power program, using the same compilation/optimization parameters than the reference or Controller programs, changes with the input size. The trend is that it is slower for small input data sizes, but faster when we arrive at the biggest input size. This indicates that oneAPI transformations are specially appropriate for big data sizes, and not so optimized for small ones. Nevertheless, in this case the compiler technology of oneAPI generates a more complex kernel that exhausts the layout space of the FPGAs for big block sizes. The higher the block size, the more data reutilization and performance. The Controller model can always use exactly the same kernel as the reference without any modification, that in this case can be up to 64×64. For this application, oneAPI fails to compile for block sizes higher than 16x16. The best combination of optimization parameters for the reference and Controller programs completely outperforms the best oneAPI program, being 7.5–10 times faster.

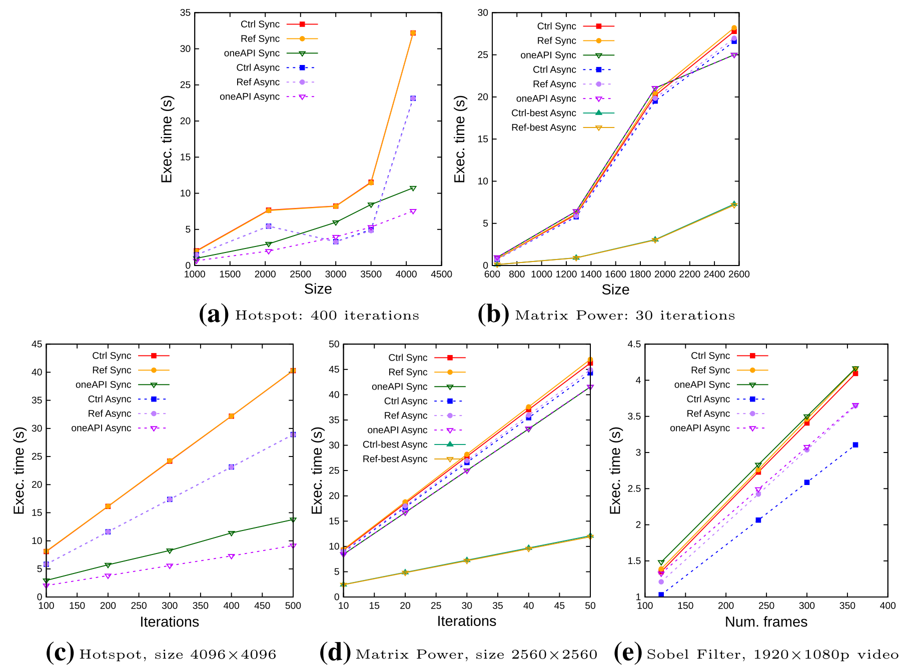
Finally, in the case of the Sobel filter, the oneAPI version presents a slightly lower performance than the reference counterparts, in both synchronous and asynchronous versions. An extra synchronization is needed to coordinate the reading of frames from the file with the data transfer and the kernel launching to process that frame.

Table 2 Performance comparison: (1) overheads observed in the Controller and oneAPI programs when compared to the reference OpenCL programs; and (2) percentage of overlap of data transfer and computation operations between the synchronous and asynchronous versions of each code

Hotspot				
Size	Impl.	Overhead Sync (%)	Overhead Async (%)	Overlap (%)
1024	Ref			25.12
	Ctrl	3.58	0.51	27.34
	oneAPI	−49.13	−54.65	33.25
2048	Ref			28.56
	Ctrl	0.78	0.20	28.97
	oneAPI	−60.49	−63.15	33.38
3000	Ref			60.06
	Ctrl	0.39	0.96	59.83
	oneAPI	−27.05	21.05	33.71
3500	Ref			57.89
	Ctrl	0.68	2.93	56.95
	oneAPI	−26.27	10.23	37.04
4096	Ref			28.16
	Ctrl	−0.05	0.04	28.10
	oneAPI	−66.62	−67.32	29.67
Matrix power				
640	Ref			9.11
	Ctrl	−1.67	−1.73	9.17
	oneAPI	19.24	33.40	−1.68
1280	Ref			4.73
	Ctrl	−1.68	−1.67	4.73
	oneAPI	4.33	9.57	−0.05
1920	Ref			3.22
	Ctrl	−1.64	−1.68	3.27
	oneAPI	2.68	6.12	−0.02
2560	Ref			4.39
	Ctrl	−1.52	−1.37	4.24
	oneAPI	−11.36	−7.31	0.02
Sobel operator				
N. fr.	Impl.	Overhead Sync	Overhead Async	Overlap
120	Ref			12.88
	Ctrl	−2.77	−14.75	23.62
	oneAPI	6.81	9.51	10.68
240	Ref			12.25
	Ctrl	−1.21	−14.81	24.33
	oneAPI	2.49	2.82	11.97

