Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Supporting efficient overlapping of host-device operations for heterogeneous programming with CtrlEvents

Yuri Torres, Francisco J. Andújar, Arturo Gonzalez-Escribano, Diego R. Llanos [*]

Departamento de Informática, Universidad de Valladolid, Spain

ARTICLE INFO

Article history:
Received 23 February 2022
Received in revised form 20 December 2022
Accepted 24 April 2023
Available online 5 May 2023

Keywords:
Parallel programming
Heterogeneous programming
Asynchronous operations
Events
GPUs

ABSTRACT

Heterogeneous systems with several kinds of devices, such as multi-core CPUs, GPUs, FPGAs, among others, are now commonplace. Exploiting all these devices with device-oriented programming models, such as CUDA or OpenCL, requires expertise and knowledge about the underlying hardware to tailor the application to each specific device, thus degrading performance portability. Higher-level proposals simplify the programming of these devices, but their current implementations do not have an efficient support to solve problems that include frequent bursts of computation and communication, or input/output operations. In this work we present CtrlEvents, a new heterogeneous runtime solution which automatically overlaps computation and communication whenever possible, simplifying and improving the efficiency of data-dependency analysis and the coordination of both device computations and host tasks that include generic I/O operations. Our solution outperforms other state-of-the-art implementations for most situations, presenting a good balance between portability, programmability and efficiency.

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Recent trends towards the end of Dennard scaling and Moore's law indicate that future computer systems will become more specialized and will comprise more complex architectures in terms of processors, accelerators, memory hierarchies, interconnect on-chip networks, storage, etc. This trend has been labeled as *extreme heterogeneity* [46]. Extreme heterogeneous systems contain more than one kind of device, such as multi-core CPUs, different types of GPUs, field programmable gate arrays (FPGAs) and/or digital signal processors. They have already been seen as the new standard platforms in the research communities of high-performance computing (HPC), integrated computing, and machine learning [37,23].

Heterogeneous computing enables programmers from different application domains to accelerate their applications by mapping calculations to specific devices. Exploiting all these devices with programming models, such as CUDA [29] or OpenCL [38], requires expertise in programming the underlying hardware, thus forcing the programmers to tailor their applications to specific devices, degrading performance portability [31].

As we will see in Sect. 2, different high-level heterogeneous programming frameworks that aim to hide details related to the management of several devices have been developed in recent years. However, these frameworks present some limitations. First, they do not always take into account the needs of some application types, such as streaming applications [13], where repetitive operations, both in host and device, should be overlapped efficiently and transparently with communications and synchronizations. Second, some of these programming frameworks either require the use of explicit synchronizations and communications primitives, or the use of specific mechanisms or programming languages and tools for different devices, thus increasing the complexity of applications development. Third, most of them do not include abstractions that allow a proper asynchronous coordination of device computations with generic, time-consuming host tasks, such as input/output operations to an external storage device. These limitations lead to unnecessary synchronizations and overheads.

In this work, we propose a new heterogeneous runtime solution, called CtrlEvents, which handles asynchronous communications between device computations and host tasks, as well as supporting different devices in a consistent and homogeneous way. We also provide a fully-functional prototype, called Controllers v2, that integrates CtrlEvents in the Controller heterogeneous programming framework [27,42].

[*] Corresponding author.
E-mail addresses: yuri.torres@infor.uva.es (Y. Torres), fandujarm@infor.uva.es (F.J. Andújar), arturo@infor.uva.es (A. Gonzalez-Escribano), diego@infor.uva.es (D.R. Llanos).

https://doi.org/10.1016/j.jpdc.2023.04.009
0743-7315/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Therefore, the main contributions of this work are the following:

- A runtime solution for heterogeneous programming frameworks, called CtrlEvents, which simplifies and improves the efficiency of data-dependency analysis, and the coordination of both device computations and host tasks that include generic operations, such as input/output.
- A fully-functional implementation of CtrlEvents to efficiently synchronize device computations, host-task operations, and data transfers in different kind of scenarios, from highly-loaded computations and data transfers, to very fast synchronization of low-loaded streaming operations. This implementation has been integrated into the Controller heterogeneous programming framework [27].
- An experimental study comparing the performance of CtrlEvents with other state-of-the-art heterogeneous programming frameworks, such as two different SYCL [21] implementations (hipSYCL [1] and Intel-LLVM for CUDA [9]), using NVIDIA and AMD GPU platforms. We use six case studies based on three applications inspired in well-known benchmarks: Rodinia Host-Spot [8], Polybench MM3 matrix multiplication sequence [33], and an implementation of a Sobel filter for image processing, applied to video sequences [18,10].

This work shows that the abstraction and techniques proposed in CtrlEvents lead to highly efficient heterogeneous programs, outperforming other state-of-the-art solutions in different workload scenarios. We also present a comparative of several development effort metrics for different versions of the programs mentioned above, using the programming frameworks considered in the experimental study. The joint use of the Controller heterogeneous programming framework and the CtrlEvents runtime presents a good balance between ease of programming and efficiency. The development effort measures are much lower than manually optimized programs with asynchronous communications, programmed in CUDA and OpenCL, with a reduction of between 20% and 75% in the number of programming-language tokens. Comparing with the same programs written in another main-trend solution (SYCL), the Controller codes sometimes present worse development effort measures (e.g. 5% to 17% more programming language tokens). However, in the case of a streaming application that applies an image filter to a video stream, the Controller program is much simpler (60% fewer programming tokens).

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 presents the Controller heterogeneous programming framework used as the starting point to develop our solution. Section 4 introduces the CtrlEvents proposal and describes how to implement it in the chosen programming framework. Section 5 shows the implementation of several study cases used as benchmarks, and describes the experimental study. Finally, Sect. 6 presents our conclusions and discusses future work.

2. Related work

Different approaches for heterogeneous parallel programming have been proposed in recent years. Many of them introduce abstractions that aim to overlap host-device communications and computation using synchronization mechanisms. In this section we review the different proposals so far, classifying them in terms of how they manage the data transfers and synchronization.

2.1. Proposals with explicit data transfers and synchronization

Low-level, hardware-oriented approaches The most intuitive approach to take advantage of the functionalities available in heterogeneous system is to manually design and program a tailored solution of our application, using either vendor-provided or native parallel programming models, such as CUDA [29], or OpenCL [38]. With this approach, it is possible to efficiently manage the hardware resources and configuration, but the programmer needs an in-depth knowledge of the target architectures and to manually manage the synchronization mechanisms. Besides this, the resulting application is unlikely to be easily portable to other hardware configurations.

Abstractions with low-level explicit data transfers Another approach is to introduce programming abstractions except for the management of data transfers, where low-level calls are still necessary. For example, OmpSs-2 [6] is a programming model composed of a set of directives and library routines that can be used in conjunction with a high-level programming language in order to develop concurrent applications. The control flow across tasks is implicitly derived from data-dependency analysis, as in OpenMP. Nevertheless, the data transfers with GPUs should be explicitly managed with CUDA or a similar low-level library. As we will show in Sect. 5, the use of generic task management mechanisms introduces overheads that can be noticeable when coordinating light-weighted host/device tasks with data transfers.

Higher level models with explicit data transfers and synchronizations There exist heterogeneous programming models with higher abstraction levels which provide portable mechanisms for communication and synchronization between host and devices. Some of them require the explicit invocation of these mechanisms by the programmer, including the management of asynchronous operations, streams, or similar concepts. For example, Kokkos [12,36] is a C++ programming model for writing performance-portable applications targeting all major HPC platforms. It is designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. However, the compilation process requires one type of target device that cannot be changed at runtime to be selected. Regarding data transfers, by design, the system never determines where or when a data transfer should be done in order to keep memory coherence across different devices. The programmer is the responsible of doing so, by explicitly invoking a specific function (deep-copy) to perform the data transfer. In Kokkos, only the use of mechanisms such as CUDA unified memory could avoid the explicit deep-copy calls, but it is a non-portable solution and introduces a performance penalty [39]. Moreover, when the deep-copy function is used to carry out a data transfer across different memory hierarchies, it always implies full *fence* operations. This means that the runtime system introduces a synchronization of the command queues before and after the data-transfer to keep memory consistency. [40]. Thus, the data transfer operations in Kokkos are always synchronous, and the programming model does not support overlapping data-transfers with computations in the same device.

HPX [20,11] is a parallel runtime system which extends the C++11 standard to facilitate distributed operations, to enable fine-grained, constraint-based parallelism, and to support runtime adaptive resource management. The data management in GPU devices should be done explicitly by the programmer using a lower-level programming model, such as CUDA. It includes explicit synchronization mechanisms, including a queue abstraction for data transfers. Other examples in this category include dCUDA [16], Groute [5], BlasX [48], G-Charm [44] or Executors [7].

2.2. Proposals with implicit data transfers and synchronization

Automatic scheduling of pattern loop iterations and other task-based approaches Some tools and libraries propose abstract approaches that are oriented towards the automatic, parallel execution of loops in heterogeneous devices. For example, Raja [4,22] is a C++ abstraction layer, developed at LLNL, that aims to enable performance portability. It targets loop-level parallelism for C++ applications, relying solely on standard C++11 language features for its external interface. Raja has internal extensions that use OpenMP, CUDA, and AMD HIP to manage heterogeneous devices. Other examples include LogFitc [47], or Maat [32]. There are also strategies for specific problems, such as the hybrid CPU/GPU approach described in [45] for iterative stencil computations, that introduce asynchronous communications and load-balancing across devices. There are also more generic task-oriented approaches with support for GPU accelerators that automatically derive dependencies to build a task-graph at runtime, using sophisticated graph-analysis techniques to schedule both the tasks and the data transfers needed [41]. The models in this category do no support the coordination of generic task graphs and data dependencies, such as those generated across different nested-loop sections.

Generic, high-level, heterogeneous programming approaches Several models use more generic programming abstractions, trying to achieve both code and performance portability in heterogeneous programming. Some of them are based on modern C++ features. For example, SYCL [21] is a standard model for cross-platform programming. Kernels are organized by a task graph that is implicitly constructed by the SYCL runtime. The control flow and the data communications can also be implicit. The SYCL ecosystem currently contains four major SYCL implementations: Codeplay's ComputeCpp [28], Intel's oneAPI [19], triSYCL [43], and hipSYCL [1] compilers. In general, the current implementations rely on different and non-compatible compiler back-ends for different types of devices. TriSYCL only supports CPUs and Xilinx FPGAs. HipSYCL supports CPUs and GPUs from different vendors, but not FPGAs. ComputeCpp supports CPUs and NVIDIA GPUs. Regarding oneAPI, it only supports the combination of CPU kernels with Intel GPUs and Intel FPGAs. There is a project to support NVIDIA devices, but using an alternative CUDA backend that takes advantage of the LLVM infrastructure [9]. Thus, all these implementations have limitations to operate with certain combinations of devices.

Another model in this category is dOCAL [34]. It presents a high-level abstraction API in C++ to simplify the implementation of distributed OpenCL/CUDA programs. It automatically manages and minimizes data transfers. dOCAL is compatible with existing OpenCL and CUDA libraries; can be connected with auto-tuning systems and can profile runtime behavior of OpenCL and CUDA programs; and it can leverage the usage of Unified Memory and pinned memory that can accelerate, hide or even avoid data transfers between devices' memories and the main memory.

All these models advocate the use of a single-code-for-multiple-devices approach. This includes the encapsulation of CPU tasks in kernels, in order to take advantage of parallel execution capabilities. Therefore, these kernels should not include code restricted to CPUs, such as I/O management, or calls to third-party libraries specific to a particular device, such as the ones used in video streaming applications. These are examples of scenarios where this kind of operations are hard to synchronize with other kernels and data transfers using an implicit data-dependency analysis.

As we will show in Sect. 5, solutions using implicit data-dependency analysis, in practice, forces the programmer to include additional explicit synchronizations to ensure correctness, thus leading to extra overheads. We will also show that, in some current implementations of these models, the overheads generated by the sophisticated mechanisms to manage the tasks or commands derived from the data-dependencies analysis are not negligible for repetitive low-loaded computations.

