electronics

Article

Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime

Raúl Nozal * and Jose Luis Bosque

Department of Computer Science and Electronics, Universidad de Cantabria, 39005 Santander, Spain; joseluis.bosque@unican.es
* Correspondence: raul.nozal@unican.es

Abstract: Heterogeneous systems are the core architecture of most computing systems, from high-performance computing nodes to embedded devices, due to their excellent performance and energy efficiency. Efficiently programming these systems has become a major challenge due to the complexity of their architectures and the efforts required to provide them with co-execution capabilities that can fully exploit the applications. There are many proposals to simplify the programming and management of acceleration devices and multi-core CPUs. However, in many cases, portability and ease of use compromise the efficiency of different devices—even more so when co-executing. Intel oneAPI, a new and powerful standards-based unified programming model, built on top of SYCL, addresses these issues. In this paper, oneAPI is provided with co-execution strategies to run the same kernel between different devices, enabling the exploitation of static and dynamic policies. This work evaluates the performance and energy efficiency for a well-known set of regular and irregular HPC benchmarks, using two heterogeneous systems composed of an integrated GPU and CPU. Static and dynamic load balancers are integrated and evaluated, highlighting single and co-execution strategies and the most significant key points of this promising technology. Experimental results show that co-execution is worthwhile when using dynamic algorithms and improves the efficiency even further when using unified shared memory.

Keywords: heterogeneous systems; parallel computing; co-execution; load balancing; SYCL; oneAPI; Data Parallel C++; scheduling; HPC; CPU-GPU

check for updates

Citation: Nozal, R.; Bosque, J.L. Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime. *Electronics* **2021**, *10*, 2386. https://doi.org/10.3390/electronics10192386

Academic Editor: George A. Tsihrintzis

Received: 13 August 2021
Accepted: 22 September 2021
Published: 29 September 2021

1. Introduction

In recent years, with the quest to constantly improve the performance and energy efficiency of computing systems, together with the diversity of architectures and computing devices, it has become possible to exploit an interesting variety of problems due to heterogeneous systems. The future of computing cannot be understood without these types of systems [1]. With the emergence of new specialized devices, a world of massively parallel computing opens up that is able to combine HPC devices, such as Intel Xeon processors powered by UHD integrated graphics processors, FPGAs, discrete GPUs like Intel Iris Xe, and even XPU devices, specialized in machine learning inference.

The keys to their importance are the excellent cost/performance ratio and energy efficiency. These characteristics allow the acceleration of a wide range of massively data-parallel applications, such as deep learning [2,3], image and video processing [4–6], and financial applications [7,8].

However, hardware heterogeneity complicates the development of efficient and portable software, especially when specialized components from various suppliers are used, since many require their own programming models. In this context, there are several hot topics that are currently being researched: supporting single source programming [9], improving the usability and efficiency of memory space [10], distributing computation and data between different devices [11], and load balancing [12], among others. In an effort to improve these topics, a variety of frameworks and tools have been proposed [13–17].

Programming models have become more abstract and expressive. OpenCL emerged as an open standard programming model for writing portable programs across heterogeneous platforms [18]. However, it has a very low level of abstraction and leaves to programmers the partitioning and transferring of data and results among the CPU and devices. Furthermore, the support from different vendors has been variable both in compatibility and performance. Even so, it continues to be the de facto open standard, and new manufacturers continue to incorporate OpenCL drivers in a range of platforms. Nevertheless, many applications have their kernels implemented in OpenCL, thus, benefiting from its support and battle-tested usage. On the other end, proposals based on compiler directives have been developed, such as OpenACC [19], and later extensions of OpenMP [20], leaving all this work to the compiler, but limiting both the expressiveness and performance.

In this way, the Khronos® Group proposed SYCL-based heterogeneous parallel programming framework for accelerating High Performance Computing, machine learning, embedded computing, and compute-intensive desktop applications on a wide range of processor architectures, including CPUs, GPUs, FPGAs, and AI processors. Moreover, market trends and industrial applications indicate a strong predominance of languages, such as C++, favoring higher level alternatives. For instance, SYCL is a cross-platform abstraction layer that builds on OpenCL, enabling the host and kernel code to be contained in the same source file with the simplicity of a cross-platform asynchronous task graph [21]. Several implementations from different vendors have been released, such as ComputeCpp, triSYCL, and hipSCYL [22,23].

In this context, Intel developed oneAPI, a unified programming model to facilitate development among various hardware architectures [24]. It provides a runtime, a set of domain-focused libraries and a simplified language to express parallelism in heterogeneous platforms. It is based on industry standards and open specifications, offering consistent tooling support and interoperability with existing HPC programming models.

The oneAPI's cross-architecture language Data Parallel C++ (DPC++) [25], based on SYCL standard for heterogeneous programming in C++, provides a single, unified open development model for productive heterogeneous programming and cross-vendor support. It allows code reuse across hardware targets while permitting custom tuning for a specific accelerator. Since the abstraction level is increased, the productivity improves and a new set of features is provided, including optimized communication patterns, automatic dependency tracking, runtime scheduling, and shared memory optimizations.

This article addresses a new challenge in improving the usability and exploitation of heterogeneous systems, providing oneAPI with the capacity for *co-execution*. This is defined as the collaboration of all the devices in the system (including the CPU) to execute a single massively data-parallel kernel [14,26–28]. However, this is a difficult task for the programmer and needs to be effortless in order to be widely used. In this way, the expression and abstraction capabilities of oneAPI, such as portability and single-source style, will be exploited to obtain codes that will be easier to implement and maintain. To efficiently exploit the computing capacity of all devices, a series of workload balancing algorithms are implemented, both static and dynamic, obtaining good results with both regular and irregular applications.

Coexecutor Runtime is presented, extending the work and preliminary results of the conference paper [29]. The key innovations are the high level API, increasing the abstraction but maintaining its compatibility and extensibility with SYCL; an efficient architectural design focused on preserving and reusing as many oneAPI primitives as possible while extending its functionality; and as far as we know, it is the first co-execution runtime for Intel oneAPI. This paper details the fundamentals of its design decisions and featured models, provides a comprehensive validation using four different architectures in two types of heterogeneous nodes, and showcases the usability and flexibility of its API design, exploiting the technology enhancements.

To validate the co-execution proposal and determine its efficiency, an empirical evaluation was performed. Experimental results show that co-execution is worthwhile from the point of view of performance and energy efficiency as long as dynamic schedulers are used, and even more if unified memory is applied.

The main contributions of this paper are the following:

- Providing Intel oneAPI with co-execution capabilities, squeezing the capacity of the heterogeneous system to exploit a single kernel.
- Implementing a set of load-balancing algorithms that allow efficient use of all devices in the system, both with regular and irregular applications.
- Validating the co-execution runtime in terms of the performance, balancing efficiency, scalability, and energy efficiency for a set of benchmarks and schedulers.

The rest of the paper is organized as follows. Sections 2 and 3 describe general background concepts needed along with the issues that motivate this work. Section 4 exposes the co-execution architecture and its design decisions, including examples of usage and its API. The methodology used for the validation is explained in Section 5, while the experimental results are shown in Section 6. Finally, Section 7 shows other related works, while Section 8 highlights the most important conclusions and future work.

2. Background

OneAPI is based on the SYCL specification, although it provides its own extensions to accelerate the computation and facilitate the development. Its programming language is called Data Parallel C++ (DPC++), making a leap in abstraction and promoting interoperability with the host code, compared to the OpenCL language. DPC++ is a community-driven, standards-based language built on ISO C++ and Khronos SYCL, allowing developers to reuse code across hardware targets. DPC++ allows the host and the device code as part of the same compilation unit, as a feature called single source property, that allows potential optimizations across the boundary between both codes. Due to this property, DPC++ establish three types of scope to distinguish between the host (application), host–device interface (command group), and device (kernel). OneAPI comprises four models based on SYCL, each of which is part of the operations that a developer has to take when using oneAPI.

2.1. Platform Model

The platform model defines a host that manages one or more devices and coordinates the application and command group scopes. A device can be an accelerator or the CPU itself, each of which contains a set of Compute Units. In the same manner, each of these provides one or more processing elements. The complete system could have multiple platforms, since the composition of drivers in execution platforms is determined by the drivers and their implementation.

2.2. Execution Model

The execution model defines and specifies how kernels execute on the devices and interact with the host. It is subdivided into the host and device execution models. The data management and execution between host and devices are coordinated by the host execution model via command groups. These are groupings of commands, like the kernel invocation and accessors, which are submitted to queues for execution. The device execution model specifies how computation is accomplished on the accelerator, specifying the range data sets. These are allocated across a hierarchy of ND-ranges, work-groups, sub-groups, and work-items, easing the programming patterns and their composition. This facilitates the memory and compute operation relationships, giving the programmer flexibility to express the algorithms.

A fundamental concept in the SYCL execution model is the Directed Acyclic Graph (DAG). Each node contains an action to be performed on a device, such as kernel invocation or data movements. The SYCL runtime controls, asynchronously, the resolution of dependencies and triggering of node executions. Thus, it tracks and orchestrates actions and their dependencies to perform in the devices, safely executing each operation when the requirements are met. On the other side, if the handler is not used, the code executes synchronously by the CPU as part of the host program, bypassing the DAG.

2.3. Memory Model

The memory model coordinates the allocation and management of memory between the host and devices and how they interact. Memory resides upon and is owned by either the host or the device and is specified by declaring a memory object. *Accessors* define the interaction of these memory objects between host and device, communicating the desired location and access mode.

An extension to the standard SYCL memory model is Unified Shared Memory (USM), which enables the sharing of memory between the host and devices without explicit accessors. It manages access and enforces dependencies with explicit functions to wait on events or by signaling a dependency relationship between events. Another important feature of USM is that it provides a C++ pointer-based alternative to the buffer programming model, which increases the abstraction by leaving the migration of memory to the underlying runtime and device drivers. On the other side, since it does not rely on accessors, dependencies between command group operations must be specified using events to help the compiler determine the data dependencies and patterns.

2.4. Kernel Programming Model

The kernel is the computing function instantiated to be executed by every processing element of the accelerator. It allows the programmer to determine what code executes on the host and device, giving an explicit computing function via lambda expression, functor or kernel class. Therefore, the separation of host and device codes is straightforward, without language extensions. Device code can specify the parallelism mechanism with a coarse-grained task, data-parallel work or data-parallel construct taking into consideration the hierarchical range of the execution model.