Table 2 (continued)

Sobel operator				
N. fr.	Impl.	Overhead Sync	Overhead Async	Overlap
300	Ref			12.30
	Ctrl	−1.54	−14.79	24.11
	oneAPI	1.13	1.38	12.09
360	Ref			12.33
	Ctrl	−1.56	−14.81	24.13
	oneAPI	0.17	0.33	12.19



(a) Hotspot: 400 iterations **(b)** Matrix Power: 30 iterations **(c)** Hotspot, size 4096×4096 **(d)** Matrix Power, size 2560×2560 **(e)** Sobel Filter, 1920×1080p video

Fig. 2 Experimental results for the case studies with different input sizes (Hotspot and Matrix Power) and different number of iterations with a fixed input size. In Matrix Power, *Ctrl-best* and *Ref-best* are generated with the best block size experimentally found (64 × 64), while the rest are generated with the best size supported by oneAPI (16 × 16)

This introduces an extra cost and prevents some proper operations from overlapping. The asynchronous Controller version clearly outperforms all the other ones. The synchronization and implementation mechanisms used in oneAPI are not as efficient as those used in the Controller model, that transforms the dependencies to the best chain of OpenCL event conditions in the low level queues as soon as the host code inserts the requests in the queue. This effect is noticeable in these kind of demanding applications with bursts of very fast small kernels and data transfers.

In summary, sophisticated compiler technology is sometimes an advantage, specially for non-specialized programmers. But, the possibility to directly integrate optimized kernels using the lower-level or vendor technology ensures the best performance in the long term. The Controller implementation of asynchronous operations and host-task model provides a fully operational and efficient form to overlap host computations with device kernels and data transfers.

4.4 Development effort measures

This section analyzes the differences in development effort after beautifying all the codes in the same way. Four classical development effort metrics are measured: number of lines of code, number of tokens, McCabe's cyclomatic complexity [21] and Halstead development effort [22]. The first two metrics measure the volume of code that the programmer should develop. The third metric measures the rational effort needed to program it in terms of code divergences and potential issues that should be considered to develop, test and debug the program. The last metric uses both code complexity and volume indicators to obtain a comprehensive measure of the development effort. The measured codes include the kernel definitions, kernel characterization, the coordination host code, and data structures management.

The results shown in Table 3 indicate that programming using the Controller library generates lower volume of code, a reduced cyclomatic complexity, and reduced Halstead measures than both, synchronous and asynchronous versions using OpenCL. This is specially significant for the asynchronous baseline versions, that introduce manually more complex mechanisms for kernel and data transfer synchronizations. A close look at the codes indicates that the highest reduction is found in the parts of the host codes related to coordination, as expected. The results for oneAPI are even better for the Hotspot and Matrix Pow programs. Nevertheless, the

Table 3 Measurements of development effort metrics: lines of code (LOC), number of code tokens (TOK), McCabe's cyclomatic complexity (CCN) and the Halstead's development effort metric (Halstead)

Case study	Version	LOC	TOK	CCN	Halstead
Hotspot	Ctrl	230	1772	40	919,321
	oneAPI Sync	198	1699	40	882,338
	oneAPI Async	196	1687	40	876,010
	Ref Sync	339	2771	57	1,770,315
	Ref Async	401	3273	53	2,332,285
Matrix Pow	Ctrl	148	1509	21	525,721
	oneAPI Sync	120	1193	22	483,196
	oneAPI Async	119	1187	22	478,754
	Ref Sync	211	1922	30	1,243,644
	Ref Async	270	2348	29	1,646,456
Sobel Filter	Ctrl	137	1231	22	907,566
	oneAPI Sync	158	1527	24	913,728
	oneAPI Async	156	1515	24	900,674
	Ref Sync	202	1944	28	1,207,349
	Ref Async	290	2561	38	1,689,124

Controller version of the Sobel operator program shows better development effort measures than even oneAPI. The more complex the synchronization structure of the program, the better results of development effort for Controller.

5 Conclusion

In this work, we present an extension of the Controller model with a new backend that supports FPGAs. The experimental results show that it achieves similar performance and overlap of computation with data transfers than manually optimized OpenCL versions. The asynchronous policy of Controller achieves overheads of less than 1% in most of the cases, and presents lower execution times than the baseline implementations in some scenarios. Programming effort is significantly lower in all the studied metrics with respect to the native OpenCL implementation. A comparison with oneAPI shows some conceptual and practical advantages and drawbacks that have an important impact in performance.