3. The Controller heterogeneous programming framework

The Controller heterogeneous programming framework [27,26,35] is used in this work as a starting point to design and implement our proposal. Controller is a heterogeneous programming model that enables performance portability across CPUs (using OpenMP), GPUs (using CUDA or OpenCL), FPGAs (using OpenCL) and Xeon Phi accelerators (using the Intel COI library). It is implemented as a library written in C99. Thus, it is compatible with any C99/C++ compiler and it is easily interoperable with other libraries and parallel programming models. It advocates a model which allows multiple explicit implementations per kernel, with specialized versions for each type of device if needed. Kernel implementations are compiled with their own vendor or native compiler. The programs are compiled with support for all the backends, linking everything together. Thus, the compiled programs can choose different target devices at runtime.

A preliminary support for host task and implicit asynchronous operations in the OpenCL backend for FPGA devices was presented in [35]. The results were promising in comparison with programs coded directly with OpenCL or with a SYCL implementation. Generalizing this support for any other backend in a portable and efficient manner requires the introduction of a new abstraction and the internal structure of the Controller backbone has to be redesigned. This issue will be addressed in Sect. 4. In this section, we summarize the features of the original Controller model, its programming abstractions, interfaces and features, together with an example of its capabilities from the programmer's point of view.

3.1. Controller's software architecture

The original Controller framework [27] proposes an abstract object, called *Controller*, to coordinate the activities of kernel execution and memory management on an accelerator or a set of CPU cores. Fig. 1 shows the elements of the original Controller model. A Controller object is associated to a particular instance of a device during its construction. Each object transparently manages the coordination and communication of the host code with that device.

The model provides a data type to encapsulate variables and data structures, such as multi-dimensional arrays. They can be allocated to one side (named internal variables), or in both host and device sides (tied variables). The programmer can push requests to the Controller object queue (see step (1) in Fig. 6). These requests can issue the execution of device kernels, or data movements for tied variables from host to device or device to host.

The execution policy module is in charge of processing the requests located in the queue. To do so, it first retrieves the following task, and determines which operations are needed to complete it, such as execution of computation, data movements, and/or synchronizations. After that, the execution policy module transfers the operations to the corresponding Controller backend using a generic API. Each backend executes the requested operations, taking advantage of the lower-level programming model of the corresponding device (step (2) in Fig. 6).

The parameters of each kernel request should be either data structures already tied to this Controller object, or input values of scalar types of the programming language used to implement the model. The kernel launching function also receives a parameter indicating the sizes of a multi-dimensional grid of fine-grain logical threads that the kernel will execute. The Controller creates the proper granularity, grouping them in blocks or in coarse-grain tasks, to adapt them to the specific device.

3.2. The tiling library used: Hitmap

Hitmap [15] is the portable library used in the Controller model to provide an abstract common interface for the data management

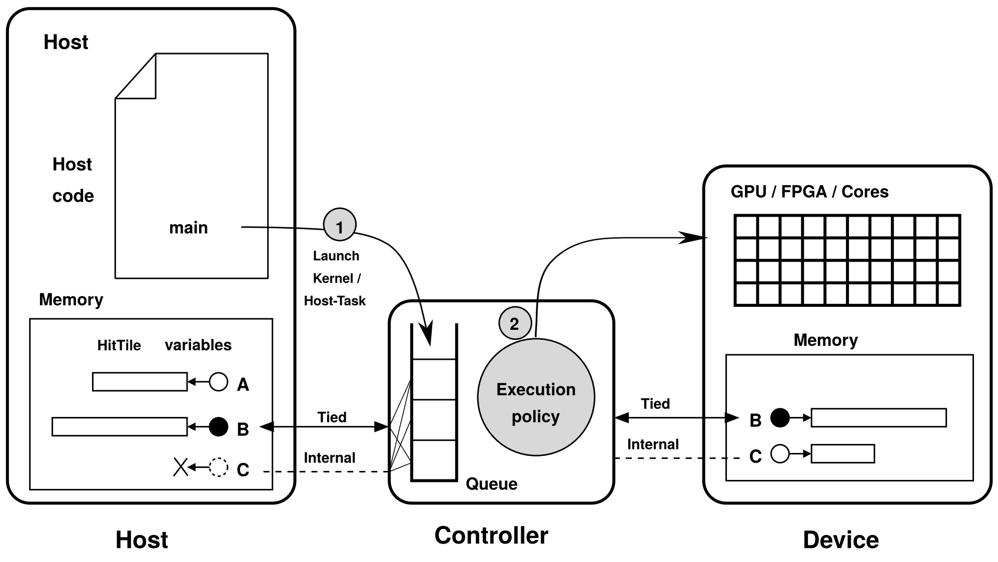


Fig. 1. The original Controller framework architecture [27].

in both the host code and the kernels executed in different types of devices. Data structures on the host are managed with HitTile objects, which are *fat-pointers* that store several meta-data, such as data sizes, and the data pointer. The Controller model extends the HitTile structure to store new meta-data related to the use of the tile in a device, including information to find the data in the device. The Hitmap's data access function, `hit()`, is used in the host or kernel codes to access the elements of a tile. It provides a portable view, with row-major order, on any device. More details on the use of Hitmap in Controllers can be found in [35]. An example of the joint use of Hitmap and Controller is described below. The reader already familiar with both frameworks may skip the following section and proceed to Sect. 4.

3.3. *Programmer's view: using the Controller framework*

To show the features of the Controller framework, we present a simple example that calculates a sequence of images representing a zoom of a chosen window of the fractal Mandelbrot set [24]. The code iteratively launches a kernel that calculates the following image, allowing an overlap of this calculation with the management of the preceding image in the CPU. In our example, this management consists of calculating how many points are under a given constant, in order to create a signature of the computation carried out, and calling a function that stores the figures, for example to later create an animation.

We now discuss the use of the Controller framework to implement this example. An important design decision of the Controller framework is to use macro functions in order to rewrite the code, allowing the use of C99-compatible compilers. In the Controller model, each kernel is declared using two macro function calls. The first is `CTRL_KERNEL_PROTO()`, which declares a common prototype for all the different implementations of a given kernel. The syntax of this function is the following:

CTRL_KERNEL_PROTO (*kernelName*, *implementationsList* , *parameterList*)

We now describe these parameters, using the code of Fig. 2 as an example of use. The first parameter is the name of a kernel, that can have several implementations for different backends or devices. In our example, line 1 shows the beginning of the kernel prototype for the kernel named `Mandelbrot`.

The second part of the invocation is a list of implementations for this kernel. This list starts with the number of different implementations available in this code, followed by two keywords for each implementation.

- The first keyword declares for which backend this implementation is designed, and which kind of lower-level code it contains. The Controller v2 library provides keywords that support these backends: `CUDA`, `OpenCL-GPU`, `OpenCL-FPGA`, and `CPU` for multi-threaded CPU tasks. There is also a `GENERIC` keyword, indicating that the implementation we provide is suitable for any backend.
- The second keyword allows a finer degree of control, allowing the declaration of a particular family of devices for a given backend. This information is used by the Controller to locate the best available implementation of a kernel at runtime for the chosen device (for example, "`CUDA, VOLTA`"). We also provide a `DEFAULT` keyword, to indicate that the code inside the kernel can be executed by any device of this backend.

In our example, line 2 declares that our code will only have one `GENERIC, DEFAULT` implementation. It is interesting to note that the different implementations of the same kernel can be a part of the same or different source code files, or can be precompiled in a different object file or library to be linked later.

Finally, the *parameterList* is the description of the kernel parameters, including their input/output roles, types and names. It starts with the declaration of the number of parameters, using three keywords. The first is its input/output role: `IN`, `OUT`, or `IO` (in/out) indicate the role of a HitTile object that is being referenced, while `INVAL` indicates a scalar value parameter. The second keyword is the type, and the third its name. In our example, lines 3-8 declare six parameters. The first is an output matrix whose values will be computed by the kernel, while the other five parameters are input values of native C types.

The `CTRL_HOST_TASK_PROTO()` is similar to the `CTRL_KERNEL_PROTO()` described above, but in this case describes sequential tasks intended to be executed by the CPU. Therefore, there is no need for keywords that indicate backend and/or devices. In our example, we declare a single host task called `Count_and_Store` (lines 11-14 in Fig. 2).

As stated above, the `CTRL_KERNEL_PROTO()` function allows the declaration of all the kernel implementations of the same kernel for different backends and/or device families that will appear later in the code. Each kernel implementation is declared by using

```
1   CTRL_KERNEL_PROTO( Mandelbrot,
2      1, GENERIC, DEFAULT,
3      6, OUT, HitTile_int, mat,
4         INVAL, int, threshold,
5         INVAL, float, x1,
6         INVAL, float, x2,
7         INVAL, float, y1,
8         INVAL, float, y2
9   );
10
11  CTRL_HOST_TASK_PROTO( Count_And_Store,
12     2, IN, HitTile_int, mat,
13        INVAL, int *, result
14  );
15
16  CTRL_KERNEL(Mandelbrot, GENERIC, DEFAULT, KHitTile_int mat
17     , int threshold, float x1, float x2, float y1, float y2,
18  {
19     float x0 = x1 + (x2-x1) / hit_tileDimCard(mat,0) * thread_id_x;
20     float y0 = y1 + (y2-y1) / hit_tileDimCard(mat,1) * thread_id_y;
21
22     float x = 0.0;
23     float y = 0.0;
24     int iteration = 0;
25     while ( x*x + y*y <= 2*2 && iteration < threshold ) {
26        int xtemp = x*x - y*y + x0;
27        y = 2*x*y + y0;
28        x = xtemp;
29        iteration++;
30     }
31     hit( mat, thread_id_x, thread_id_y ) = iteration;
32  });
33
34  CTRL_HOST_TASK(Count_And_Store, HitTile_int mat, int *result ) {
35     int count = 0;
36     for (int j = 0; j < hit_tileDimCard( mat, 0 ); j++)
37        for (int k = 0; k < hit_tileDimCard( mat, 1 ); k++)
38           if ( hit(mat, j, k) < 10 ) count++;
39     *result = count;
40     store( mat );
41  }
```

Fig. 2. Example of kernel and host-task prototype and their implementation for the Mandelbrot example using the Controller library.

a `CTRL_KERNEL()` macro function call. The syntax of this function is the following:

CTRL_KERNEL (*kernelName*, *backend*, *family*, [*paramType*, *paramName*]. . . , *code*)

The first three parameters are the kernel name and the tuple that describes one of the particular implementations declared in the corresponding `CTRL_KERNEL_PROTO()` function for this kernel name. These parameters are followed by the list of types and names of the kernel parameters enumerated in the `CTRL_KERNEL_PROTO()` function. Finally, the last parameter comprises the kernel code as a structured block (or as a string in the case of the OpenCL backends). In our example, the `CTRL_KERNEL()` invocation in lines 16-32 of Fig. 2 includes the code that computes the value of a single pixel of the Mandelbrot image. To determine the particular pixel to be computed, `thread_id_x` and `thread_id_y` are variables offered by the Controller framework that take different values for each logical thread in a grid defined by the user. This solution is a portable alternative to the `threadIdx.x` and `threadIdx.y` offered by CUDA, but maintaining the row-major layout across all devices supported by the Controller framework. Threads outside the user's grid, added in devices such as GPUs due to their block system, are internally skipped before the execution of the user code.

Regarding `hit_tileDimCard()` (lines 19-20), it is a Hitmap function that returns the cardinality of a given dimension for an input tile. The use of this function avoids the need to provide the dimensions of the tile as extra parameters.

The `CTRL_HOST_TASK()` macro function (lines 34-41) is similar to `CTRL_KERNEL()`, except that, in this case, the code to be executed is a structured block located just after the macro. In our example, the code traverses the image, counting the number of pixels below an arbitrary value (as a way to perform a signature of the resulting image), and calls a function that stores the image.

We now discuss the main host code (Fig. 3). It iteratively calculates snapshots of the Mandelbrot set, zooming into the image. As program arguments, the user provides the image size (lines 3-4), the coordinates of the initial window (lines 5-8), the zoom-in ratio (line 9), the number of images to be created (line 10), and the threshold as defined by the Mandelbrot set algorithm (line 12). The number of images is used to declare an integer array to store a signature of each generated image, just for testing purposes (line 11). The remaining program arguments are related to the Controller framework. Line 14 stores the ID of the device to be used in this execution.

The Controller framework allows us to choose between *synchronous* or *asynchronous* execution policies (line 15). This selects, at runtime, whether the Controller object executes kernel and data movements with or without computation and communication overlapping.

Lines 16-17 of Fig. 3 allow the programmer to choose in which NUMA node of the host platform the Controller object will be executed. This aims to reduce latencies with device communications. Although this could be done automatically, this mechanism allows us to experiment with different hardware configurations. For example, if the device is attached to a NUMA node while the disks used for the output are attached to a different NUMA node, the best choice is not straightforward.