It supports the single source property, meaning the host code and device code can be in the same source file. Therefore, it improves the usability, safety between host and device boundaries (matching kernel arguments), and optimization strategies due to better understanding of the execution context (aliasing inference and propagating constants). Finally, DPC++ kernels execute asynchronously via forced allocations of kernel class instances, implicit waits of C++ destructors or explicit queue waits.

3. Motivation

The host–device programming model is used in Intel oneAPI, where the host offloads compute-heavy functions, called kernels, to a set of hardware accelerators, such as GPUs and FPGAs. Its runtime is able to manage complex applications composed of a set of kernels, even if they have dependencies between them, through a Directed Acyclic Graph (DAG). The assignment of a kernel to a particular device can be done by the programmer; therefore, it is determined at compile time, or let oneAPI chooses the device at runtime. In either case, a kernel can only be scheduled to a single device when the dependencies are satisfied.

However, in this scenario, the only possibility of co-execution is for the programmer to split the work into several kernels, as many as there are devices in the system. The data partition and workload distribution must be done manually. Furthermore, the compiler must detect that these kernels are independent and schedule them simultaneously. This complicates the co-execution and, therefore, the exploitation of the whole system to solve a single kernel.

Even if the programmer is willing to face this extra effort, an additional problem arises with workload balancing. Since the division of the workload is done at the compile time, it is necessarily static. That is, the portion of work assigned to each device is pre-fixed at the beginning of execution. This partitioning works well for regular applications, where the execution time of a data set depends only on its size [26]. The programmer needs to estimate off-line how much workload to allocate to each device so that both finish at the same time, thus, obtaining a balanced execution, as seen in the left part of Figure 1, for the Gaussian kernel. In this case, the kernel execution time is 5 s on the CPU and 2 s on the GPU, which means that the GPU has 2.5× the performance of the CPU.

Each device takes about the same amount of time to compute each pixel, regardless of color or intensity. Therefore, assigning the work to devices proportionally to their computing capabilities, a balanced distribution is obtained, and the execution time is reduced to approximately 1.5 s.

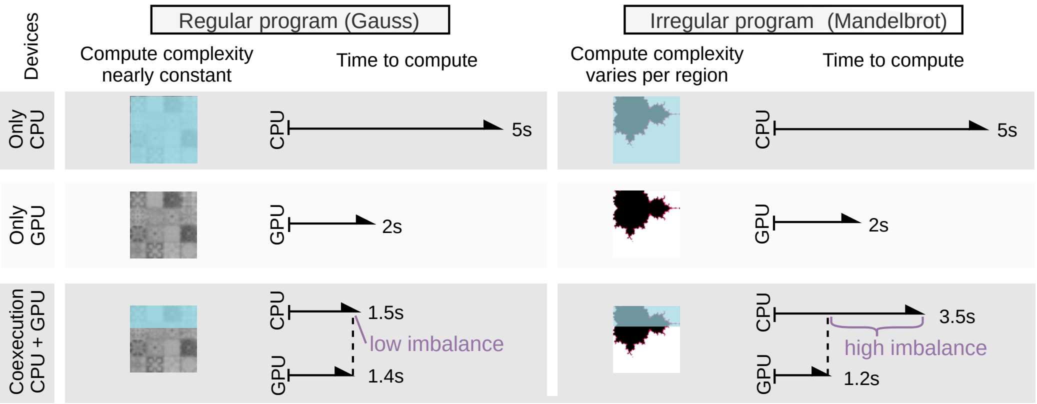


Figure 1. Static co-execution for regular and irregular programs.

However, it is well known that static scheduling cannot adapt to the irregular behavior of many applications, leading to significant load imbalances [14]. In these applications, the processing time of a data set depends not only on its size but also on the nature of the data. Thus, different portions of data of the same size can generate different response times. This is shown in the right part of Figure 1, which presents the execution of a Mandelbrot fractal computation in two devices.

Each device varies the time to compute each pixel, due to the computational region of the Mandelbrot function, with the darkest and reddest areas being the most computationally intensive data regions. Performing the same static balancing as in the regular case, it coincided that the most computationally heavy regions have fallen on the CPU (slower device). This resulted in a significant imbalance with the CPU taking 3.5 s, while the GPU took only 1.2 s. This situation can only be addressed with dynamic balancing algorithms that allocate portions of work to the devices on demand.

This paper addresses both of these problems. On the one hand, we propose to provide oneAPI with mechanisms that allow the implementation of co-execution without additional effort for the programmer. On the other hand, we provide the oneAPI scheduler with a set of dynamic load balancing algorithms to squeeze the maximum performance out of the heterogeneous system, even with irregular applications.

4. OneAPI Coexecutor Runtime

The approach to achieve co-execution focuses on using the DPC++ compiler and runtime, hereafter referred to as oneAPI for simplicity.

Co-execution allows to exploit data-parallel applications with multiple devices simultaneously. There are essentially two different approaches to provide co-execution to oneAPI. On the one hand is adapting and manipulating the compilation process and runtime recognition to force co-execution of the same problem by different devices. In this way, the primitives and methods of the API itself can be extended to facilitate the recognition of working distributions. This approach is discarded in this paper because it requires the transformation of the compilation engine and runtime, as well as their different highly optimized and interconnected stages.

On the other hand, the chosen approach proposed in this paper. The proposed *Coexecutor Runtime* is built on top of oneAPI as a runtime library to allow the parallel exploitation of the CPU along with multiple hardware accelerators that facilitate the implementation of workload balancing algorithms.

This approach has several architectural and adaptive advantages. First, the design and implementation are based on open standards, both C++ and SYCL, following easily recognizable architectural patterns. Secondly, since it is drawing on previous standards, such as OpenCL, this facilitates the adaptation for a whole repertoire of libraries and software generated over a decade, helping to benefit from co-execution. Thirdly, it serves as a skeleton upon which to apply different strategies and workload balancing algorithms for using oneAPI and SYCL. Finally, as it is designed from a sufficiently standardized and abstract approach, it allows the adaptation and extension to execution engines and proposals created by other manufacturers, both compilers and accelerator drivers.

Therefore, to provide oneAPI with co-execution depends mainly on the correct detection of a potential concurrent execution path by the compiler and the runtime. This materializes a parallel execution of several tasks of the DAG, due to the existence of completely independent hardware resources. In this way, the proposal is flexible enough to adapt to a variable number of computation entities, while remaining simple enough to assist the compiler in the detection, favoring the creation of totally independent nodes recognizable by the runtime.

As shown in Section 3, this work addresses the limitations of oneAPI, taking into account three challenges in obtaining effective co-execution. First, considering the static strategies promoted by oneAPI, determining their advantages but also their limitations. Subsequently, the application of dynamic mechanisms is more sophisticated and complex but allowing sufficient adaptability and efficiency to facilitate more elaborated workload distribution strategies. To achieve good performance results, it is necessary to use dynamic strategies, which are currently not available in oneAPI. Finally, various well known load balancing algorithms are incorporated and adapted to this architecture with the purpose of obtaining the best possible efficiency.

4.1. Static Co-Execution

The SYCL standard does not determine the behavior in the face of different computational regions used by independent devices, but the DPC++ implementation of the standard is not able to guarantee simultaneous execution. The main problem arises in the detection of disjointed memory regions when the same data of a problem is used by many oneAPI scopes. This problem occurs when using both independent and shared kernel execution regions, even though the programmer is able to recognize the independence between the execution and data spaces.

For this reason, it is necessary to provide an architecture that facilitates the recognition and management of the system devices, as well as their transfer and computation regions, part of the oneAPI command queues and scopes. The main conceptual idea is to provide with a multi-threaded architecture that isolates every oneAPI scope and, therefore, each device used in the computation. This allows the underlying compiler and DPC++ runtime to recognize the disjoint spaces and be able to perform operations simultaneously. Since this is a static scheduling and workload distribution approach, it is necessary to establish a runtime layer that is as light as possible, reducing the management overhead, because it only schedules one work package per device.

The static co-execution mechanism in the *Coexecutor Runtime* ensures that there is simultaneous execution among the devices, while reducing runtime management operations by not requiring dynamic scheduling operations. Therefore, an *asynchronous pattern* is used to isolate each scope of OneAPI. C++ futures, and its asynchronous mechanisms facilitate an acceptable degree of usability without the need to complicate the management code to solve co-execution problems. This solution favors the independency of regions and captures in a lambda region that makes the offload to another device asynchronous. Furthermore, due to being included in a bigger scope (parent scope of the lambda scope), it takes advantage of the primitives and variables previously initialized, reducing the time and favoring reuse. This is a key aspect if the programmer is using USM, since it allows reusing the memory regions directly, giving direct access to the original queue at any time.

4.2. Dynamic Co-Execution

The dynamic co-execution is based on a generalization of the asynchronous pattern presented in the previous section. As a result of working with runtimes employed on a variety of vendors (drivers and devices), high efficiencies have been contrasted by using strategies based on event chaining and multithreading architectures with mixed management: notifications based on callbacks and workload management threads [14,17,28]. The main problem when trying to extrapolate these strategies, based on events, futures, and C++ asynchronicity, is the limitation of expressiveness in the iterative distribution of workloads, complicating and preventing dynamic strategies and disabling all usability.

Moreover, the most similar mechanisms to use notifications based on callbacks are not supported in OneAPI, and thus it would be necessary to lower the level of abstraction and use strategies combined with OpenCL. However, these mechanisms were discarded by design as it would affect the resulting productivity and the benefits of single-source property. On the other hand, although callback based mechanisms have become efficient in several architectures, there are others for which they are prohibitive or directly impossible to use, as is the case of FPGAs [30].

The strategy proposed for dynamic co-execution is to promote multithreaded management architectures based on the runtime of oneAPI. The *Coexecutor Runtime* enhances the isolation between devices, since one of the key points is to make it easier for the compiler to detect disjoint memory structures as well as the independence between queues and tasks. In addition, since oneAPI offers a sufficiently sophisticated and complete memory model, the management architecture must be adapted to favor both buffer management and the possibility of exploiting unified memory (USM).