Finally, on-going and future work include: (1) a deeper study with more complex applications including several different kernels and more dynamic work-flows; (2) exploring the use of the OpenCL *pipe* feature to improve asynchronous overlapping when using more than one kernel in the same program, and (3) studying the possibility to introduce the ideas proposed in this paper into DPC++ or other SYCL compilers.

References

1. McNamara M (2008) IEEE standard verilog hardware description language. https://www.verilog.com/IEEEVerilog.html. Accessed 9 Jun 2020
2. IEEE Design Automation Standards Committee and others (2008) Std 1076–2008. IEEE Standard VHDL Language Reference Manual, IEEE, New York
3. Intel (2019) Intel FPGA SDK for OpenCL Programming Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl_programming_guide-20-1.pdf. Accessed 15 Aug 2020
4. Xilinx (2019) SDAccel Programmers Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1277-sdaccel-programmers-guide.pdf. Accessed 15 Aug 2020
5. Intel (2020) Intel oneAPI Specs. https://www.oneapi.com/spec/. Accessed 12 Sept 2020
6. Keryell R, Rovatsou M, Howes L (2020) SYCL specification—generic heterogeneous computing for modern C++. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf. Accessed 15 Aug 2020
7. Moreton-Fernandez A, Ortega-Arranz H, Gonzalez-Escribano A (2018) Controllers: an abstraction to ease the use of hardware accelerators. Int J High Perfor Comput Appl 32(6):838–853
8. Moreton-Fernandez A, Gonzalez-Escribano A, Llanos DR (2019) Multi-device controllers: a library to simplify parallel heterogeneous programming. Int J Parallel Progr 47(1):94–113
9. Rodriguez-Canal G, Torres Y, Gonzalez-Escribano A (2020) Integrating FPGAs in a heterogeneous and portable parallel programming model (Poster). In: Int. Conf. for HPC, Networking, Storage and Analysis (SC'20). https://sc20.supercomputing.org/proceedings/tech_poster/poster_files/rpost145s2-file3.pdf
10. Nozal R, Bosque JL, Beivide R (2020) EngineCL: usability and performance in heterogeneous computing. Fut Gen Comput Syst 107:522–537

11. Parnassos I et al (2017) A programming model and runtime system for approximation-aware heterogeneous computing. In: 2017 27th International Conference on Field Programmable Logic and Applications (FPL). IEEE, pp 1–4
12. Alpay A, Heuveline V (2020) SYCL beyond OpenCL: the architecture, current state and future direction of hipSYCL. In: Proceedings of the International Workshop on OpenCL, IWOCL '20. NY, USA). Association for Computing Machinery, New York
13. Bosch J et al (2018) Application Acceleration on FPGAs with OmpSs@FPGA. In: 2018 Int. Conf. on Field-Programmable Tech. IEEE, pp 70–77
14. Wang C, Li X, Zhang J, Chen P, Feng X, Zhou X (2012) FPM: a flexible programming model for MPSoC on FPGA. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, pp 477–484
15. Ozkan MA et al (2020) AnyHLS: high-level synthesis with partial evaluation. https://arxiv.org/pdf/2002.05796.pdf. Accessed 12 Sept 2020
16. Lee S, Kim J, Vetter JS (2016) OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 544–554
17. Zohouri HR (2018) High performance computing with FPGAs and OpenCL. ArXiv preprint arXiv:1810.09773
18. Zohouri HR, Maruyama N, Smith A, Matsuda M, Matsuoka S (2016) Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, pp 409–420
19. Pouchet L-N, Grauer-Gray S (2015) PolyBench: the polyhedral benchmark suite (2011). http://www-roc.inria.fr/~pouchet/software/polybench
20. Zohouri HR et al (2020) Rodinia's Hotspot v7. https://github.com/fpga-opencl-benchmarks/rodinia_fpga/blob/master/opencl/hotspot/hotspot_kernel_v7.cl. Accessed 12 Sept 2020
21. McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 4:308–320
22. Halstead MH et al (1977) Elements of software science, vol 7. Elsevier, New York

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Gabriel Rodriguez-Canal[1] · Yuri Torres[2] · Francisco J. Andújar[2] · Arturo Gonzalez-Escribano[2]

Gabriel Rodriguez-Canal
grodriguez-canal@acm.org

Yuri Torres
yuri.torres@infor.uva.es

Francisco J. Andújar
fandujarm@infor.uva.es

[1] Bayes Centre, The University of Edinburgh, 47 Potterrow, Edinburgh, Scotland EH8 9BT, UK

[2] Departamento de Informática, Universidad de Valladolid, Escuela de Ingenieria Informatica, Campus Miguel Delibes s/n, 47011 Valladolid, Spain