Lines 19-20 define and initialize a structure that sets the logical threads space, in order to assign each pixel of the image that will be calculated to a different thread.

```
1  int main(int argc, char *argv[]) {
2
3     int size_x = atoi(argv[1]); // Image width (px)
4     int size_y = atoi(argv[2]); // Image height (px)
5     float x1 = atof(argv[3]); // initial X coord of starting window
6     float x2 = atof(argv[4]); // last X coord of starting window
7     float y1 = atof(argv[5]); // initial Y coord of starting window
8     float y2 = atof(argv[6]); // last Y coord of starting window
9     float zoom = atof(argv[7]); // Zoom ratio
10    int num_images = atoi(argv[8]); // Number of images generated
11    int *p_results = (int *)malloc(num_images * sizeof(int));
12    int threshold = atoi(argv[9]); // Mandelbrot threshold.
13
14    int device = atoi(argv[10]); // Device id
15    Ctrl_Policy policy = (Ctrl_Policy) atoi(argv[11]); // Sync or async
16    int host_aff = atoi(argv[12]); // NUMA node where affinity is set
17    Ctrl_SetHostAffinity(host_aff);
18
19    Ctrl_Thread threads;
20    Ctrl_ThreadInit(threads, size_x, size_y);
21
22    CTRL_BLOCK (1)
23    {
24       PCtrl ctrl = Ctrl_Create(CTRL_TYPE_CUDA, policy, device);
25
26       HitTile_int mat
27          = Ctrl_DomainAlloc(ctrl, int, hitShapeSize(size_x, size_y));
28       HitTile_int mat2
29          = Ctrl_DomainAlloc(ctrl, int, hitShapeSize(size_x, size_y));
30       HitTile_int tmp;
31
32       for (int i = 0; i < num_images; i++) {
33          // COMPUTE NEW IMAGE
34          Ctrl_Launch( ctrl, Mandelbrot, threads, CTRL_THREAD_NULL, mat
35             , threshold, x1, x2, y1, y2 );
36
37          // SWAP RESULT-COMPUTE MATRICES
38          tmp = mat2;
39          mat2 = mat;
40          mat = tmp;
41
42          // PROCESS RESULT IMAGE
43          int *result = &p_results[i];
44          Ctrl_HostTask( ctrl, Count_And_Store, mat2, result );
45          x1 += zoom * (x2 - x1);
46          x2 -= zoom * (x2 - x1);
47          y1 += zoom * (y2 - y1);
48          y2 -= zoom * (y2 - y1);
49       }
50
51       /* PRINT RESULTS */
52       printf("%d ", p_results[num_images-1]);
53
54       Ctrl_Free(ctrl, mat, mat2);
55       Ctrl_Destroy(ctrl);
56    }
57
58    free(p_results);
59    return 0;
60 }
```

Fig. 3. Main code implementation for the calculation of the Mandelbrot example using the Controller library.

The core of the program is a structured block preceded by the `CTRL_BLOCK` macro (line 22). This macro declares the number of Controller objects to be used inside the structured block that follows. Among other tasks, this macro creates OpenMP threads to manage the different controllers.

The structured block (lines 22-56) contains the program itself. It first creates the Controller object, selects the backend, the policy, and the device chosen above (line 24). Lines 26-29 declare and allocate two data tiles with the image size, using the `Ctrl_DomainAlloc()` function. Each invocation of this function allocates two buffers, one in the host and one in the device, that represent the data of the same tile. The Controller model transparently coordinates the execution of code and the data transfers between them in order to keep consistency, thus ensuring that the code executing in the GPU device or the host works with correct and updated data. Line 30 declares an auxiliary variable to perform swaps between these tiles, and lines 32-33 define the window ranges, which will be updated at the end of each iteration.

The main iteration of the program (lines 32-49) first launches the kernel to compute the image (lines 34-35). The `Ctrl_Launch()` function inserts the request of a new computing task into the Controller queue. This function receives the Controller object; the kernel name; the variable that contains the logical thread space; a parameter that allows, if needed, the group size automatically chosen by the framework (not used in this example) to be modified; and the actual parameters to be received by the chosen kernel.

The rest of the code allows the processing of the image just calculated with the launch of the calculation of the following image to be overlapped. To do so, the code exchanges the images using a double buffer technique (lines 38-40), and processes the newly calculated image using a host task (line 44). Lines 45-48 update the window size and position for the following iteration.

The call to the host task in line 44 inserts the request of a new host task into the Controller queue. In the default behavior, that we call "implicit mode", each time a new task is prepared for execution by the Controller scheduler, the scheduler first checks whether the input structures needed have already been retrieved from the corresponding device. If such is not the case, a previous request is inserted into the corresponding device queue to retrieve these values. The Controller programming model also offers a pair of `Ctrl_MoveTo()` and `Ctrl_MoveFrom()` functions that allows an experienced programmer to manually insert these requests into the Controller queue ("explicit mode"). In applications that launch different kernels and host tasks, these advanced functions allow some data movements to be prioritized, leading to a better performance. Explicit data-transfers are compatible with the implicit mode to force data transfers at specific places.

After the main loop, line 52 prints a signature of the last image; line 54 frees the memory previously allocated for the tiles in both host and device; and line 55 destroys the Controller object. Finally, after the structured block, other structures used by the host are freed, and the program finishes (lines 58-60).

After showing the main characteristics of the Controller framework programming, the following section introduces the CtrlEvents proposal, which simplifies and improves the efficiency of data-dependency analysis, and the coordination of both device computations and host tasks that include time-consuming, generic operations, such as input/output.

4. Managing devices synchronization: the CtrlEvents proposal

In this section, we present a proposal for a new abstraction and implementation of the system to execute asynchronous operations in the Controller model in a portable and efficient form. We first formally define the operations involved in the asynchronous programming model. Then, we discuss how to design a synchronization model that supports these operations. After that, we focus our analysis on how implicit communications can be automatically derived. Finally, the fourth subsection introduces CtrlEvents, a proposal that offers a portable and efficient way to use all the features described, across different backends.

4.1. Asynchronous model operations

In this section, we first formalize the asynchronous programming model operations, in order to simplify the discussion of the proposal and its implementation. The transactions between the host and the device can be described as a *Sequence of Operation Requests* ($R = \{r_0, \ldots, r_n\}$) issued by the coordination code executed in the host. An operation request r_i should be one of the following[1]:

- **Allocate:** *Alloc(x, l)*. A request to allocate a data structure in the host and/or device memory. The parameter x represents a data structure and l represents the location where memory is allocated (*Host, Device,* or *Both*).
- **Deallocate:** *Free(x)*. A request to deallocate the memory image/s of a data structure x.
- **Host-to-Device:** *HTD(x)*. A request to transfer the values of the data structure x from the host memory space to the device memory space. Memory in both sides should have been allocated first.
- **Device-to-Host:** *DTH(x)*. A request to transfer the data values of x from the device memory space to the host memory space.
- **Kernel launching:** *KL(f, In, Out)*. A request to execute a kernel in the device, as soon as its parameters are available. It receives a function name f and two sets of data structures as parameters. The set *In* indicates the references to data structures that are inputs. The set *Out* indicates the references to data structures that are outputs. The same data structure may appear in both the input and output sets of references, indicating that the content is both read and written during the kernel execution.
- **Host-task launching:** *HL(g, In, Out)*. A request to execute a function g in the host, as soon as its parameters are available. It has the same format as the kernel-launching requests, including the input and output sets.
  Kernels and host-tasks can also have a set V of parameters received as values: We omit this in their formal definitions because these parameters are not related to dependencies or to the problem of data transfers.
- **Wait:** *Wait(x)*. Blocks the execution of the host coordination code until all requests involving the data structure x have finished. In many programming models, such as CUDA, this operation is implicit by default after *DTH(x)* blocking requests, although it can be relaxed by the programmer using explicit asynchronous transfers. In our asynchronous model, this operation is used only when the coordination code needs to use values retrieved from the device. For example, this operation would be necessary in a loop with a convergence condition that is calculated with a reduction operation in the device.

[1] Unless otherwise indicated, these are non-blocking execution requests.

4.2. Designing a synchronization model

The original Controller execution policy module was synchronous, thus executing each operation only after the previous requests have finished. In order to choose a synchronous or asynchronous policy at runtime, not only when creating the Controller but even during the execution, we design a new execution policy module for asynchronous operations and we adapt the original synchronous module to use the same internal mechanisms. Both modules are fully compatible and the policy can be changed at any time. Each operation request is processed synchronously or asynchronously with respect to previous requests, depending on the active policy when the request was launched.

In our proposal, the order of the kernel's execution is strictly preserved. The decision of which kernels should be or not executed concurrently, in order to improve the overall performance of the application, is a problem that is orthogonal to the overlapping of data transfers with kernel or host-task execution, and it is beyond the scope of this work. Host-tasks execution requests are also executed in order. In the asynchronous model, host tasks and kernels can be overlapped if dependencies allow it (preserving their partial orders), and data transfers can be overlapped with both kernels and host-tasks execution.

The internal rules that decide when a request can be safely overlapped with others are designed by studying the dependencies between the different types of requests, and taking into account the input/output role of their parameters. Each data structure can have at most two memory images, one in the host and one in the device. The kernel, host task, and data transfer requests between them are analyzed, considering them as a multiple-reader/multiple-writer problem. The dependencies generated by this scheme are depicted in Fig. 4. The rules for asynchronous execution associated to each operation follow:

1. *Alloc(x,l)*: All subsequent requests involving the use of the data structure x should wait until the allocation of x has finished.

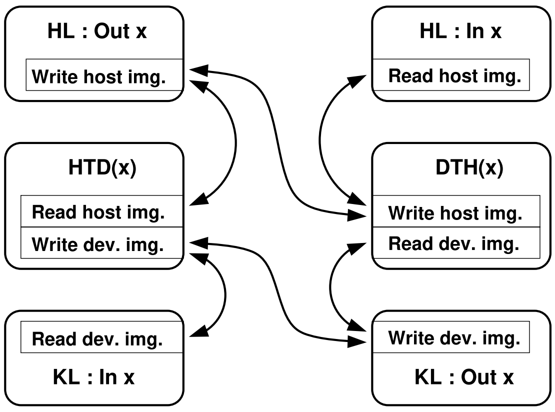


Fig. 4. Dependencies between request types. Rounded boxes identify request types, using the *x* data structure as parameter. We distinguish between *KL* or *HL* requests that use *x* with an input or output role. For clarity, the *Wait* and *Alloc/Free* operations are skipped in the figure. Inside each box, we represent in small boxes whether the request reads or writes in the host or device memory image of *x*. Arrows express dependencies between request types implied by the use (read or write) of the memory images of *x*. Requests that are not linked with arrows can be executed concurrently. Remember that this model does not consider the case of several concurrent kernels or several concurrent host tasks.

2. *Free(x)*: A request for deallocating *x* should wait until all the previous requests involving *x* have finished.
3. *HTD(x)*: Before executing this request, wait for pending operations on *x* if: (a) *x* is a parameter of a non-finished *DTH* operation; (b) *x* appears as a parameter (input or output) of any previous non-finished kernel (*KL* operation); or (c) *x* appears as an output parameter of a non-finished host-task execution (*HL* operation).
4. *DTH(x)*: Before executing this request, wait for pending operations on *x* if: (a) *x* is a parameter of a non-finished *HTD* operation; (b) *x* appears as an output parameter of any previous non-finished kernel (*KL* operation); or (c) *x* appears as an input or output parameter of a non-finished host-task execution (*HL* operation).
5. *KL(f,I,O)*: $x \in I$, $y \in O$: Before executing this request, wait for pending operations on *x* if *x* is a parameter of a non-finished *HTD* operation; before executing this request, wait for pending operations on *y* if *y* is a parameter of a non-finished *HTD* or *DTH* operation. Wait also for any previous non-finished *KL* operation.
6. *HL(g,I,O)*: $x \in I$, $y \in O$: Before executing this request, wait for pending operations on *x* if *x* is a parameter of a non-finished *DTH* operation; before executing this request, wait for pending operations on *y* if *y* is a parameter of a non-finished *HTD* or *DTH* operation. Wait also for any previous non-finished *HL* operation.
7. *Wait(x)*: Wait until all previous operations with *x* as parameter are finished.