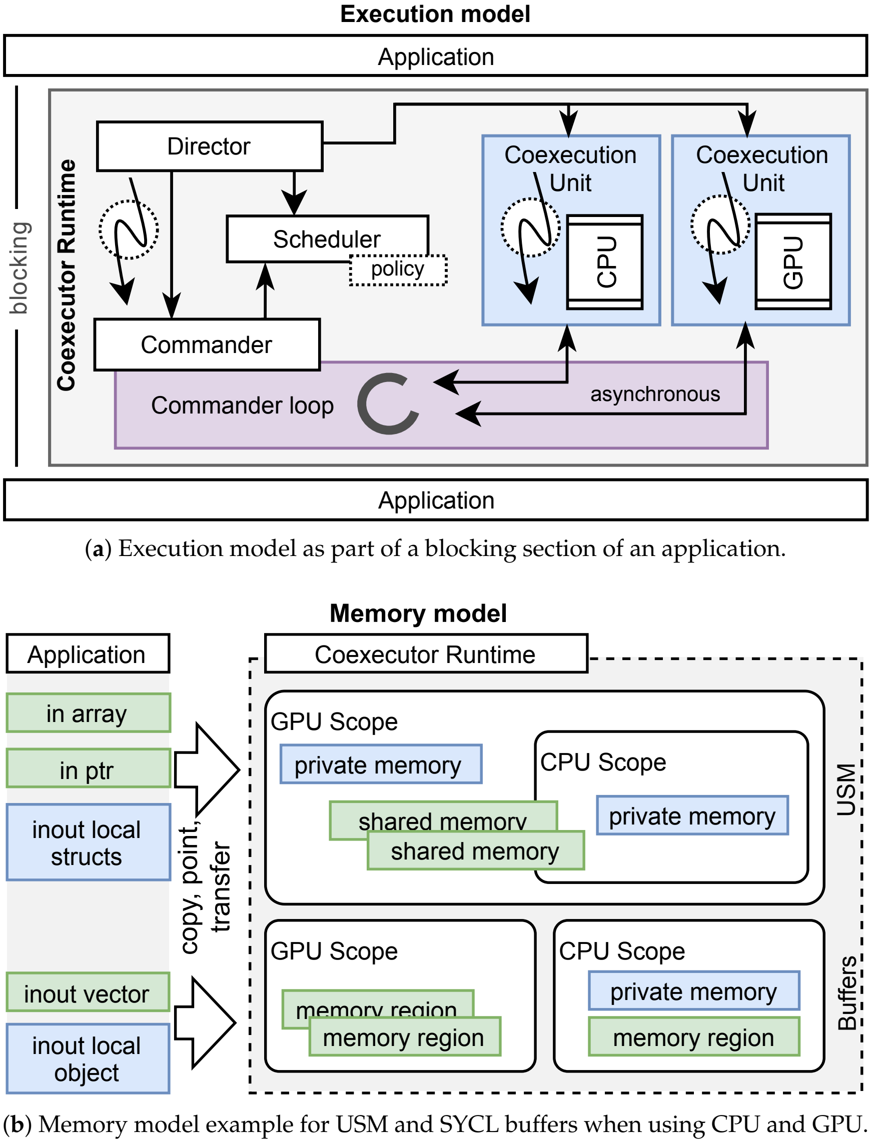
To define the proposal, three perspectives are considered, the execution model, from the memory point of view, and the last one, the relationship of the *Coexecutor Runtime* with the runtime of oneAPI, as explained in Section 4.3.

The *execution model* is shown in Figure 2a, representing the interaction of the runtime as part of the execution process of an application. Execution is blocked from an application point of view, although internally it works asynchronously. In this way, the programmer only has to wait to finish the computation taking advantage of all its devices. However, there is the possibility to continue extending operations to be run on devices from the application side since the task graph is managed by the runtime of oneAPI. The *Coexecutor Runtime* is in charge of the creation and control of management threads (curved arrows), which will be part of the operation mechanisms of the co-execution architecture until the control is returned to the application. There is a thread belonging to the main manager, with its core component called the *Director*. In addition, there are lightweight management threads, termed *Coexecution Units*, requiring one per computing device.

The *Director* configures the *Coexecution Units* and manages both the *Commander* and its communication with the rest of the entities. The *Scheduler* is instantiated and plugged in with a policy established by the programmer, using one of the schedulers explained in Section 4.3. The *Commander* is responsible for packaging the work, emitting tasks, and receiving events, as part of the computation workflow with the *Coexecution Units*. This process is termed as the *Commander loop*, and it follows the scheduling strategy defined by the *Scheduler*.

Regarding the *Coexecutor Runtime* internal workflow, the *Director* instantiates and configures oneAPI primitives and structures necessary both for the operation with oneAPI runtime and used by the *Scheduler* itself. Among these are work and queue entities, execution contexts, and mapping of memory structures between the application and the runtime. These oneAPI structures and instantiations are shared by the components of the *Co-execution Runtime*, favoring the reuse and detection of data types by the oneAPI runtime. In parallel, the management threads of the *Coexecution Units* initialize the communication mechanisms within the runtime, as well as the request of devices and their configuration with oneAPI.

The communication is bidirectional between *Commander* and each *Coexecution Unit*, since it is co-executed with an independent scheduler that handles the decisions. As soon as there is a *Coexecution Unit* ready to receive work and the management thread has finished the initial phase, it establishes communication with the *Commander loop*. As the rest of the devices are completing their initialization, they incorporate into the loop where the scheduling phase starts.



(**a**) Execution model as part of a blocking section of an application.

(**b**) Memory model example for USM and SYCL buffers when using CPU and GPU.

Figure 2. Coexecutor runtime considering an example of CPU-GPU dynamic co-execution.

In multi-core CPUs, where oversubscription has a significant impact, it might be convenient to disable the *Director* management thread via its configurable behavior and merge its management as part of the CPU Coexecution Unit. This is a compromise in terms of the oversubscription overhead and runtime acceleration when overlapping Commander tasks with the computation or communication of Coexecution Units.

The *memory model* is presented in Figure 2b. This shows the separation between structures and memory containers, taking into account the two types of strategies used: USM or buffers of SYCL, although the *Coexecutor Runtime* supports the combination of both during the co-execution. On the left side are shown the structures, C++ containers, and memory pointers used by the application, while the right side outlines the view of the runtime. The *Director* and its *Coexecution Units* handle the allocations and configuration of the memory space with oneAPI, and the programmer only has to request the use. The runtime will distribute them in the oneAPI memory model, either by transferring pointers, copying memory regions or by sharing unified memory blocks.

Since the co-execution proposal was designed and evaluated to run on host and accelerator devices, two operating modes of the runtime are distinguished regarding OneAPI memory environments. If USM is used, the *Coexecutor Runtime* provides two scopes: a larger one for a device (GPU) and a smaller for another (CPU). In this way, the memory spaces initialized by the GPU are reused in the CPU using oneAPI primitives. On the other hand, if SYCL buffers are used, the scope of each device will manage independent buffers with memory regions that will be part of a higher container or structure, therefore, favoring the recognition of disjointed data spaces by the compiler. Private memory allocations can be made in both memory models, in the form of buffers and variables, where each field is controlled independently by each *Coexecution Unit* and its oneAPI scopes.

Finally, both ways of operating can be combined, due to the regions of the kernel that use the USM model and others that rely on buffers and variables. *Coexecutor Runtime* will reuse the scope of each device to map any C++ containers and memory regions, each of which will be governed by a memory model.

The interaction between the *Coexecutor Runtime* and oneAPI is shown in Figure 3. Three stages are presented during the execution of the runtime, with two different queues Q1 and Q2. It starts from a situation where the runtime has established two independent parallel execution queues, due to the existence of two separate underlying architectures. The nodes of each queue are managed by the runtime through the DAG, and they can be in three different states: execution (blue), blocked waiting for resources (white), or finished (gray with a dashed line). The *Director* waits for events related to the DAG or performs independent tasks, such as resource management, receiving and sending notifications, status control, or work reparation, some of which are essential within the *Scheduler*.

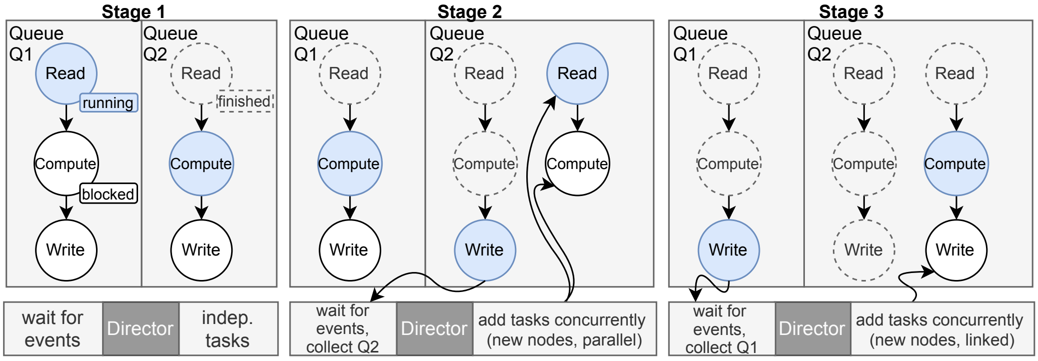


Figure 3. Example of interaction with the DAG from oneAPI's perspective while running a dynamic approach with two queues.

By switching to stage 2, we can distinguish how Q2 is able to process nodes more efficiently, so the *Director* collects results of the write operation and enqueues new nodes of the DAG to the same queue, overlapping computation and communication. Collection operations are dependent on the memory model, the type of operations (explicit or implicit) and the amount of bytes used; thus, they could be fast, as in unified memory, or slow, when using mixed models or while transferring large blocks. Finally, in stage 3, the end of Q1 is represented with the output data collection while, in Q2, a next writing task is added. This is linked to the branch created in stage 2 as soon as its computation task has started, thereby, distributing the DAG management among different time periods.

4.3. Load Balancing Algorithms

Load balancing algorithms make it possible to distribute the work in different ways among the devices that compose the heterogeneous system. Each algorithm offers different behaviors that determine the efficiency when computing a problem, influencing the throughput, the synchronization penalties between host and devices, as well as the efficiency of the balancing, as shown in Section 6. There are static and dynamic algorithms; thus, it is necessary to provide a runtime flexible enough to accommodate such modes of operation in an efficient manner.

In order to generalize and standardize the architecture, the two previous approaches, static and dynamic co-execution designs, are unified in favor of the latter. To do this, a comparison is made between the two when it comes to launching a package for each device, as a static offline scheduling. As no overheads have been found due to the additional management in the dynamic strategy with respect to the static one, we decided to discard the latter due to its disadvantages and impossibility of implementing decisions dynamically.

In addition to the advantages described in the Section 4.2, the *Coexecutor Runtime* architecture offers an efficient scheduling module (*Scheduler*) that allows the dynamic strategies to be exploited easily, with a common internal scheduling interface and offering negligible synchronization overheads. However, it increases the internal complexity of the runtime, albeit it is hidden to the programmer, who receives a simple and straightforward API to operate with. Nevertheless, the dynamic co-execution design is sophisticated enough to use any static co-execution algorithm with no efficiency penalties as shown in Section 6. Moreover, the runtime excels in solving the main problem exposed in Section 3 due to the performance achieved when using dynamic load balancing algorithms.