4.3. Differences between synchronization models: an example

To better show the differences between synchronous and asynchronous execution models, suppose that a and b are tiles that have been just initialized in the host. After that, the host coordination code launches the following sequence of operations:

1. $K1 = KL(I = \{a\}, O = \{a\})$
2. $H1 = HL(I = \{a\}, O = \{a\})$
3. $K2 = KL(I = \{b\}, O = \{b\})$
4. $H2 = HL(I = \{b\}, O = \{b\})$
5. $W1 = Wait(a)$
6. $W2 = Wait(b)$

Fig. 5 represents the sequence of operations that are carried out in synchronous and asynchronous mode. In the figure, HT is the Host Task execution thread, DHT is the Device-to-Host stream/queue, HTD is the Host-to-Device stream/queue, and K is the Kernel execution stream/queue. Solid arrows represent the synchronizations that are implemented with events in both policies. The dashed ar-

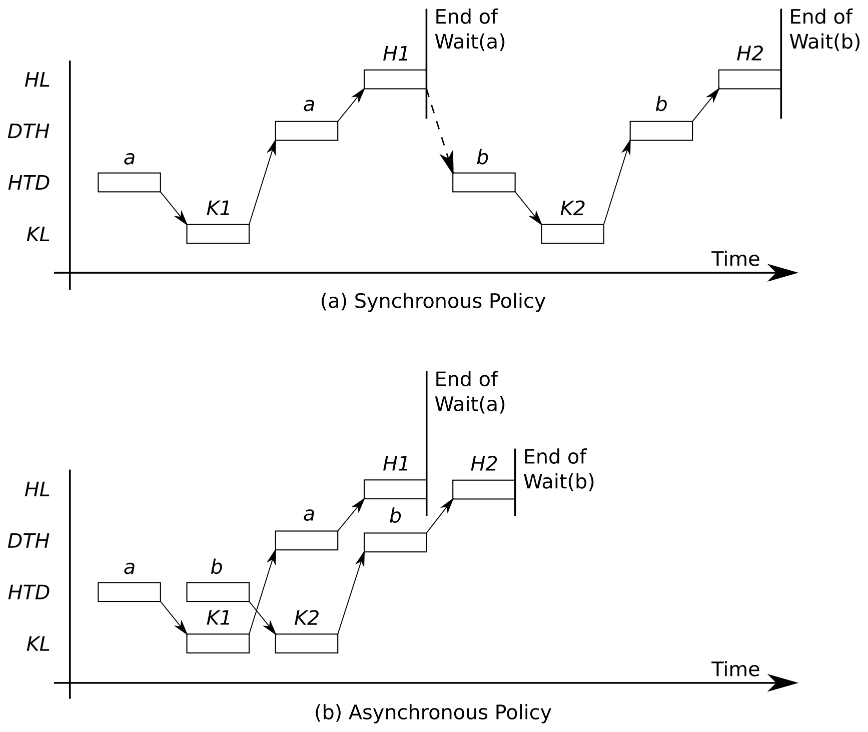


Fig. 5. Execution of a set of operations in synchronous (top) and asynchronous (bottom) execution models.

Table 1
Rules to automatically insert data transfer operations. The rule triggered depends on the request type (*KL* or *HL*); the input/output role of the parameter tile *x*; and the previous status of the tile. A warning is issued when a computation that uses *x* as input is invoked before the data has been previously initialized.

Operation	$s_h(x)$	$s_d(x)$	Actions
$KL : x \in I$	0	0	Warning
	0	1	-
	1	0	$s_d(x) = 1$; issue $HTD(x)$
	1	1	-
$KL : x \in O$	0	0	$s_d(x) = 1$
	0	1	-
	1	0	$s_h(x) = 0$; $s_d(x) = 1$; issue $HTD(x)$
	1	1	$s_h(x) = 0$
$HL : x \in I$	0	0	Warning
	0	1	$s_h(x) = 1$; issue $DTH(x)$
	1	0	-
	1	1	-
$HL : x \in O$	0	0	$s_h(x) = 1$
	0	1	$s_h(x) = 1, s_d(x) = 0$; issue $DTH(x)$
	1	0	-
	1	1	$s_d(x) = 0$

row represents the extra synchronization that only appears in the synchronous model.

4.4. Deriving implicit communications

An important design decision for maximum efficiency is the choice of the dependency-checking and memory-consistency preserving mechanisms. We introduce in the original HitTile structures (described in Sect. 3.2) new meta-data to record the status of events and the memory consistency information. Thus, the status of a HitTile can be directly checked and modified by the library functions at runtime.

The need to issue *HTD* or *DTH* operations to keep the memory consistency between host and device images can be detected when the operations in the queue are evaluated. While a kernel or host task is evaluated for dependencies, the meta-data in the tiles are updated. These updates express the future state of the parameters regarding memory consistency and synchronization events when the operation will be finally executed, after all the waiting conditions are met. Thus, the next time a tile is used, during a request evaluation, whether a data transfer is needed to ensure memory consistency can be derived.

This process is carried out as follows. Let x be a HitTile structure. Let $s_h(x) \to \{0, 1\}$ and $s_d(x) \to \{0, 1\}$ indicate the state of the host and device memory images of x respectively, where 0 represents a *non-updated* state, and 1 represents an *updated* state. When allocating images, the initial allocation operation $Alloc(x, l)$ sets both bits to 0. In practice, it indicates that none of the images are valid. During the evaluation phase of the request, if the request uses the tile as output parameter, it means that either the host or the device will write its image. In this case, the corresponding bit is set to 1. If the counterpart bit was also 1, it is cleared (indicating that the copy of the counterpart will not be updated), and a data transfer operation is inserted into the device queues to guarantee consistency in order to avoid false sharing issues. If the request being evaluated states that the tile will be used as input by one side, and its bit indicates that the copy will not be updated at that time, a data transfer operation is inserted into the device queues before the operation being evaluated, and its bit is set. Finally, if the tile is used as input/output, the rules for input and for output are applied in this order.

Table 1 summarizes the rules to detect and request a data transfer operation, and to update the state indicators in the HitTile, when a kernel or host task is evaluated. When these rules are applied, explicit *DTH* or *HTD* requests are no longer needed in the coordination host code.

Since operations are non-blocking, the queue evaluation process can easily advance faster than the computations. Thus, in practice, the data transfer operations are submitted to the device queues earlier than the beginning of the execution of the kernel or host-task execution that requires them. In any case, correctness is ensured.

Explicit data transfers can also be included in the code if the programmer finds opportunities to prefetch data on the device or host before the coordination code inserts the corresponding tasks into the queue. These explicit operations update the state indicators directly, maintaining consistency with implicitly-derived transfers.

4.5. The CtrlEvents proposal

Having formally defined the operations involved in the asynchronous programming model; discussed how a synchronization model that supports these operations can be designed; and having described how implicit communications can be automatically derived, we now introduce CtrlEvents, a proposal that offers a portable way to use all the features described across different backends in an efficient way.

In [35] we introduce a Controller backend specifically designed for FPGA devices using OpenCL. In this work, we generalize that previous proposal to make it portable and interoperable for different backends with different technologies. This work presents results for the CUDA and OpenCL backends for NVIDIA and AMD GPU devices. The current Controller version also includes an OpenMP backend for multi-core CPUs, an OpenCL backend for Intel FPGAs, and a Xilinx Vitis backend for Xilinx FPGAs, using the same abstractions.

CtrlEvents and events adapter CtrlEvents is an abstraction to represent event objects used for synchronization in different technologies, such as CUDA, OpenCL, OpenMP, etc. It is conceptually an abstract class, with a different implementation for each technology. In C language, we implement its fields with a union structure, with specialized fields to create synchronization points and triggers for different operations using the features offered by the target technology.

Fig. 6 shows the process in detail. A Controller instance is associated at construction time with a specific device and with a technology to manage it. When an operation is requested (1), the task is inserted into the queue. Then, the new execution policy module calls the API of the Events Adapter (2) to interact with the events information stored in the HitTiles used as parameters, and to transform the requests to the proper technology associated with the Controller instance. Once the operations are enqueued/launched into the device queues, the corresponding drivers are responsible for coordinating the execution of the kernels and data transfers (3), following the schedule implicit on the event triggers programmed by the execution policy and the events adapter.

The Events Adapter currently supports OpenCL for GPUs and FPGAs, CUDA for NVIDIA GPUs, and sets of cores with OpenMP for parallel computing with CPUs. In the case of OpenCL, it enqueues operations and event triggers in the OpenCL driver queues. In the case of CUDA, operations are launched in CUDA streams, using CUDA events to synchronize them. In the case of OpenMP, the kernel executions are implemented as OpenMP tasks, allowing the programmer to choose the particular cores to be used, thanks to an affinity control system provided by our library. Using OpenMP locks, we have both implemented synchronizations for the event abstraction, and a queue that is similar to OpenCL queues or CUDA

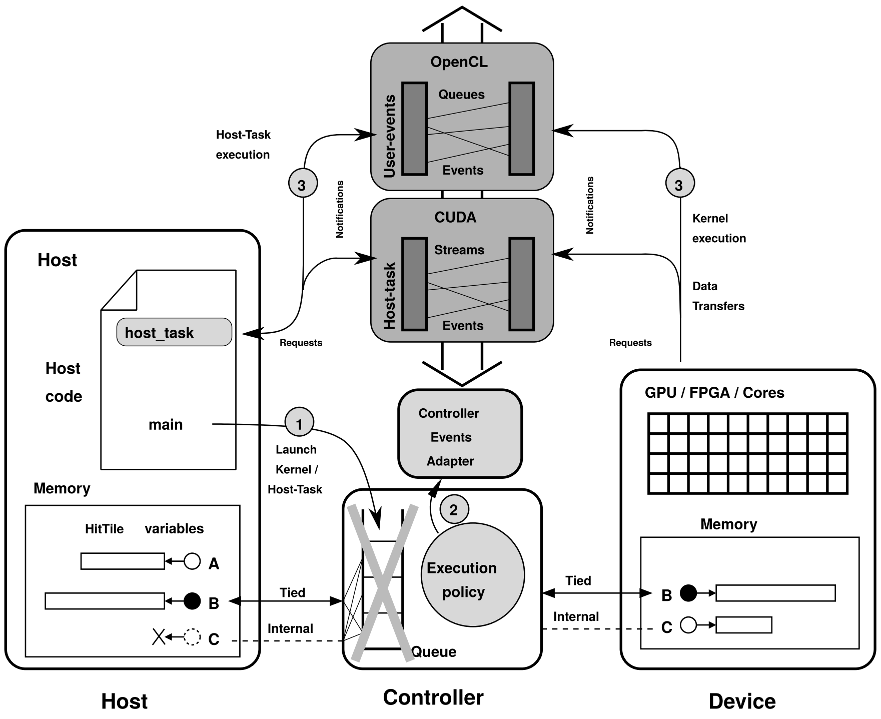


Fig. 6. The Controller model with CtrlEvents support: White boxes indicate the elements of the original model. Light gray elements are re-designed or introduced to support the CtrlEvents proposal. Dark gray shaded boxes indicate the operations and activities requested to the drivers of lower-level programming models associated with different Controller backends.

streams. As a result, all device types are interoperable using the same abstractions.

Host-task synchronization Execution and synchronization with different technologies is more complex for the host task than for kernels. In all cases, we use a different queue or stream (depending on the lower-level technology) to enqueue host-task operations. We have found that the most efficient and versatile mechanism to implement the synchronization with the host task is different for each low-level technology. To offer different, compatible implementations for our host tasks, the host-task declaration creates a wrapper that calls the function containing the actual user code. The wrapper is launched as an OpenMP task each time the host task is invoked, and implements a different synchronization mechanism that can be chosen by the events adapter, depending on the particular backend used. In OpenCL, the wrapper is synchronized with operations in other queues using the event-wait functionality and user-defined events. In CUDA, the wrapper function is synchronized with other stream operations using the CUDA interface to call host functions. Finally, in OpenMP, the host task is simply synchronized with the native OpenMP task synchronization mechanism. Thus, the events adapter and the host-task wrapper enclose all the technology dependent details.

Elimination of the Controller internal queue With the proposed solution, the execution policy takes the decisions, and the events adapter introduces all the synchronization control needed in the lower-level technology. The execution control is now carried out by the lower-level drivers in the most efficient way. Thus, there is no need of internal queues in the Controller backends. In particular, the main queue to store and process operation requests is no longer needed and can be eliminated. Instead, the requests can be directly processed by the execution policy and the Events Adapter, returning the control to the host code as soon as the events and operations are enqueued in the lower-level technology driver (see the discarded queue in Fig. 6). This leads to an additional slight improvement in the efficiency of processing the operation requests, avoiding the small overheads associated with the synchronizations with the queue control, originally executed by a different thread that is no longer needed.

5. Experimentation

In this section, we describe an experimental study to evaluate the potential advantages and constraints of the asynchronous execution model implemented with CtrlEvents in the Controllers v2 library, comparing with other state-of-the-art programming models. We show how the proposed implementation of CtrlEvents performs efficiently in different scenarios of computation load, and balance between data transfers and host/device computations. Both synchronous and asynchronous versions of programs are studied to show the overheads introduced and the level of asynchronicity exploitation achieved by the different programming models considered.

The section includes: (1) A description of the case study applications considered; (2) a description of the experimental environment and setup; (3) a performance study of our proposal; and (4) a development effort comparison between using the Controller model with CtrlEvents and using CUDA, OpenCL, SYCL or dOCAL directly.

The Controllers v2 library with CtrlEvents, all codes used in the study, and the whole set of experimental data are freely available at the Controllers repository [42].

5.1. Case studies

To test our approach and implementation, we select three different applications, one with four variants, to obtain a total of six case studies. They are representative of the different situations that may arise, including scenarios with memory, compute, communication, and input-output bound situations. There exist other, well-known set of applications that could be used for these purpose, such as the NAS Parallel Benchmark (NPB) [3]. There are indeed several NPB implementations for GPU devices, e.g. [2]. All of them implement the original algorithm specifications, executing the computation in a single GPU device. All the NPB programs, both kernels and pseudo-applications, have an initialization stage, a computation stage (a single kernel or a sequence of kernel executions), and a last stage to move the results to the host in order to check the correctness. Thus, there are no kernel execution and data transfer interleaving, and no possibilities to exploit asynchronous data transfers for overlapping them with computation. Instead, the applications chosen for this paper are based on real GPU benchmarks or applications, presenting potential overlapping of data-transfer and computation using asynchronous operations. This feature is common in streaming applications.

The first one is an iterative stencil program, a well-know example of memory-bound kernel. On each iteration, the result matrix is transferred to the host to be stored there, in order to generate an animation of the simulation evolution. The computational load of the device kernel is smaller than the data transfers between device and host. Thus, it can become an example of data-transfer bound application. Nevertheless, the higher cost of the memory-copy operations to save the results in a host-task makes the whole application a memory-bound case. This application is appropriate to test the effects of overlapping host-task and costly data-transfers with small computational loads in the device.

The second one is based on iterative matrix multiplications on the device, with a non-trivial post-processing of the results in a host task. In this case, the computational cost grows with the input size faster than the cost of the data transfers. For very small sizes (less than 512×512 elements) it is a data-transfer-bound application. Nevertheless, the execution times are so small for those input sizes that it is faster to execute the whole application in the CPU. Thus, for real useful input sizes it becomes a compute-bound application in the device. This application is appropriate to test the effects of overlapping high device-computational loads with data-transfers and host-task that becomes relatively lower and lower with the input size.

The last application applies a image filter on the frames of a video stream. We consider variants combining both files and memory as input/output channels, to represent the behavior of the filter at different stages of an image processing pipeline. The different versions explore from input/output bound to memory-bound scenarios. These programs feature very fast kernels with lower load than the communications, bidirectional data transfers, and host tasks with different execution costs, presenting a demanding scenario to exploit asynchronicity for concurrency exploitation.

We now describe the details of each application considered.

Hotspot Rodinia's Hotspot stencil computation. The base program for this case study is included in the Rodinia benchmarks suite [8]. It computes the stability point of Poisson's Partial Differential Equation (PDE) for heat diffusion. It uses a Jacobi iterative method on a 2-dimensional discrete space. It is a 4-point stencil program that executes a fixed number of time iterations. The kernel exploits the shared-memory for both spatial and temporal locality. We test the program with 300 iterations, and a *height* parameter of 4, meaning that each launched kernel executes 4 matrix update iterations using halos of width 4. In order to integrate it in a production stencil framework that creates an animation of the computation evolution, and to check partial results, we add a transfer of the result matrix to the host after each kernel launching operation, saving it in a different host buffer with a host task. This leads to opportunities for communication and computation overlapping.

Matrix Pow This program is an evolution of the *2mm* and *3mm* programs in the PolyBench Benchmarks [33], to generate a chain of matrix multiplications of arbitrary length. It computes the normalization of the matrices $C_i = A^i : i \in [1 : n]$. In the device, it iteratively computes the multiplication of the original matrix by the partial result of the previous step: $C^k = C^{k-1} \times A : k \in [1 : n]$ where $C^0 = A$. The kernel to multiply matrices is obtained from the CUDA Toolkit samples [30], and it has also been ported to OpenCL. This optimized kernel uses shared memory to take advantage of the GPU 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 matrix normalization consists of the following phases: (a) Determining the minimum and maximum values in the matrix; (b) subtracting the minimum from each element of the matrix, and dividing each element by the maximum; (c) computing the *elements norm* as the square root of the sum of each element to the power of two; and (d) dividing each matrix element by the elements norm.

Sobel The Sobel Operator [14] processes gray-scale images to detect edges. It applies two stencil operators to the input image, to obtain the derivatives in the X and Y directions. The gradient magnitude is computed on each cell as the Euclidean distance of the corresponding cells on the matrices obtained as the output of the filters. For our experimental study, we choose an implementation that iteratively processes frames from a video in YUV format [10]. It reads an input video stream from a file, frame by frame. Each frame has three components that are sent to the device. The kernel of this application contains all the operations of the Sober filter, and the kernel is applied to each component separately. The resulting image is transferred back to the host to store it in an output video file. Each component of a frame is read, written, computed, or transferred separately. To simulate different scenarios of the application of the Sobel filter, alone or as part of a parallel image processing pipeline, we consider four scenarios: (a) The input/output images are read from and written to files (this is the original program, that we name "File to File"), an I/O-bounded situation; (b) the input images are read from a file, but the output images are stored in memory, to simulate the starting stage of a pipeline (that we name "File to Memory"); (c) the input images are read from a memory buffer, but the output images are written to a file, to simulate the last stage of a pipeline (that we name "Memory to File"); and (d) the input/output images are retrieved and stored in memory buffers, to simulate a stage in the middle of a pipeline (that we name "Memory to Memory"), a memory-bounded situation. The computation of the Sobel filter is a very fast operation on a GPU. Thus, this case study program is very demanding in terms of concurrency exploitation and asynchronous data-transfer executions. Each case presents different memory vs. computation or host tasks vs. kernels load balance, and introduces a different cost variability in the host tasks. The File to File case represents the extreme situation of higher load in the host-tasks (I/O bounded) and the Memory to Memory case represents the extreme situation of lower load in the host-task (memory-bounded). Thus, they are the two most interesting cases.

5.2. Experimental environment and setup

The experiments have been conducted in a server provided with GPU accelerators of different vendors. It features two Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz, an NVIDIA Tesla V100 PCIe 32GB GPU, and an AMD Vega 10 XT Radeon PRO WX 9100 GPU. The server runs a CentOS 7 operating system and the jobs are launched from a frontend using Slurm.

The compiling and execution tools and libraries include GCC 10.3, LLVM 12.0.0, hipSYCL 0.9.1, LLVM-INTEL 12.0.0 (the Intel backend for SYCL on top of CUDA), CUDA 11.2, and ROCm 4.1.0 for the hipSYCL support of AMD GPUs. The OpenCL libraries included

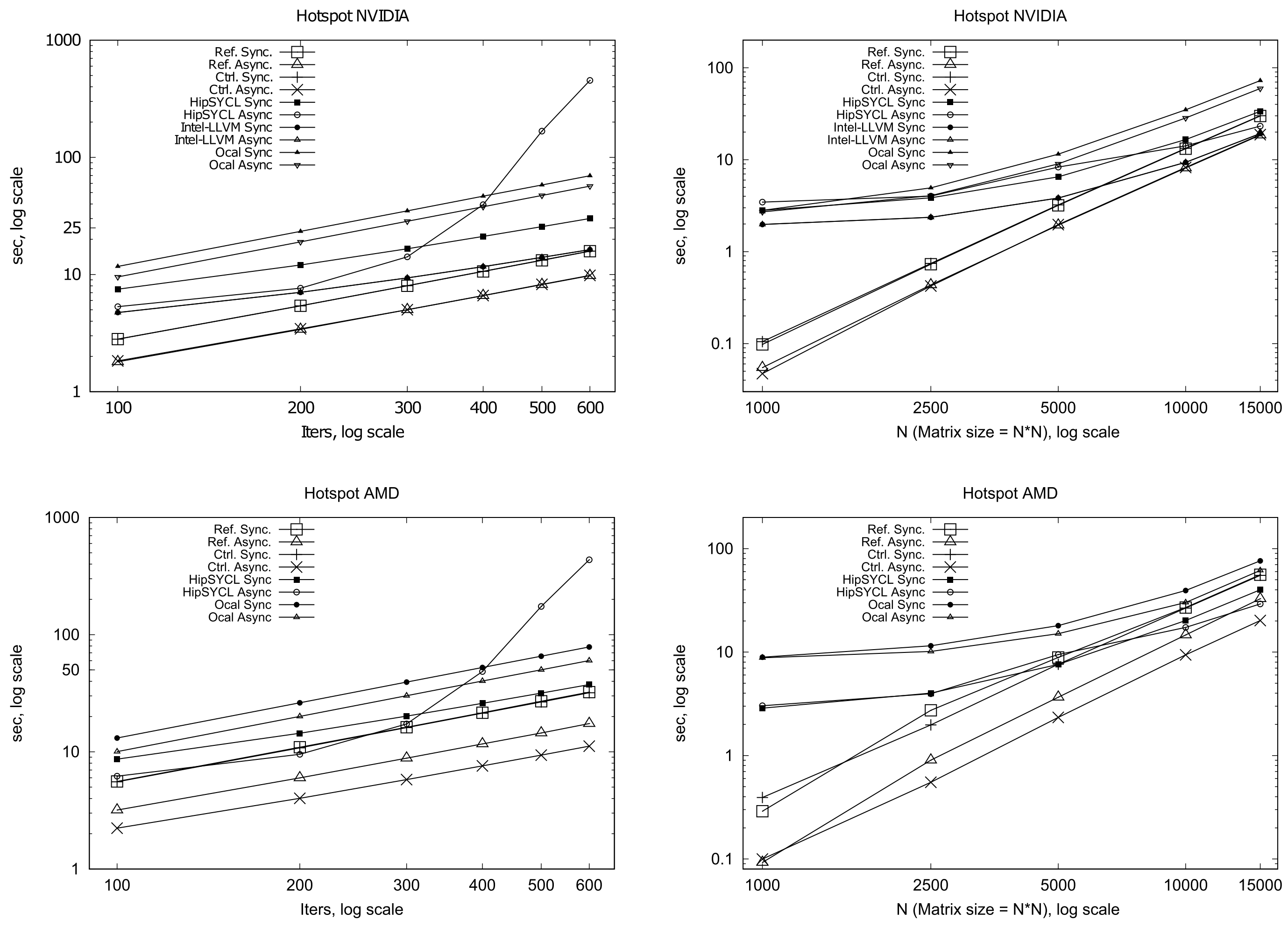


Fig. 7. Experimental results for the Hotspot case study, using CUDA (“NVIDIA” plots) and OpenCL (“AMD” plots).

in the corresponding CUDA and ROCm releases are used for the OpenCL experiments.

Programs for each case study are developed in CUDA, OpenCL, SYCL, dOCAL and Controller. We develop two versions of the reference CUDA programs, and two versions of the reference OpenCL programs: A simple synchronous one (*Ref. Sync*), and a manually optimized version with the best possible combination of asynchronous operations (*Ref. Async*). The *Sync* version is in fact equivalent to the original Controller model (Sect. 3), but with the Controller internal queue removed as described at the end of Sect. 4.5. Therefore, the performance figures shown by this *Sync* version are slightly better than those that can be obtained with the original Controller proposal [27].