To enable dynamic policies to squeeze all the computing capacity out of the heterogeneous system, the *Scheduler* component is introduced as shown in Figure 2a. It configures the behavior of the load balancer, the distribution and division of the work packages (amount and region of data to be computed by each device), as well as the way to communicate with the different execution devices.

Figure 4 depicts the relationship of the *Coexecutor Runtime* with the runtime of oneAPI, all involved as part of the *Commander loop*. The *Coexecutor Runtime* internal communication is performed between the management threads, either those associated with the devices (right) or the global manager, usually associated with the *Director* (left). This view simplifies the runtime of oneAPI and its internal DAG management, being considered as a single entity, as part of the *Coexecution Units* (right). The *Director* performs a set of periodic actions, such as loop managing events and operations, including preparing the next job to be issued; collecting completed jobs; updating pending jobs; managing the end of a work block and its completion; preparing and reusing the queue and command groups as well as other oneAPI primitives; and updating the indexes, ranges, and offsets of memory entities, as well as variables and containers.

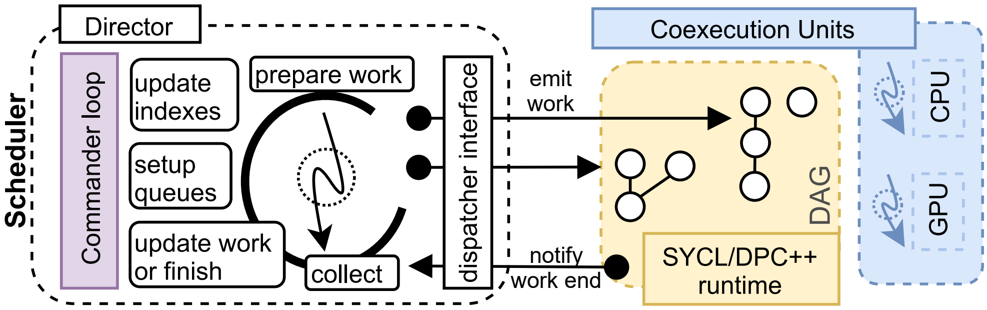


Figure 4. A Commander's loop where the scheduling strategy is performed to coordinate the behaviors of the Coexecution Units.

Every time a work package is prepared, the runtime adds a task in the DAG. Similarly, with the completion of a job, the *Commander* receives the notification to collect and merge the output data, if needed. This operation can be lightweight in the case of using USM or using implicit operations, delegating more responsibility to oneAPI. The emission and reception of work is requested through a dispatch interface, as a way of unifying requests. Finally, when there are no more pending jobs, the *Commander* will notify the *Director* to close and destroy the primitives and management objects to return control to the application.

Based on the proposed architecture and its designed dynamic co-execution model, three algorithms are implemented in the *Scheduler* [11,14,28]. The *Static* algorithm divides the kernel in as many packages as devices are in the system, minimizing the number of host–device interactions. The size of each package is proportional to the relative computing speed of each device. Its drawbacks are that it is difficult to find a suitable division and that cannot adapt its behavior dynamically to irregular applications. It has a minimum management inside the *Scheduler* component because it only runs as many iterations in the loop of events as devices are co-executing.

Regarding the strictly dynamic strategies, the *Dynamic* algorithm divides the data in a number of packages prefixed by the programmer and all of them of a similar size, which are assigned to the devices on demand, as soon as they are idle. This allows it to adapt to the irregular behavior, but increases the overhead communication between host and devices. On the other side, *HGuided* starts with large packages and decreases their size as the execution progresses. The size of the initial packages is proportional to the computing capacity of the devices. Therefore, it reduces the number of synchronization points while retaining most of its adaptiveness.

Considering these dynamic policies inside the *Scheduler*, it is not possible to know in advance the quantity of iterations, because it will depend on the execution parameters, as well as the number and type of devices. These operations increase the management overhead due to the operations related to the update of indexes and ranges, as well as the division of the problem into independent regions. Finally, concerning the differences in the operations carried out by the *Commander*, *Dynamic* will simplify the number of instructions involved in the calculation of work packages compared to *HGuided*. This is explained since the latter performs a more sophisticated algorithm that takes into account certain conditions, including the computing power of each device. However, the calculation overheads of the latter are compensated by the efficiency of its workload distribution policy.

4.4. API Design

The *Coexecutor Runtime* was designed to offer an API that is flexible as well as closely linked to the SYCL standard, favoring the reuse of existing code and a slightly higher usability. Furthermore, it offers two modes of computation from the point of view of the programmers. The results of architectural and design decisions concerning expressiveness and usability enhancement are materialized in both modes, providing distinct usage facilities. The same SAXPY kernel is computed using both modes to compare its differences. Both code snippets show the runtime usage from the perspective of the programmer, thus, omitting the initialization of the problem and its data as well as the subsequent use of the results.

The simple mode is shown in Listing 1, where an explicit embedded context (lambda function) is used to perform the computation in a few lines of code. This shows an example of use when computing the SAXPY problem simultaneously exploiting two different devices, being, in this case, the CPU and GPU. Line 1 instantiates the `coexecutor_runtime` prepared to compute a program using the HGuided balancing algorithm. In the next line, it is configured to use both the CPU and GPU, giving a hint of the computational power of 35% for the CPU compared to the GPU. This value will leverage the algorithm to further exploit the co-execution efficiency.

Next, the co-execution scope associated with the problem is provided (lines 3–14), where a lambda function captures the values used by reference. This scope is executed by each of the *Coexecution Units*, and therefore they must establish independent memory reservations (or shared, if unified shared memory is exploited), using the values provided by the runtime itself through the *package* class. Line 6 opens an execution scope that is associated to the kernel computation for each device. In lines 7 and 8, a read and write access is requested for the previous memory region (buffer *accessors*), indicating the execution space based on the given package size. Finally, lines 9 to 11 show the data-parallel execution, traversing the execution space (*R*) and using the *accessors* and variables needed (*datav, input*).

Listing 1. Coexecutor Runtime computing SAXPY with a dynamic algorithm using simultaneously CPU and GPU.

```
1  coexecutor_runtime<hg> runtime;
2  runtime.config(CounitSet::CpuGpu, coexecutor_runtime::dist(0.35));
3  runtime.launch(data.size(), [&](coexecutor_unit *counit, package pkg) {
4    sycl::buffer<int, 1> buf_input(data.data() + pkg.offset,
5                                   sycl::range<1>(pkg.size));
6    counit->dispatch([&](sycl::handler &h) {
7      auto R = sycl::range<1>(pkg.size);
8      auto input = buf_input.get_access<sycl::access::mode::read_write>(h);
9      h.parallel_for(R, [=](sycl::item<1> it) {
10       auto tid = it.get_linear_id();
11       input[tid] = input[tid] * datav;
12     });
13   });
14 });
```

After executing the lines shown in the code snippet, the problem is computed simultaneously using both devices. The `launch` call blocks the program execution since it is a synchronous operation from the point of view of the programmer. Therefore, the data resulting from the computation is in the expected data structures and containers that the programmer uses in the C++ program (vector *input* in the example) without the need to create any new buffer exchange structures specific for the coexecution.

On the other side, the extended computation mode offers more flexibility to the runtime and the kernel computation. It allows accessing extended methods and callback functions, based on the `CommanderKernel` interface provided by *Coexecutor Runtime* as is implemented in Listing 2 and instantiated and executed in Listing 3. The advantages of this computation mode are the encapsulation of the program, increasing the maintainability, and the enhanced flexibility regarding runtime operations.

Some of these allow the programmer to share buffers between devices (lines 8–10) and perform custom operations at specific places during the *Coexecutor Runtime* execution process, such as during the initialization and completion of the *Director* scope (lines 8 and 11), during the initialization or completion of the *Coexecution Units* or at the end of every work package completed (lines 36–40).

The drawbacks are increased complexity and verbosity compared with the simple computation mode; however, the latter is only recommended for smaller and easier kernel algorithms. Moreover, in this example, the SAXPY computation is performed using a dynamic load balancing algorithm (`dyn`), splitting the workload into 128 packages, scheduled at the runtime among the devices (lines 3–4 of Listing 3).

Finally, *Coexecutor Runtime* allows using oneAPI features and extensions, integrated as part of the architecture of the runtime, presented in Section 4, or exposed via API calls to the programmer, such as using unified shared memory by calling the `alloc` and `free` methods of the runtime (shown in lines 7 and 14 of Listing 3). Taking into account the runtime design principles as outlined in these two examples, the *Coexecutor Runtime* hides all the implementation details, easing the use of its co-execution capabilities to exploit easily any oneAPI program.

Listing 2. SAXPY program definition using the `CommanderKernel` interface provided by Coexecutor Runtime to implement the kernel behavior as an independent unit.

```
class SAXPY : public CommanderKernel {
public:
 SAXPY(int *x_ptr, int *y_ptr, int *out_ptr, size_t data_l, float sc_fl)
     : m_x_ptr(x_ptr), m_y_ptr(y_ptr), m_out_ptr(out_ptr),
       m_sc(sc_fl), m_data_l(data_l){}

 // pre-setup config: shared internally between Coexecution Units
 void init(coexecutor_unit *counit){  // Director Scope
   counit->add_buffer<int, 1>(0, m_x_ptr, sycl::range<1>(m_data_l));
 }
 // void init_completed() {} /* @callback */
 program_size size() { return m_data_l; }

 void compute(coexecutor_unit *counit, package pkg){
   std::cout << "[" << counit->id() << "] computing...\n";
   sycl::buffer<int, 1> buf_x =
       *counit->get_buffer<int, 1>(0); // use all buffer
   sycl::buffer<int, 1> buf_y(m_y_ptr+pkg.offset, sycl::range<1>(pkg.size));
   sycl::buffer<int, 1> buf_out(m_out_ptr+pkg.offset,
                                 sycl::range<1>(pkg.size));
   // communicate with the Commander to dispatch a transaction
   counit->dispatch([&](sycl::handler &h){ // Director-CoExecUnit comm.
     auto R = sycl::range<1>(pkg.size);
     auto x = buf_x.get_access<sycl::access::mode::read>(h);
     auto y = buf_y.get_access<sycl::access::mode::read>(h);
     auto out = buf_out.get_access<sycl::access::mode::discard_write>(h);
     auto sc = (int)m_sc; // caching
     h.parallel_for(R, [=](sycl::item<1> it){
       auto tid = it.get_linear_id();
       out[tid] = x[tid] * sc + y[tid];
     });
   });
 }
 /* @callback */
 void compute_completed(coexecutor_unit *counit, package pkg){
   std::cout << "[" << counit->id() << "] package" << pkg.id
             << "computed with throughput" << pkg.throughput << "\n";
   counit->dump_statistics(); // Stats per package - CoExecUnit related
 }
private: // organized data:
int *m_x_ptr; int *m_y_ptr; int *m_out_ptr; size_t m_data_l; float m_sc;
};
```