The SYCL codes are compiled first with hipSYCL, using the CUDA backend for NVIDIA’s GPUs, and a different compilation using the ROCm backend for AMD GPUs. The SYCL codes are also compiled for NVIDIA with the Intel LLVM-CUDA backend. Synchronous versions of the SYCL programs are also developed, introducing explicit synchronization code to also test the internal implementation mechanisms for this kind of operations. Regarding dOCAL, we have run the experiments with the use of pinned buffers for better performance. We have used OpenCL to create an additional kernel to allow task synchronization.

We describe now the issues encountered. Regarding SYCL codes, file operations inside kernels to be executed by a CPU generate compile-time or runtime errors. We solve this issue by placing file operations in the coordination host code. This, in turn, forces us to add additional synchronization operations, because the automatic system does not take into account dependencies between kernels and operations in the host coordination code. Regarding the Intel-LLVM compiler, trying to launch CPU kernels and GPU kernels in the same program leads to a runtime error, since it only supports one type of device at a time. This forces us to remove CPU kernels when using this compiler, moving the tasks assigned to them to the host coordination code, thus needing additional synchronization operations. This affects the Matrix-Pow code, which needs to execute matrix normalization in a host task.

The Controller programs are compiled with OpenCL or CUDA specialized kernels. The executable names are *hipSYCL Sync, hipSYCL Async, Intel-LLVM Sync and Intel-LLVM Async*. The Controller code can select the synchronous or asynchronous execution policy at runtime. Thus, the code is the same in both cases. In the following discussion, we use the names *Ctrl Sync* and *Ctrl Async* to refer to the executions using the synchronous or asynchronous policies, respectively.

For Hotspot and Matrix-Pow, we conduct two types of experiments: *Iterations*, and *Sizes*. In *Iterations*, for a fixed input size, we modify the number of iterations, to test the effect of accumulated overheads and the queue management when the host code submits many tasks. In *Sizes*, for a fixed number of iterations, we modify the input sizes, to test the effect of different workloads and the balance between computation and data-movement times. For the Hotspot program, we measure 300 iterations of the program using matrices with input sizes from $1\,000 \times 1\,000$ to $15\,000 \times 15\,000$, and also a fixed input size of $2\,500 \times 2\,500$, with iterations from 100 to 600. For Matrix-Pow, we test the program using iterations

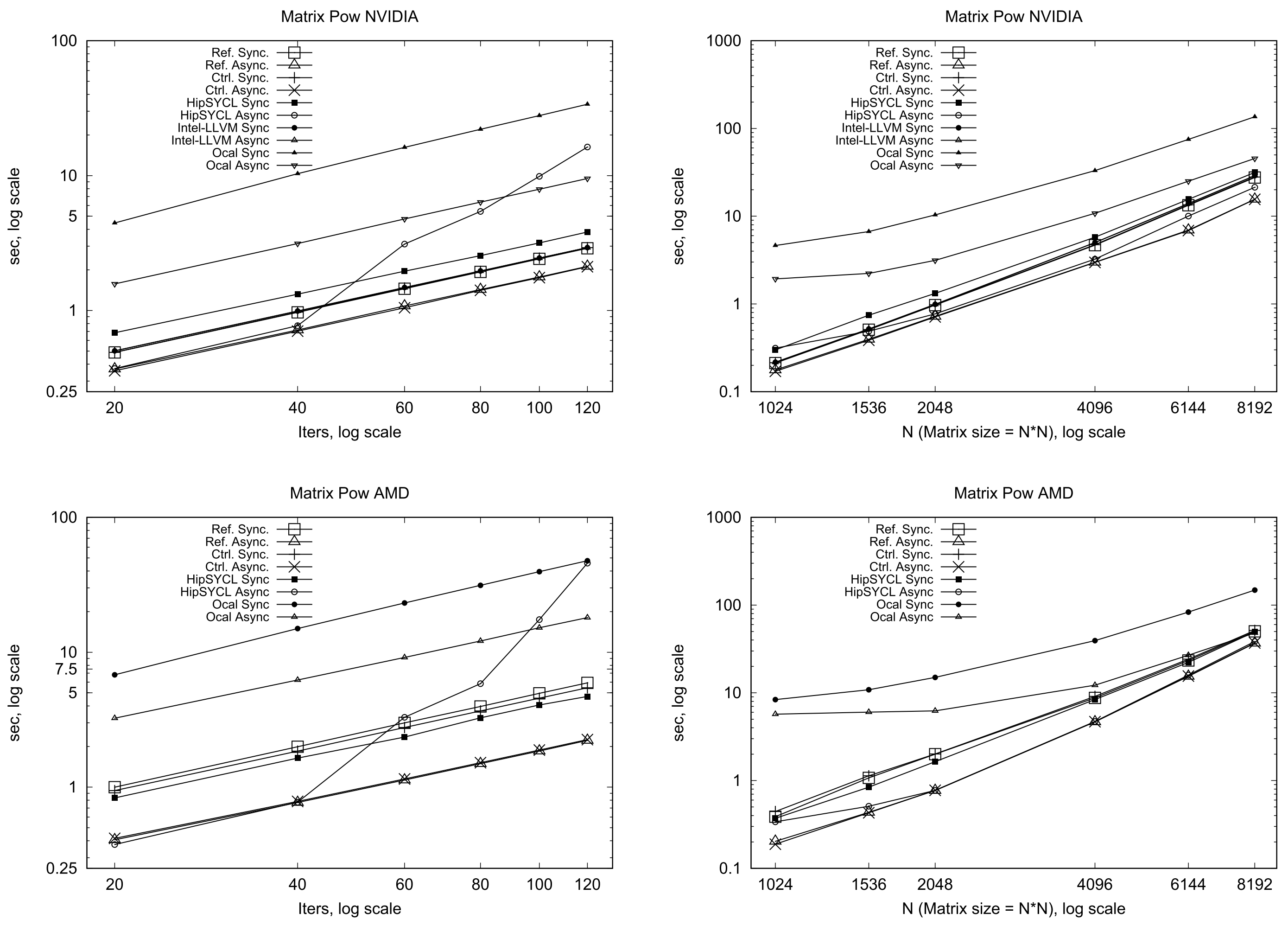


Fig. 8. Experimental results for the Matrix Pow case study, using CUDA ("NVIDIA" plots) and OpenCL ("AMD" plots).

from 20 to 120, with fixed input size of 2 048 × 2 048, and also with input sizes from 1 024 × 1 024 to 8 192 × 8 192 and a fixed number of 40 iterations. In the Sobel programs, the input size is fixed due to the specific size of the video frames. Thus, only the effect of the number of iterations/frames can be explored. The program is tested with 20 to 100 frames of a high-definition video (Full HD images of 1 920 × 1 080 pixels).

The design of this experimental study minimizes the internal validity threats. The experiments are conducted with the machines in the same conditions, with controlled temperature and exclusive use to avoid interference from other processes during the experiments. Several measures are done at different times, and data is statistically analyzed. A full range of valid values of the input parameters is considered. Thus, history, maturation, instrumentation and testing threats are minimal. The same combinations of study cases and parameter values are tested for every programming model considered, eliminating selection bias and attrition threats. Regarding external validity, the selection of the study cases and input parameters ensure that we are checking situations with bottlenecks in either the host-task or the kernels, with memory bounded, computation bounded, and I/O bounded scenarios, using a full range of parameter values that select the different situations and tune the amount of potential overlapping. Nevertheless, the study is limited to simple iterative applications with regular loads. More complex real applications should be tested to further generalize the results.

5.3. *Performance study*

Figs. 7, 8, and 9 show the performance results obtained for the different versions of the Hotspot, Matrix Pow, and Sobel case studies, respectively. Results using the NVIDIA Tesla V100 and the AMD WX 9100 are shown in separated plots. Due to the high differences of scale, logarithmic scale is used in both, the x- and y-axis.

From the analysis of the performance results, we can draw the following observations.

First, a profiler analysis of the execution of the Controller programs reveals that for each study case and combination of input parameters, the bottleneck is in one of the computational units; the CPU or the GPU device. On each case, the asynchronous mode of CtrlEvents achieves a usage of more than 99% of the bottleneck computational unit. The corresponding sequence of either the host-tasks or the kernels is executed without any significant delay between them, and both the data-transfers and the computations in the other computational unit are completely overlapped.

Second, regarding the versions that use CUDA, in almost all the cases, the Controller code with synchronous execution policy presents the same performance as the synchronous native implementation in CUDA. The same situation can be observed between the Controller code with asynchronous execution policy and the asynchronous native CUDA implementation. Remarkable situations appear, for example, for the very fast kernels surrounded by data-transfer operations of the Sobel filter. The results indicate that the bottleneck for these programs is always in the host tasks. In Fig. 10, we can see a graphical representation provided by the

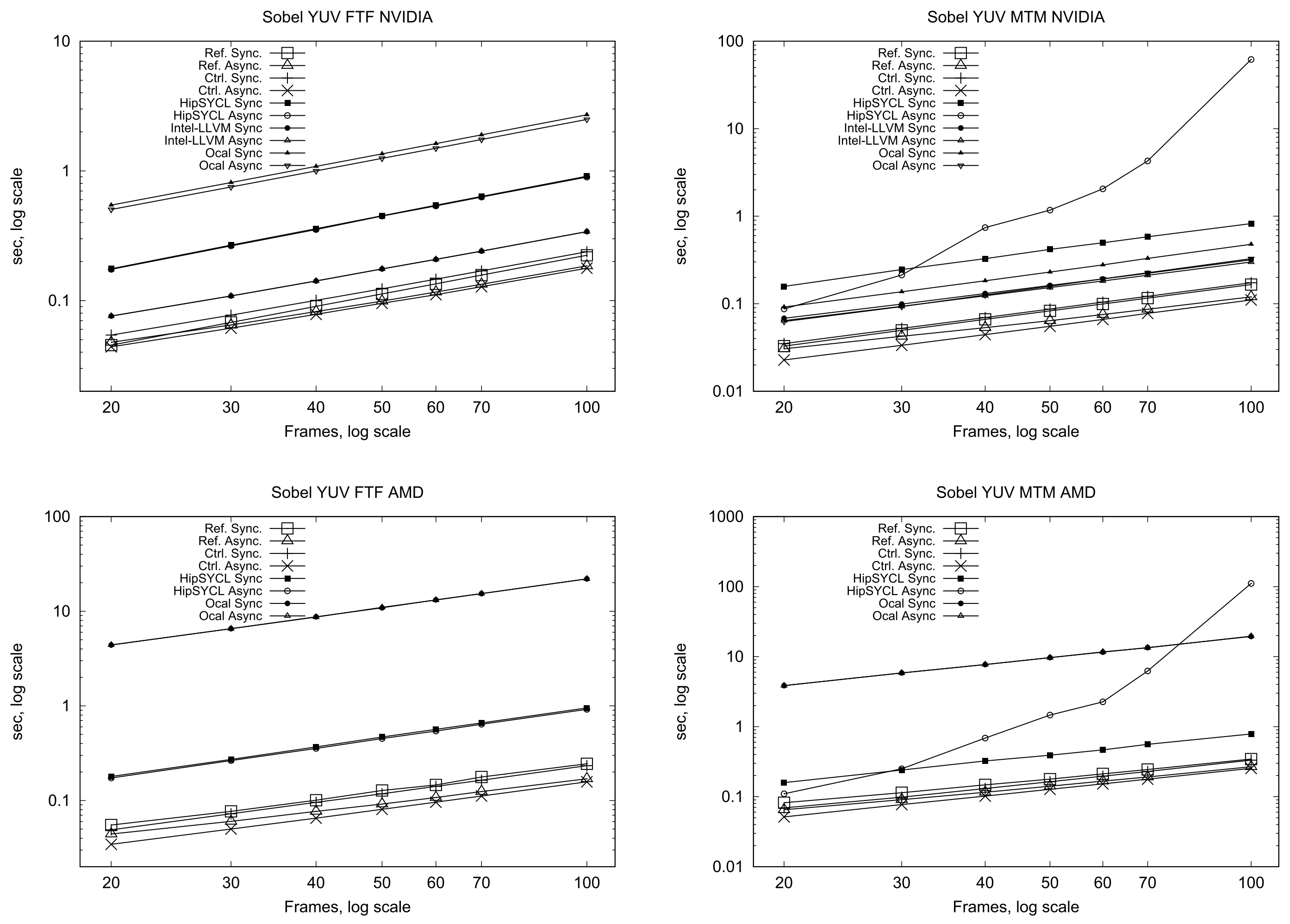


Fig. 9. Experimental results for two Sobel case studies: FTF (File To File) and MTM (Memory to Memory). They represent the scenarios where the host tasks are the slowest and fastest respectively. The other cases FTM and MTF present results between them. The results were calculated using CUDA ("NVIDIA" plots) and OpenCL ("AMD" plots).