Listing 3. Coexecutor Runtime using the CPU and GPU simultaneously to compute the SAXPY kernel definition of the Listing 2. This example shows the exploitation of the extended computation mode to enhance the flexibility of the runtime.

```
auto N = pow(10,8); std::vector<int> x; int* y; std::vector<int> out(N);

coexecutor_runtime<dyn> runtime;
runtime.config(CounitSet::CpuGpu, 128); // dynamic; 128 packages

x.assign(N, 1);
y = runtime.alloc<int>(N);
// more assigns: foreach, assign x[i], y[i] ...

SAXPY program{x.data(), y, out.data(), N, 3.14159};
runtime.run(program);
// from here on, we have the data in our C++ containers

runtime.free<int>(y);
```

5. Methodology

The experiments to validate the *Coexecutor Runtime* [29] (https://github.com/oneAPI-scheduling/CoexecutorRuntime, accessed on 28 September 2021) were carried out in two nodes, labeled *Desktop* and *DevCloud*. Desktop was a computer with an Intel Core i5-7500 Kaby Lake architecture processor, with four cores at 3400 MHz, one thread per core and three cache levels. Kaby Lake's on-chip GPU was an Intel HD Graphics 630, a mid-range (GT2) IGP integrated graphics processor, member of the family of Gen 9.5 GT2 IGP, with 24 execution units running between 350 and 1100 MHz. An LLC cache of 6 MB was shared between the CPU and GPU.

On the other side, DevCloud was an Intel server node with an Intel Xeon E-2176G processor, with 12 logical cores at 3700 MHz, two threads per physical core, and three cache levels. Coffee Lake's on-chip GPU was an Intel UHD Graphics P630 with 24 execution units running up to 1200 MHz and sharing a 12 MiB LLC cache with the CPU.

To accomplish the validation, six benchmarks were selected to represent both regular and irregular behavior. A kernel is defined as regular when different packages of the same size take up the same execution time on a particular device and irregular in the opposite case. *Gaussian*, *MatMul* and *Taylor* correspond to regular kernels, while *Mandelbrot*, *Rap* and *Ray Tracing* are the irregular ones. Additionally, *NBody* is included as a special case, since it exposes a peculiar dynamic behavior to be detailed in Section 6.4. Taylor, Rap, and Ray Tracing are open source implementations, while the rest belong to the AMD APP SDK, all ported to oneAPI. Table 1 presents the most relevant parameters of the benchmarks, providing sufficient variety to validate the behavior of the runtime.

Table 1. Benchmarks and their properties.

Property	Gauss	Matmul	Taylor	Ray	Rap	Mandel	*NBody*
Local Work Size	128	1.64	64	128	128	256	64
Read:Write buffers	2:1	2:1	3:2	1:1	2:1	0:1	2:2
Kernel args	6	5	7	11	4	8	7
Use local memory	no	yes	yes	yes	no	no	no
Work-items (N $\times 10^5$)	262	237	10	94	5	703	4
Mem. usage (MiB)	195	264	46	35	6	1072	26
Use custom types	no	no	no	yes	no	no	no

The validation of the proposal was conducted by analyzing the co-execution when using four scheduling configurations in the heterogeneous system. As summarized in Section 4.3, Static, Dynamic, and HGuided algorithms were evaluated, labeled as St, Dy, and Hg, respectively. In addition, the dynamic scheduler was configured to run with 5 and 200 packages. Finally, two different memory models were also tested: unified shared memory (USM) and SYCL's buffers (Buffers).

To guarantee the integrity of the results, 50 executions were performed per case with an initial execution discarded to avoid warm-up penalties. Since the GPU is on-chip and there are hardware policies regarding frequency throttling due to temperature thresholds, some decisions were applied to stabilize the results. First, CPU turboboost was disabled, and the CPU governor was set to performance. The CPU and GPU were set to 2400 and 600 Mhz fixed frequencies, respectively. Finally, every execution started when the CPU socket temperature was under 38°. In DevCloud, where some of the previous conditions could not be applied, waiting times were introduced between executions, increasing the duration of the experiments but stabilizing their measurements and results. The standard deviation is not shown because it was negligible in all cases.

To evaluate the performance of the *Co-executor Runtime* and its load balancing algorithms, the total response time, as well as the response time of each of the devices, were measured, including kernel computing and data transfer. Then, three metrics were calculated: balancing efficiency, speedup, and heterogeneous efficiency. The former measures the effectiveness of the load balancing, calculated as $\frac{T_{GPU}}{T_{CPU}}$, where T_{GPU} and T_{CPU} are the execution time of each device. Thus, the optimum is 1.0, meaning both devices finish simultaneously without idle times. The speedup is computed as $S = \frac{T_{GPU}}{T_{co-exec}}$, because the GPU is the fastest device for all the benchmarks. Due to the heterogeneity of the system and the different behaviors of the benchmarks, the maximum achievable speedups depend on each program. These values are computed as:

$$S_{max} = \frac{1}{max_{i=1}^{n}\{T_i\}} \sum_{i=1}^{n} T_i$$

Additionally, the heterogeneous efficiency was computed as the ratio between the empirically obtained speedup and the maximum achievable speedup for each benchmark. $HE = \frac{S}{S_{max}}$ [31]. Furthermore, a scalability analysis of the co-execution with respect to the problem size is presented. To do this, the total execution time of the heterogeneous system using co-execution and of each of the individual devices was measured, increasing the size of the problems.

Finally, two metrics were used to assess the behavior of co-execution with respect to energy. The energy consumption of the whole system was measured using RALP counters, providing measurements in Joules. On the other hand, the energy efficiency was calculated using the Energy-Delay Product metric, which, as its name suggests, is obtained by multiplying the energy consumed by the response time. Since the values obtained have a very wide range, this metric is provided normalized with respect to the EDP of the only-GPU execution.

6. Validation

This section presents the experimental results carried out to evaluate performance and energy when co-executing, materialized with three scheduling policies. A scalability analysis was also performed for the CPU, GPU and the best scheduling algorithm. Finally, a special case is detailed when using dynamic approaches for the NBody benchmark.

6.1. Performance

As explained above, three different metrics were used to assess the performance of co-execution. They were the balancing efficiency, speedup, and heterogeneous efficiency achieved by co-execution in the CPU-GPU system, with respect to the only-GPU execution. The values measured in the experiments for the two different architectures evaluated, Desktop and DevCloud, are shown in Figures 5–7, respectively.

The abscissa axes show the benchmarks, each one with four scheduling policies and two memory models, as defined in Section 5. Moreover, the geometric mean for each scheduling policy is shown on the right side (average). The main conclusion that is important to highlight is that co-execution is always profitable from a performance point of view, as long as it is done with dynamic schedulers, and even more if using unified memory (*USM*), as the geometric mean summarizes for these benchmarks and scheduling configurations.

Regarding the balancing efficiency, the optimal is 1.0, where both devices finish simultaneously without idle times. Any deviation from that value means more time to complete for one device compared with the other. Generally, the imbalance is below 1.0 due to the overheads introduced by the CPU because it has to process part of the workload as a device, but also to manage the *Coexecutor Runtime*, as the host. In Desktop, it rarely completes its computation workload before the GPU finishes, since the latter requires more resource management by the host, increasing the CPU load. As the number of cores in the system increases, the management cost of the runtime is reduced, benefiting the CPU and requiring more work than is allocated to it. This behavior can be appreciated when using the DevCloud node, with 12 logical cores, compared to the 4 in Desktop.

Speedup allows assessing how much faster co-execution is compared to GPU-only execution, while heterogeneous efficiency helps understand how well the whole system is being utilized. Therefore, the latter metric allows comparing performance on both architectures, which cannot be done with speedup, as speedup in a heterogeneous system is always relative to the computational power of each device.

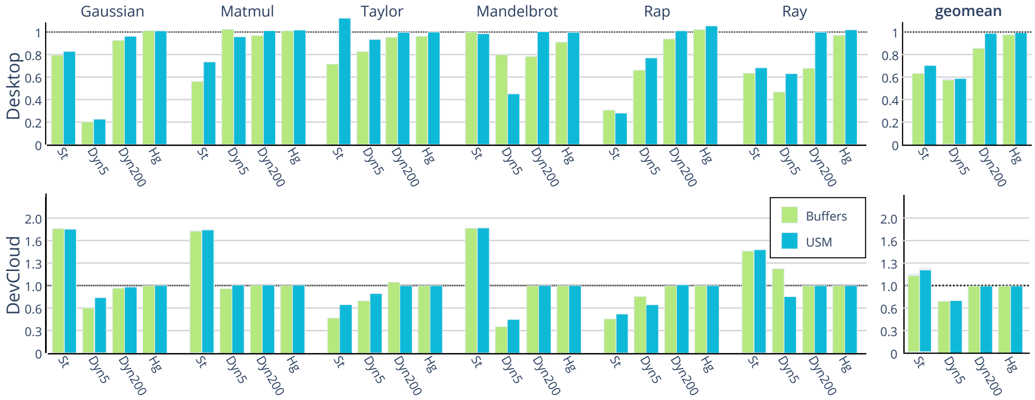


Figure 5. The balancing efficiency for a set of benchmarks when doing CPU-GPU co-execution in Desktop and DevCloud nodes.

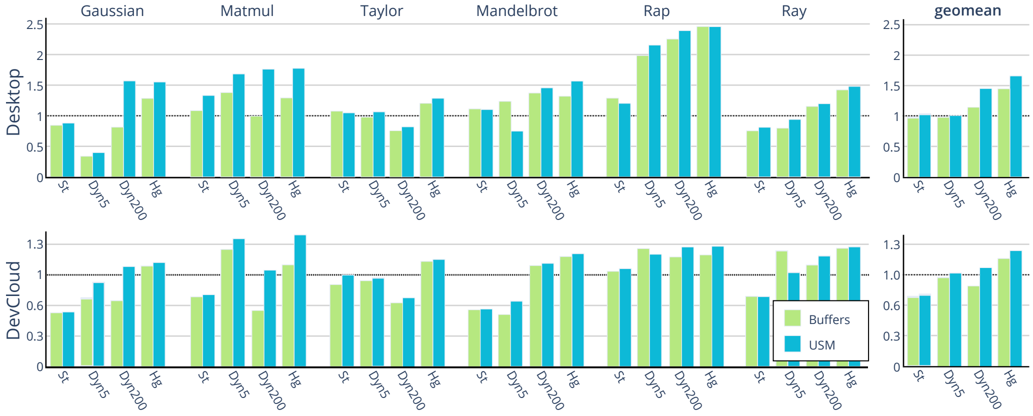


Figure 6. Speedups for a set of benchmarks when doing CPU-GPU co-execution in Desktop and DevCloud nodes.