CUDA visual profiler of the execution on both synchronous and asynchronous modes of the Sobel memory-to-memory program. As can be seen, the synchronous version sequentially executes all operations (kernel executions, communications, and host tasks), while the asynchronous version overlaps these operations whenever possible. Besides this, the way that the CtrlEvents implements the synchronizations with CUDA and OpenCL events makes our versions faster than the manual reference implementations, even the synchronous one, which simply consist of a sequence of blocking operations. Regarding overall performance, the Controller asynchronous programs process up to 500 frames per second for "File to File", or near 1 000 frames per second for "Memory to Memory". A similar advantage can be noticed in the Hotspot programs, but only for the AMD platform with OpenCL. Therefore, with CtrlEvents the overlapping is in general complete, as in the case of Matrix and Hotspot. The exceptions occur when the size of the data to be transferred is too small, or if there are limitations due to synchronization activities, which is precisely the case of Sobel, since the write operations to a file forces a synchronization.

Third, the current implementation of hipSYCL has a problem with the queue management, that appears when many requests saturate the queue. In the synchronous versions of the programs, the problem does not appear because the host code is synchronized with the operations, and the queue has very few requests at the same time. In the asynchronous versions, when enough iterations are launched, the queue is saturated, and the execution times increase even much higher than the synchronous versions with artificially forced synchronizations. In the "File to File" Sobel filter, the execution times are low, and the good balance between the communications and computations avoids the problem. For small input sizes, the Hotspot case study also shows a very bad performance in both synchronous and asynchronous versions. In general, the hipSYCL programs perform badly in scenarios with small input sizes or low-loaded computations and communications. For scenarios with higher loads, such as medium to big input sizes in matrix-pow, the hipSYCL asynchronous versions perform as well as the Controller and reference programs manually developed and optimized in CUDA or OpenCL.

Fourth, as we discussed in Sect. 5.2 the asynchronous SYCL programs compiled with Intel LLVM-CUDA need an extra synchronization to produce correct results. This modification converts them into almost-synchronous versions. Thus, the performance is almost the same in both asynchronous and synchronous versions. In general, these codes perform worse than the synchronous reference versions. The only remarkable case of very good performance appears for the biggest size tested in the Hotspot program, where the execution time converges to the one observed for both the asynchronous reference and the Controller codes.

Finally, regarding dOCAL performance, the results for NVIDIA of the Sync and Async versions of Matrix Pow and Hotspot are clearly worse than those obtained for Controllers and SYCL. With respect to AMD, in the case of Matrix Pow and Hotspot, dOCAL also performs worse than Controllers and SYCL, although, for the biggest problem sizes, dOCAL Async shows similar performance

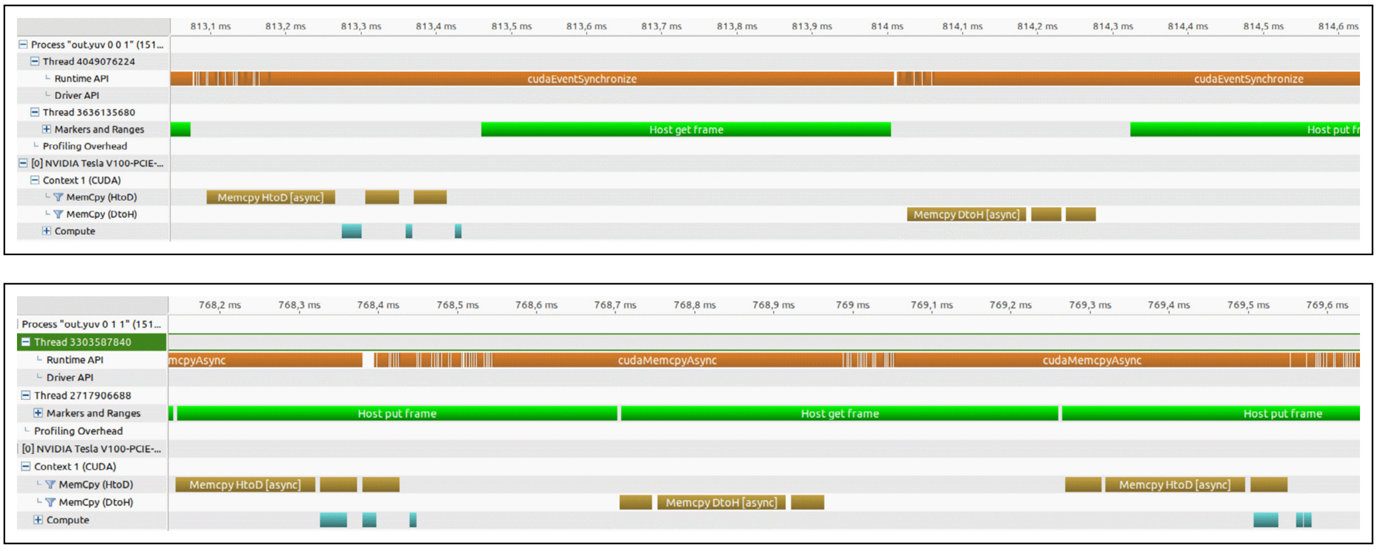


Fig. 10. Screen captures of the CUDA 11.2 Visual Profiler showing a range of approximately 1.4 ms of execution in the Manticore machine of the Controller program implementing the Sobel filter, version MTM, with input images and results stored in host memory: Synchronous mode (top), and asynchronous mode (bottom).

figures than those obtained with the Controller Sync version and the reference implementation. With respect to Sobel F2F, dOCAL has the same problem that we discussed above regarding SYCL: An explicit synchronization is needed after each file read/write operation. Due to its peculiarities, dOCAL does not need a host task, but even without it, the execution time is one order of magnitude bigger than SYCL and Controllers. Finally, with respect to Sobel M2M, the Sync version of dOCAL for NVIDIA works better than HypSCYL, and the Async dOCAL version shows the same performance than Intel-LLVM Async. However, in AMD, the execution time of both the Sync and Async version of dOCAL is one order of magnitude bigger than the alternatives studied.

5.4. *Development effort measures*

This section analyzes the differences in development effort between all the versions of the developed codes. We consider four classical development effort metrics: The number of lines of code, Number of tokens, McCabe's cyclomatic complexity [25], and Halstead's development effort [17]. The first two measure the volume of code that the programmer should develop. The third 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 and characterization, the coordination host code, and the data structures management. For a fair comparison, they have been formatted following the same criteria, with no line breaks in expressions or calls to functions, closing curly brackets always on their own line, etc. Regarding dOCAL, the complexity measurement method counted kernels just as single strings, thus preventing a proper measurement. Therefore, just for this comparison, we transformed kernels into separate functions, in order to measure its complexity more accurately.

Table 2 shows the results. They indicate that programming using the Controller library generates a lower volume of code, a reduced cyclomatic complexity, and reduced Halstead measures than both synchronous and asynchronous versions using CUDA or OpenCL (e.g., from 20% to 70% less Halstead's effort). The highest reductions are found when comparing with the asynchronous baseline versions, whose native versions in CUDA or OpenCL manually introduce more complex mechanisms for kernel and data transfer synchronizations. These mechanisms are transparent and portable in the Controller programs. A closer look at the codes indicates that, as expected, the higher reduction is found in the parts of the host codes related to coordination. The small differences between a Controller program using CUDA or OpenCL kernels are inside the kernels, or in one extra program argument added to the *main* function in OpenCL version to select the GPU platform, that is not needed in the CUDA only version. The coordination code and host tasks are completely portable.

Table 2
Measurements of development effort metrics for the reference and Controller codes. It includes a comparison between the number of code lines (LOC), number of code tokens (TOK), McCabe's cyclomatic complexity (CCN) and Halstead's development effort metric (Halstead).

Case study	Version	LOC	TOK	CCN	Halstead
Hotspot	Ctrl (CUDA kernels)	150	1 493	32	475 799
	Ctrl (OpenCL kernels)	152	1 520	32	447 411
	dOCAL Sync	153	1 646	33	493 637
	dOCAL Async	151	1 626	33	484 398
	SYCL Sync	165	1 835	32	646 559
	SYCL Async	163	1 823	32	641 711
	CUDA Sync	162	1 769	33	710 780
	CUDA Async	203	2 310	37	1 194 609
	OpenCL Sync	242	2 734	37	1 193 405
	OpenCL Async	314	3 345	37	1 712 732
Matrix Pow	Ctrl (CUDA kernels)	101	929	18	250 311
	Ctrl (OpenCL kernels)	106	958	18	229 714
	dOCAL Sync	104	1 028	17	233 938
	dOCAL Async	102	1 008	17	226 546
	SYCL Sync	127	1 353	20	465 916
	SYCL Async	125	1 341	20	461 310
	CUDA Sync	144	1 297	25	444 197
	CUDA Async	205	1 735	25	862 150
	OpenCL Sync	203	1 934	25	668 596
	OpenCL Async	278	2 383	28	936 740
Sobel filter (File-to-File)	Ctrl (CUDA kernels)	101	1 338	17	431 889
	Ctrl (OpenCL kernels)	109	1 351	17	433 397
	dOCAL Sync	110	1 391	24	479 865
	dOCAL Async	109	1 381	24	479 803
	SYCL Sync	131	1 726	22	779 085
	SYCL Async	130	1 720	22	773 085
	CUDA Sync	100	1 324	23	546 646
	CUDA Async	167	1 929	30	878 473
	OpenCL Sync	151	1 948	18	823 562
	OpenCL Async	227	2 505	26	1 249 711

SYCL and dOCAL are based on modern C++ abstractions. The same SYCL code is used with both compilers: hipSYCL and Intel-LLVM. dOCAL abstractions are more powerful than SYCL, presenting lower measures except for the notable cases of the slightly higher cyclomatic complexity on the Hotspot and Sobel cases. There is less volume of code but with more logical branches. The Controller programs report clearly better measures than SYCL programs in all cases (e.g. 25% to 45% reduction in Halstead's effort). They also show similar measures than dOCAL in most situations, with a slight reduction of 5% to 8% in tokens, reductions of less than 10% in Halstead's effort for Hotspot and Sobel, and an increment of 10% in Halstead's effort for Matrix Pow. The Controller codes present slightly lower cyclomatic complexity than dOCAL in Hotspot (reduction of 3%), slightly higher in Matrix Pow (increment of 5%, and a significant reduction of 29% in Sobel. The best case for the Controller programs is Sobel, a streaming application that applies an image filter to a video stream with potential to overlap kernels, host tasks and communications in both directions. In this case the Controller program presents the best measures for all the metrics comparing with any other model. These results indicate that the Controller model abstractions are competitive with other state-of-the-art high-level abstractions.

6. Conclusion

In this work, we present a proposal to simplify and improve the efficiency of the runtime data-dependencies analysis and the coordination of both device computations and host tasks that include generic operations, such as input/output. We discuss the design and implementation of CtrlEvents, a functional prototype that implements this abstraction. CtrlEvents extends our previous Controller heterogeneous programming framework, adding the possibility of executing asynchronous operations in a portable and efficient form, and removing some inefficiencies of the original proposal, such as the use of internal queues. We also present an experimental study using several iterative programs with different scenarios for asynchronous operations between host and device, and several demanding workload scenarios, including typical streaming operations. Implementations using our proposal and two different state-of-the-art SYCL implementations (hipSYCL with CUDA and ROCm backends, and Intel LLVM for CUDA) are compared on both NVIDIA and AMD GPUs. The results indicate that our solution outperforms the SYCL implementations for most situations, presenting a good balance between portability, programmability and efficiency. Future work includes a more extensive study of the portability and cooperation between different and new types of devices, and the behavior with other types of applications.

The benefits of CtrlEvents are related to the asynchronous execution and the potential overlapping of computations and communications. Thus, they are limited to iterative or task-based programs that interleave device and host computations that requires memory transfers between them due to dependences. The experimental study presented in this work is limited to simple iterative applications with regular loads. More complex real applications should be tested in future work to further generalize the results.

CRediT authorship contribution statement

Yuri Torres: Investigation, Software, Original draft preparation, Validation, Writing – Reviewing and Editing.