Analyzing the different load balancing algorithms, it can be seen that the Static offers the worst performance, even in regular applications where it should excel. This is because the initial communication overhead caused by sending a large work package, leads to a significant delay at the beginning of the execution, strongly penalizing the final performance. As can be seen later in the Section 6.3, even in a single, full kernel offload on GPU, considerable CPU (host) management is required.

Therefore, if CPU intensive is being performed without the possibility of alternating runtime management with kernel computation, as is the case with static co-execution, the whole system will generally be penalized. In theory, this algorithm should cause less communication and synchronization overhead. However, it fails to balance the workload properly, as can be seen in Figure 5, resulting in very low speedups, with averages below or equal to 1.0 in both architectures, which means that co-execution is not profitable.

It is important to note that static algorithms and, in general, all those with low load balancing efficiency, such as those that distribute few packages among the devices (Dyn5), offer worse performance (and energy consumption, as it will be shown) with oneAPI. This is because Coexecutor Runtime and oneAPI offer excellent performance in workload management when dynamic scheduling policies are exploited, even when there are many or small packages, two cases that penalize greatly in other runtimes and technologies [14,17,28].

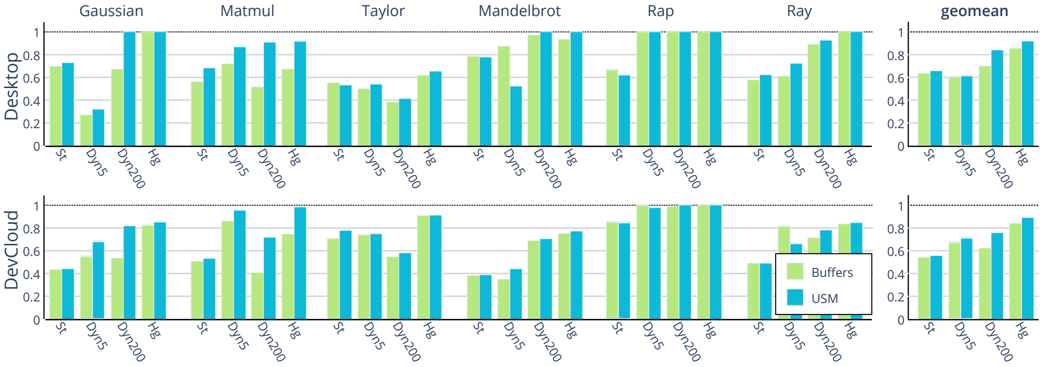


Figure 7. Efficiency for a set of benchmarks when doing CPU-GPU co-execution in Desktop and DevCloud nodes.

Regarding dynamic algorithms, they provide good results in general, especially when the *USM* memory model is used. However, they have the drawback that the number of packages for each benchmark has to be carefully selected. A very small number of packages can lead to higher imbalances causing a performance penalty, as can be seen in Gaussian, Mandelbrot or Ray, in the case of Dyn5. At the other extreme, a very large number of packages increases the communication overhead, negatively impacting performance, as in Gaussian, Taylor, and MatMul with *Buffers*.

In between, there is a tendency that the greater the number of packages, the better the balancing. Moreover, in such cases with good balancing efficiency, it achieves better performance, especially if USM is used. This is an expected behavior because the packages are smaller, and their computation is faster, giving less chance of imbalance in the completion of both devices. This is an interesting behavior since the *Coexecutor Runtime* is delivering high performance when using dynamic strategies due to the low overhead of the *Commander loop* when managing packages and events. This behavior is also found and detailed in the special case of NBody, explained later in Section 6.4.

The HGuided algorithm offers the best scheduling policy, both for regular and irregular kernels, and for the two architectures evaluated as Desktop and DevCloud. This result is achieved due to a combination of two properties. On the one hand, its excellent balancing efficiency, that on average is very close to 1, as can be seen in Figure 5. On the other hand, the *Co-executor Runtime* does a great job in overlapping computation and communication, thus minimizing the impact of the synchronization and communication overhead inherent to dynamic algorithms.

For example, HGuided scheduled, on average, 42 and 53 packages for Mandelbrot and Ray, while the best dynamic configuration (*Dyn200*) used 200 packages for both cases. Although the balancing efficiency was good in both scheduling configurations, the reduction of communications and synchronization mechanisms boosted the HGuided strategy compared with the Dynamic one.

HGuided yielded the best performance in all the analyzed benchmarks with average speedup values of 1.65 and 1.26 in the Desktop and DevCloud architectures, respectively. Therefore, the co-execution is able to squeeze the maximum performance out of all the resources available in the system, such as the CPU and GPU shown in this validation, offering an efficiency of 0.92 on Desktop and 0.89 on DevCloud. Moreover, since it is a dynamic algorithm with high balancing efficiency, it does not require any a priori parameters, simplifying the programming effort.

Considering the memory models, there was a general improvement in balancing and performance when using *USM* compared with *Buffers*. It can be observed than *USM* performs much better than *Buffers* on regular kernels and with dynamic strategies; however, this difference practically disappears on irregular kernels.

Finally, it is important to note that there is generally a tendency for good balancing efficiency to be related with increased speedups. However, it is important to qualify this pattern, since the validation presents two different architectures and an interesting variety of benchmarks with distinct properties, as indicated in Section 5. For this reason, there are cases with imbalance issues; however, the total work is completed in less time, because the device that receives more work executes faster.

Thus, it achieves more throughput per packet, fewer synchronizations and communications, as well as less host management overheads, enhancing the total computation. This behavior is observed in MatMul, Ray, and Taylor for dynamic algorithms, specifically in configurations that include too many packages (*Dyn200*), where there is an increase in host–device communications. In such cases, the workload distribution and scheduling management can penalize performance, especially if Buffer-type structures are being used instead of shared memory optimizations (*USM*). This is also the main reason why HGuided excels since it solves these problems.

6.2. Scalability

This section presents a scalability analysis of the *Co-executor Runtime*, varying the size of the problems. To this aim, Figure 8 shows the evolution of the execution time of each benchmark with respect to the size of the problem, in different configurations: CPU-only, GPU-only, and co-executing using HGuided scheduler.

The most important conclusion to be drawn is that, in all the cases studied, there was a turning point from which co-execution improved the performance of the fastest device. For very small problem sizes, the overhead introduced by the *Co-executor Runtime* cannot be compensated by the performance increase provided by the co-execution. These points are more noticeable in Gaussian, Mandelbrot, Ray, and Taylor, because the differences in computing capacity between CPUs and GPUs were much more pronounced (13.5×, 4.8×, 4.6×, and 3.2×, respectively).

Experiments were done considering the memory models, but from a scalability and representation point of view, it is difficult to discern the differences between *USM* and *Buffers*. As reflected in Section 6.1, *USM* offers better efficiencies, and thus it was chosen to show scalability using only this memory optimized mode.

Matmul is a special case since, by increasing the size of the problem, a point is reached where co-execution obtains the same performance as the GPU-only. A detailed analysis of the hardware counters indicates how the LLC memory suffers constant invalidations between CPU and GPU. Temporary locality of the shared memory hierarchy is penalized when co-executing with very large matrices, because the GPU requests memory blocks aggressively.

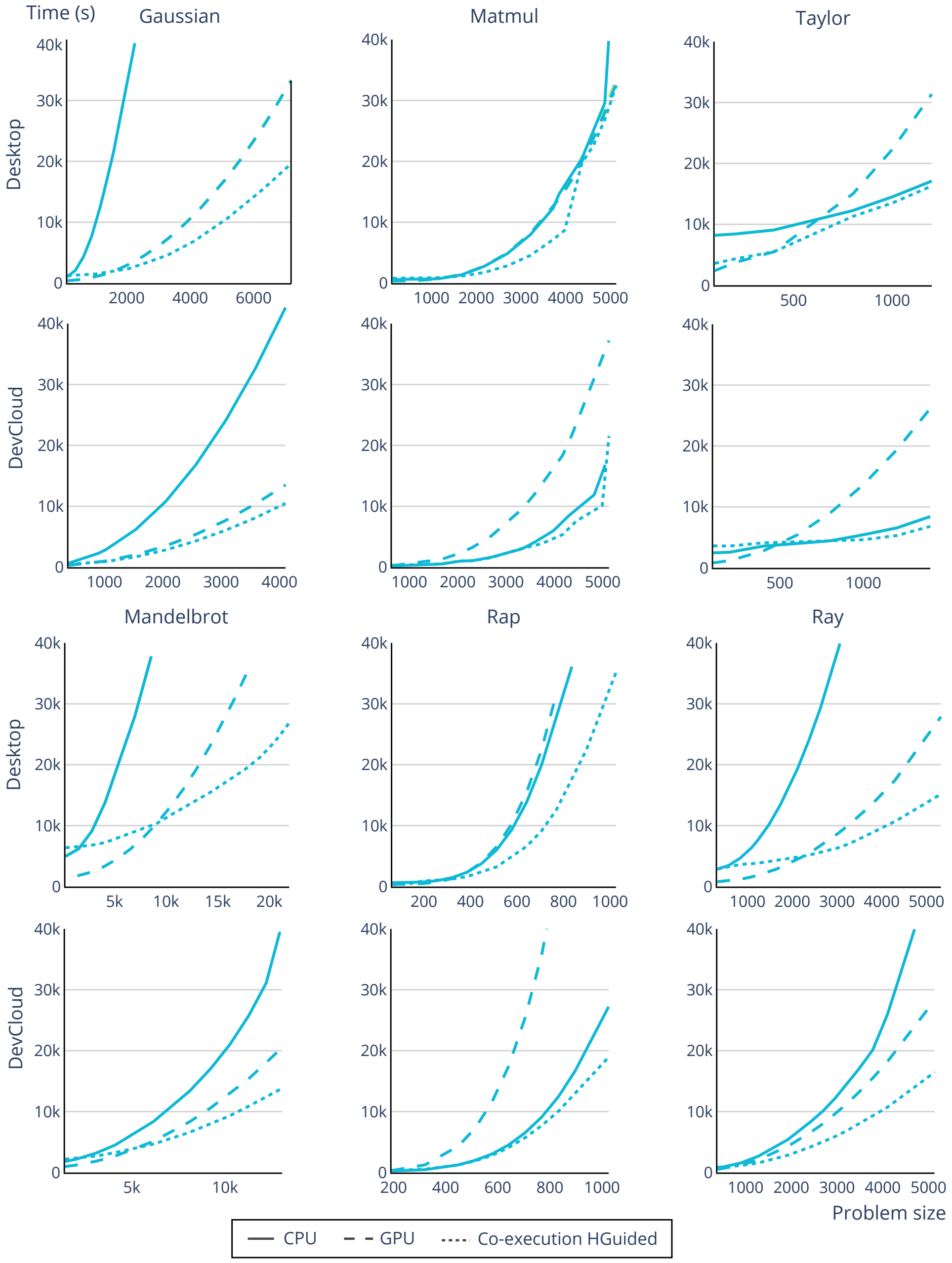


Figure 8. Scalability for CPU, GPU and CPU-GPU coexecution using the Coexecutor runtime with its HGuided scheduling policy and USM memory model for Desktop and DevCloud nodes.

6.3. Energy

This section presents the analysis of the energy consumption as well as the energy efficiency of the Desktop system, both when using only the CPU and the GPU and with different configurations of co-execution runtime. The DevCloud node does not offer any possibility to measure power consumption, such as RAPL counters, perf events, or any Intel system tools. Therefore, this section focuses on analyzing the energy behavior of the Desktop node when using the oneAPI *Coexecutor Runtime*.

Figure 9 presents the energy consumption with each bar composed of up to three regions representing the energy used by: the CPU cores, the GPU, and the rest of the CPU package together with the DRAM (*uncore* + *dram*).

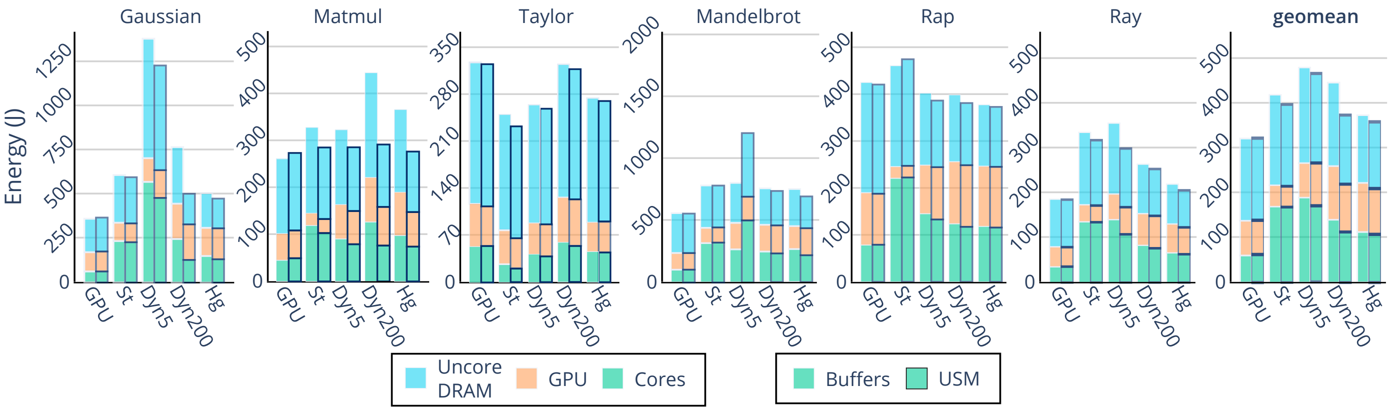


Figure 9. The energy consumption by cores, GPU, and the other units of the package with the DRAM consumption for the Desktop node.

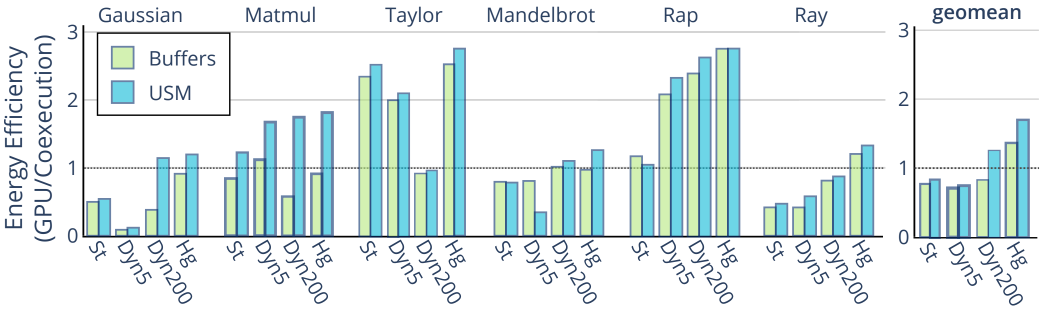


Figure 10. The energy efficiency compared with GPU (more is better) for the Desktop node.

Considering the average energy consumption, using only the GPU is the safest option to ensure minimum energy consumption. This is because the energy savings achieved by the reduction in execution time due to co-execution, is not sufficient to counteract the increase in power consumption caused by the use of CPU cores. However, there are also benchmarks, such as Taylor and Rap, where co-executing does improve the power consumption over GPU, and others where co-execution and GPU-only have similar energy consumption, such as MatMul.

This is because the computational powers between the two devices are considerably close, achieving very balanced distributions that result in efficient energy consumption. Taylor has a computational power ratio of 0.44:0.56 (CPU and GPU), while Rap has a ratio of 0.39:0.61. The case of MatMul is particular as it is due to LLC sharing penalties as is explained in Section 6.2.

Regarding the schedulers, there is a clear correlation between performance and energy consumption. Therefore, the algorithms that offer the best performance in co-execution are also the ones that consume the least energy. On the contrary, the schedulers that cause a great deal of imbalance by giving more work to the CPU, spike the energy consumption, due to the higher usage of CPU cores, like Gaussian and Mandelbrot with Dyn5, and RAP with Static.

An interesting behavior can be observed in MatMul. The balancing efficiency using dynamic algorithms is very close to 1.0 in all cases. However, the performance obtained is very different when using buffers compared with USM. The explanation is found by analyzing the memory power consumption results shown in Figure 9. It can be seen that the memory power consumption (and therefore the memory usage) in the case of buffers is much higher than in the case of USM. The memory power consumption results correlate perfectly with the performance results. Taylor presents a similar behavior.

Another interesting metric is the energy efficiency, which relates performance and energy consumption. In this case, it is represented by the ratio of the Energy-Delay Product of the GPU with respect to the co-execution, presented in Figure 10. Therefore, values higher than 1.0 indicate that the co-execution is more energy efficient than the GPU.

Looking at the geometric mean, it can be concluded that co-execution is 72% more energy efficient than the GPU execution, using the HGuided scheduler and the USM memory model. Furthermore, this metric is favorable to co-execution in all benchmarks studied, reaching improvements of up to 2.8× in Taylor and RAP. Thus, while co-execution consumes more energy in absolute terms on certain benchmarks, the reduction in execution time compensates for this extra consumption, thus, resulting in a better performance-energy trade-off.

6.4. NBoby Benchmark

NBody presents an outstanding behavior, and a detailed study of the characteristics of the application, together with an evaluation of the throughputs, highlighted the importance of dynamic strategies. Figure 11 shows the speedups when using a single device, the CPU and GPU independently, for each supported memory model, when using different package distributions with the Dynamic scheduler. Each of the graphs shows five configurations of the scheduler, with 5, 50, 100, 400, and 800 packages of the same size, all dynamically scheduled. The abscissa axis shows the speedup with respect to the execution of a single package offloaded with the full problem size for a single device (*Single*). In addition, there are four rows of graphs, showing the results for four different numbers of molecules to compute in the NBody simulation, considering increasing problem sizes, from size 1 to 4.

It can be contrasted how NBody offers an extraordinary behavior when using the dynamic strategy offered by *Coexecutor Runtime*, generally offering higher speedups as the number of packages increases. However, when using a very high number of packages, a turning point is reached, and the speedup starts to drop. This behavior occurs earlier when using *Buffers* than when using shared memory *USM*. Additionally, the speedups obtained are not as good in the CPU compared with the GPU; however, on the other hand, the difference between memory models is not as important when using the CPU device.

In bigger problem sizes (*Size4*), the larger package configurations, such as *Dyn400* or *Dyn800*, obtain even better speedups, especially on CPU. Therefore, the inflection point for these cases is at an even higher number of packages where there are too many packages, which increases the communications penalization, thus, degrading the overall performance.

Furthermore, even if the management and synchronization are increased, the overhead is considerably reduced when using USM. NBody's behavior is excellent due to its kernel characteristics, architecture and efficiency of the co-execution runtime, which is able to overlap computation and communication with the maximum possible throughput. In addition, it is possible that the implementation of DPC++ and the oneAPI extensions recognize the independence of the data computation given by the multi-threaded architecture of *Coexecutor Runtime*, generating multiple command queues to transfer memory regions and perform parallel kernel executions, overlapping computation and communication, as has been seen in other runtimes [14,17,28,30].

The results presented in this section are from the Desktop node, but similar behaviors are found in the DevCloud server.

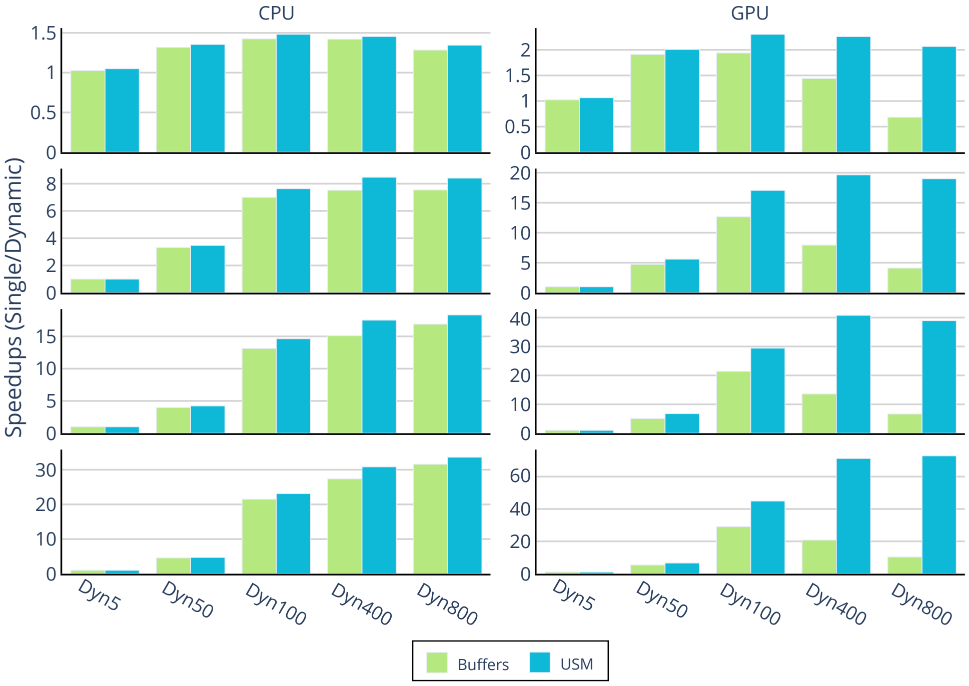


Figure 11. NBody speedups when using single-device dynamic policies for a set of increasing problem sizes. Baselines are single CPU or GPU execution per memory model.