Francisco Andújar: Investigation, Software, Original draft preparation, Validation, Writing – Reviewing and Editing.

Arturo Gonzalez-Escribano: Conceptualization, Methodology, Software, Writing – Original draft preparation, Writing – Reviewing and Editing, supervision.

Diego R. Llanos: Investigation, Writing – Original draft preparation, Writing – Reviewing and Editing, supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The authors would like to thank the Editor and the anonymous reviewers for their valuable suggestions. This work was supported in part by the Spanish Ministerio de Ciencia e Innovación and by the European Regional Development Fund (ERDF) program of the European Union, under Grant TIN2017-88614-R (PCAS Project); in part by the Junta de Castilla y León - FEDER Grants, under Grant VA226P20 (PROPHET-2 Project), by Junta de Castilla y León, Spain. This work was also supported in part by grant TED2021–130367B–I00, funded by MCIN/AEI/10.13039/501100011033 and by "European Union NextGenerationEU/PRTR".

References

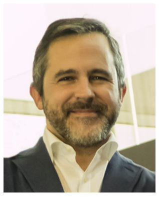
[1] A. Alpay, V. Heuveline, SYCL beyond OpenCL: the architecture, current state and future direction of HipSYCL, in: Proceedings of the International Workshop on OpenCL, Association for Computing Machinery, New York, NY, USA, 2020.
[2] G. Araujo, D. Griebler, D.A. Rockenbach, M. Danelutto, L.G. Fernandes, Nas parallel benchmarks with cuda and beyond, Softw. Pract. Exp. (2021).
[3] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, et al., The nas parallel benchmarks summary and preliminary results, in: Supercomputing'91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, IEEE, 1991, pp. 158–165.
[4] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R. Scogland, Raja: portable performance for large-scale scientific applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 71–81.
[5] T. Ben-Nun, M. Sutton, S. Pai, K. Pingali, Groute: an asynchronous multi-GPU programming model for irregular computations, in: Proc. PPoPP '17, ACM, Austin, Texas, USA, 2017, pp. 235–248.
[6] BSC (Barcelona Supercomputing Center), mpSs2 Programming model, https://pm.bsc.es/ompss-2, 2020. (Accessed 1 February 2022).
[7] C++ Standards Committee Papers, A unified executors proposal for C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html, 2020. (Accessed 1 February 2022).
[8] S. Che, et al., Rodinia: a benchmark suite for heterogeneous computing, in: Proc. IISWC'09, IEEE, 2009, pp. 44–54.
[9] I. Corporation, intel/llvm: home for intel llvm-based projects, https://github.com/intel/llvm, 2021. (Accessed 1 May 2021).
[10] Cuda Programing Blog, Implementation Sobel operator in CUDA C on YUV video File, Web, http://cuda-programming.blogspot.com/2013/01/implementation-sobel-operator-in-cuda-c.html, 2013. (Accessed 1 February 2022).
[11] P. Diehl, M. Seshadri, T. Heller, H. Kaiser, Integration of CUDA processing within the C++ library for parallelism and concurrency (HPX), arXiv:1810.11482, 2018.
[12] H.C. Edwards, C.R. Trott, Kokkos: enabling performance portability across manycore architectures, in: 2013 Extreme Scaling Workshop (xsw 2013), 2013, pp. 18–24.
[13] M. Fragkoulis, P. Carbone, V. Kalavri, A. Katsifodimos, A survey on the evolution of stream processing systems, arXiv:1810.11482, 2020.
[14] R. Gonzalez, R. Woods, Digital Image Processing, 3rd edition, Prentice Hall, 2007.
[15] A. Gonzalez-Escribano, Y. Torres, J. Fresno, D. Llanos, An extensible system for multilevel automatic data partition and mapping, IEEE Trans. Parallel Distrib. Syst. 25 (2014) 1145–1154, https://doi.org/10.1109/TPDS.2013.83.
[16] T. Gysi, J. Bär, T. Hoefler, dCUDA: hardware supported overlap of computation and communication, in: Proc. SC16, IEEE, Salt Lake City, Utah, EE. UU., 2016, pp. 609–620.
[17] M. Halstead, Elements of Software Science, Operating and Programming Systems Series, Elsevier Science Inc., 1977.

[18] Intel, Sobel Filter Design Example, https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/sobel-filter.html, 2018. (Accessed 1 August 2020).
[19] Intel Corporation, Intel oneAPI webpage, https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html, 2022. (Accessed 1 February 2022).
[20] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, HPX: a task based programming model in a global address space, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14, Association for Computing Machinery, New York, NY, USA, 2014.
[21] Khronos OpenCL working group, SYCL 1.2.1 specification standard, https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, 2020. (Accessed 1 February 2022).
[22] Lawrence Livermore National Laboratory, RAJA Performance Portability Layer (C++), https://github.com/LLNL/RAJA, 2020. (Accessed 1 February 2022).
[23] Y. LeCun, 1.1 deep learning hardware: past, present, and future, in: 2019 IEEE International Solid- State Circuits Conference - (ISSCC), 2019, pp. 12–19.
[24] B.B. Mandelbrot, Fractal aspects of the iteration of z → λz (1-z) for complex λ and z, Ann. N.Y. Acad. Sci. 357 (1980) 249–259.
[25] T. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 4 (1976) 308–320, https://doi.org/10.1109/TSE.1976.233837.
[26] A. Moreton-Fernandez, E. Rodriguez-Gutiez, A. Gonzalez-Escribano, D. Llanos, Supporting the Xeon Phi coprocessor in a heterogeneous programming model, in: Proc. Euro-Par 2017, Springer, Santiago de Compostela, Spain, 2017, pp. 457–469.
[27] A. Moreton-Fernandez, H. Ortega-Arranz, A. Gonzalez-Escribano, Controllers: an abstraction to ease the use of hardware accelerators, Int. J. High Perform. Comput. Appl. 32 (2018) 838–853, https://doi.org/10.1177/1094342017702962.
[28] A. Murray, E. Crawford, Compute aorta: a toolkit for implementing heterogeneous programming models, in: Proceedings of the International Workshop on OpenCL, IWOCL '20, Association for Computing Machinery, New York, NY, USA, 2020.
[29] NVIDIA, CUDA Toolkit Documentation v11.6.0, http://docs.nvidia.com/cuda/, 2022. (Accessed 1 February 2022).
[30] NVIDIA, matrixMul - matrix multiplication (CUDA Runtime API version), https://docs.nvidia.com/cuda/cuda-samples/index.html#matrix-multiplication--cuda-runtime-api-version-, 2022. (Accessed 9 February 2022).
[31] S. Pennycook, J. Sewall, V. Lee, Implications of a metric for performance portability, Future Gener. Comput. Syst. 92 (2019) 947–958, https://doi.org/10.1016/j.future.2017.08.007.
[32] B. Pérez, J. Bosque, R. Beivide, Simplifying programming and load balancing of data parallel applications on heterogeneous systems, in: Proc. GPGPU '16, ACM, Barcelona, Spain, 2016, pp. 42–51.
[33] L-N. Pouchet, et al., PolyBench/C, the Polyhedral Benchmark suite, GPU 1.0, http://web.cs.ucla.edu/~pouchet/software/polybench, 2012. (Accessed 1 February 2022).
[34] A. Rasch, J. Bigge, M. Wrodarczyk, R. Schulze, S. Gorlatch, dOCAL: high-level distributed programming with OpenCL and CUDA, J. Supercomput. 76 (2020) 5117–5138, https://doi.org/10.1007/s11227-019-02829-2.
[35] G. Rodriguez-Canal, Y. Torres, F.J. Andújar, A. Gonzalez-Escribano, Efficient heterogeneous programming with fpgas using the controller model, J. Supercomput. (2021), https://doi.org/10.1007/s11227-021-03792-7.
[36] Sandia National Laboratory, Kokkos C++ Performance Portability Programming EcoSystem: The Programming Model – Parallel Execution and Memory, https://github.com/Kokkos/kokkos, 2020. (Accessed 1 February 2022).
[37] M.J. Schulte, M. Ignatowski, G.H. Loh, B.M. Beckmann, W.C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S.K. Reinhardt, G. Rodgers, Achieving exascale capabilities through heterogeneous computing, IEEE MICRO 35 (2015) 26–36, https://doi.org/10.1109/MM.2015.71.
[38] The Khronos Group Inc, Open Computing Language (OpenCL), http://www.khronos.org/opencl/, 2022. (Accessed 1 February 2022).
[39] The Kokkos Team, The Kokkos Lectures, Module 2: Views and Spaces, https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/KokkosTutorial_02_ViewsAndSpaces.pdf, 2020. (Accessed 26 July 2022).
[40] The Kokkos Team, The Kokkos Lectures, Module 5: Stream, Tasking and SIMD, https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/KokkosTutorial_05_SIMDStreamsTasking.pdf, 2020. (Accessed 26 July 2022).
[41] P. Thoman, et al., A taxonomy of task-based parallel programming technologies for high-performance computing, J. Supercomput. 74 (2018) 1422–1434, https://doi.org/10.1007/s11227-018-2238-4.
[42] Trasgo Group, Controllers v2 heterogeneous programming framework, https://gitlab.com/trasgo-group-valladolid/controllers/-/tags/Controllers_v2, 2021. (Accessed 1 February 2022).
[43] triSYCL, The triSYCL Project, https://github.com/triSYCL/triSYCL, 2021. (Accessed 15 November 2021).
[44] R. Vasudevan, S. Vadhiyar, L. Kalé, G-Charm: an adaptive runtime system for message-driven parallel applications on hybrid systems, in: Proc. ICS 2013, ACM, Eugene, Oregon, United States, 2013, pp. 349–358.
[45] S. Venkatasubramanian, R. Vuduc, Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems, in: Proc. ICS'09, ACM, Yorktown Heights, NY, USA, 2009, pp. 244–255.
[46] J.S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, J. Wilke, Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity, Report for DOE ASCR Workshop on Extreme Heterogeneity, Technical Report, U.S. Department of Energy Office of Scientific and Technical Information, 2018, https://doi.org/10.2172/1473756.
[47] A. Vilches, A. Navarro, F. Corbera, A. Rodriguez, R. Asenjo, Heterogeneous parallel for template based on TBBs, in: Proc. HLPP'17, Springer, Valladolid, Spain, 2017.
[48] L. Wang, W. Wu, Z. Xu, J. Xiao, Y. Yang, BLASX: a high performance level-3 BLAS library for heterogeneous multi-GPU computing, in: Proc. ICS '16, ACM, Istanbul, Turkey, 2016, pp. 20:1–20:11.

Yuri Torres de la Sierra received the B.S. degree in Computer Science and Engineering from University of Valladolid, Spain, in 2009. He received the M.S. degree in Information Communications in 2010, and the Ph.D. degree in Computer Science in 2014, both from the University of Valladolid, Spain. From 2014 to 2017, he was Associate Professor at Isabel I University, Burgos, Spain. He is currently Assistant Professor of computer science at the Universidad of Valladolid. His research interests include parallel and distributed computing, parallel programming models, and embedded computing. More information about his current research activities can be found at http://www.infor.uva.es/~yuri.torres.

Francisco J. Andújar received the M.Sc. degree in Computer Science from the University of Castilla-La Mancha, Spain, in 2010, and the Ph.D. degree from the University of Castilla-La Mancha in 2015. He worked at the Universitat Politécnica de Valencia under a post-doctoral contract Juan de la Cierva, and he currently works at the University of Valladolid as Assistant Professor. His research interests include multicomputer systems, cluster computing, HPC interconnection networks, switch architecture and simulation tools.

Arturo Gonzalez-Escribano received his MS and PhD degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2003, respectively. Dr. Gonzalez-Escribano is Associate Professor of Computer Science at the Universidad de Valladolid since 2008. His research interests include parallel and distributed computing, parallel programming models, portability in heterogeneous systems, and embedded computing. More information about his current research activities can be found at http://www.infor.uva.es/~arturo.



Diego R. Llanos received his MS and PhD degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is a recipient of the Spanish government's national award for academic excellence. Prof. Llanos is Full Professor of Computer Architecture at the Universidad de Valladolid, and his research interests include parallel and distributed computing, automatic parallelization of sequential code, and embedded computing. He is a Senior Member of the IEEE and Senior Member of the ACM, and has co-founded RDNest, a company that transfers to market research results in the field of Internet of Things and high-performance computing. More information about his current research activities can be found at http://www.infor.uva.es/~diego.