7. Related Work

Intel oneAPI is a specification and framework to unify programming across its compute product portfolio (CPUs, GPUs, NPUs, and FPGAs) with a single set of APIs. Although its release is very recent (end of 2019), it has quickly attracted the attention of industry and the scientific community, which works on heterogeneous systems.

A SYCL-based version of the well-known Rodinia benchmark suite was developed in [32], using Intel oneAPI toolkit, to track the development of the SYCL compilers and provide programs for the study of heterogeneous systems. Christgau and Steinke [33] used both the compatibility tool *dpct* of oneAPI, as well as SYCL extensions for the CUDA base code of the easyWave simulator.

A study of the performance portability between different Intel integrated GPUs, of a computationally intensive routine derived from the Hardware Accelerated Cosmology Code (HACC) framework is presented in [34]. The authors used oneAPI to build the OpenCL and SYCL programs. Moreover, Wang et al. evaluated the portability effort and performance achieved with oneAPI for a medical ultrasonic image processing application using a GPU and a FPGA [4].

A debugger based on GDB for SYCL programs that offload kernels to CPU, GPU, or FPGA emulator devices, was developed as part of the oneAPI distribution [35]. Due to the code unification provided by SYCL, this debugger allows working on both the main program and the kernel, simultaneously. DPC++ as part of oneAPI, has proposed and implemented several extensions on top of SYCL [25]. Some of the extensions are devoted to improving application performance, but all of them improve programmer productivity, by integrating easily into existing C++ applications and by simplifying common patterns found in SYCL and C++. A study about how oneVPL (oneAPI Video Processing Library) is used in Certiface technology designed to combat fraud and protect honest people by ability to differentiate between a live person and a recorded video is presented in [36].

As far as we know, the only work that addresses co-execution with oneAPI is [37]. The authors extended the Intel TBB *parallel_for* function to allow simultaneous execution of the same kernel on CPU and GPU. They implemented three schedulers on top of oneAPI, static, dynamic, and adaptive LogFit. The main differences with our work are that we provided a pure oneAPI architecture (without TBB) and presented a rich variety of kernels, both regular and irregular, which revealed differences in the behavior of the schedulers.

8. Conclusions

Hardware heterogeneity complicates the development of efficient and portable software, due to the complexity of their architectures and a variety of programming models. In this context, Intel developed oneAPI, a new and powerful SYCL-based unified programming model with a set of domain-focused libraries, facilitating development among various hardware architectures.

This paper provides co-execution to oneAPI to squeeze the performance out of heterogeneous systems. This new and powerful unified programming model, based on SYCL, is a proposal from Intel to facilitate the programming of heterogeneous systems. In this work, the mechanisms of operation of the oneAPI runtime are exposed, as well as the data-parallel strategies to simultaneously use all devices. The *Coexecutor Runtime* overcomes one of the main challenges in oneAPI, the exploitation of dynamic decisions efficiently. Three load balancing algorithms were implemented on this runtime, showing the behavior with a set of regular and irregular benchmarks.

Furthermore, a validation of performance, balancing efficiency and energy efficiency was conducted, as well as a scalability study. Two heterogeneous systems were used to validate the runtime: a commodity desktop node and a HPC server. The results indicate that co-execution is worthwhile when using dynamic schedulers, specifically when using HGuided algorithm and unified memory. This was achieved due to efficient synchronization, architecture design decisions, computation and communication overlap, and the underlying oneAPI technology and its DPC++ compiler and runtime.

Co-execution was validated with CPUs and integrated GPUs; however, the proposed runtime is also capable of using other types of architectures that will be incorporated into oneAPI. Therefore, in the future, the co-execution runtime will be extended to evaluate new heterogeneous devices, such as FPGAs and discrete GPUs. New load balancing algorithms will be designed, taking advantage of the benefits offered by this programming model and the *Coexecutor Runtime*, since they can further exploit the performance due to the contributions related to the dynamic mechanisms presented in this paper.

Author Contributions: Investigation, R.N. and J.L.B., conceptualization, R.N.; methodology, R.N. and J.L.B.; software, R.N.; experiments, R.N.; validation, R.N. and J.L.B.; ; resources, R.N.; data curation, R.N.; writing—original draft preparation, R.N. and J.L.B.; writing—review and editing, R.N. and J.L.B.; visualization, R.N.; supervision and management, J.L.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Spanish Ministry of Education (FPU16/03299 grant) and the Spanish Science and Technology Commission under contract PID2019-105660RB-C22.

Acknowledgments: The European HiPEAC Network of Excellence.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Zahran, M. Heterogeneous Computing: Here to Stay. *Commun. ACM* **2017**, *60*, 42–45, doi:10.1145/3024918.
2. Lin, F.C.; Huy, N.H.; Dow, C.R. A cloud-based face video retrieval system with deep learning. *J. Supercomput.* **2020**, *76*, 8473–8493, doi:10.1007/s11227-019-03123-x.
3. Robles, O.D.; Bosque, J.L.; Pastor, L.; Rodríguez, A. Performance Analysis of a CBIR System on Shared-Memory Systems and Heterogeneous Clusters. In Proceedings of the Seventh International Workshop on Computer Architecture for Machine Perception, Palermo, Italy, 4–6 July 2005;IEEE Computer Society: 2005; pp. 309–314, doi:10.1109/CAMP.2005.40.
4. Wang, Y.; Zhou, Y.; Wang, Q.S.; Wang, Y.; Xu, Q.; Wang, C.; Peng, B.; Zhu, Z.; Takuya, K.; Wang, D. Developing medical ultrasound beamforming application on GPU and FPGA using oneAPI. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Portland, OR, USA, 17–21 June 2021; pp. 360–370.

5. Costero, L.; Igual, F.D.; Olcoz, K.; Tirado, F. Leveraging knowledge-as-a-service (KaaS) for QoS-aware resource management in multi-user video transcoding. *J. Supercomput.* **2020**, *76*, 9388–9403, doi:10.1007/s11227-019-03117-9.
6. Toharia, P.; Robles, O.D.; Suárez, R.; Bosque, J.L.; Pastor, L. Shot boundary detection using Zernike moments in multi-GPU multi-CPU architectures. *J. Parallel Distrib. Comput.* **2012**, *72*, 1127–1133, doi:10.1016/j.jpdc.2011.10.011.
7. Castillo, E.; Camarero, C.; Borrego, A.; Bosque, J.L. Financial applications on multi-CPU and multi-GPU architectures. *J. Supercomput.* **2015**, *71*, 729–739, doi:10.1007/s11227-014-1316-5.
8. Wang, X.; Gan, L.; Liu, S. Research on intelligence analysis technology of financial industry data based on genetic algorithm. *J. Supercomput.* **2020**, *76*, 3391–3401, doi:10.1007/s11227-018-2584-2.
9. Shin, W.; Yoo, K.H.; Baek, N. Large-Scale Data Computing Performance Comparisons on SYCL Heterogeneous Parallel Processing Layer Implementations. *Appl. Sci.* **2020**, *10*, 1656.
10. Mrozek, M.; Ashbaugh, B.; Brodman, J. Taking Memory Management to the Next Level: Unified Shared Memory in Action. In Proceedings of the International Workshop on OpenCL, Munich, Germany, 27–29 April 2020; doi:10.1145/3388333.3388644.
11. Pérez, B.; Bosque, J.L.; Beivide, R. Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, Barcelona, Spain, 12 March 2016; pp. 42–51.
12. Pérez, B.; Stafford, E.; Bosque, J.L.; Beivide, R. Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. *J. Supercomput.* **2017**, *73*, 330–342, doi:10.1007/s11227-016-1864-y.
13. Beri, T.; Bansal, S.; Kumar, S. The Unicorn Runtime: Efficient Distributed Shared Memory Programming for Hybrid CPU-GPU Clusters. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 1518–1534, doi:10.1109/TPDS.2016.2616314.
14. Nozal, R.; Bosque, J.L.; Beivide, R. EngineCL: Usability and Performance in Heterogeneous Computing. *Future Gener. Comput. Syst.* **2020**, *107*, 522–537, doi:10.1016/j.future.2020.02.016.
15. Augonnet, C.; Thibault, S.; Namyst, R.; Wacrenier, P.A. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. Pract. Exp.* **2011**, *23*, 187–198.
16. Gautier, T.; Lima, J.; Maillard, N.; Raffin, B. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Cambridge, MA, USA, 20–24 May 2013; pp. 1299–1308.
17. Nozal, R.; Perez, B.; Bosque, J.L.; Beivide, R. Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels. *J. Supercomput.* **2019**, *75*, 1123–1136.
18. Gaster, B.R.; Howes, L.W.; Kaeli, D.R.; Mistry, P.; Schaa, D. *Heterogeneous Computing with OpenCL-Revised OpenCL 1.2 Edation*; Morgan Kaufmann: Burlington, MA, USA, 2013.
19. Farber, R. *Parallel Programming with OpenACC*, 1st ed.; Morgan Kaufmann Publishers: San Francisco, CA, USA, 2016.
20. Vitali, E.; Gadioli, D.; Palermo, G.; Beccari, A.; Cavazzoni, C.; Silvano, C. Exploiting OpenMP and OpenACC to accelerate a geometric approach to molecular docking in heterogeneous HPC nodes. *J. Supercomput.* **2019**, *75*, 3374–3396, doi:10.1007/s11227-019-02875-w.
21. Ronan, K. Modern C++, Heterogeneous Computing & OpenCL SYCL. 2015. Available online: https://www.iwocl.org/wp-content/uploads/iwocl-2015-tutorial-SYCL-part2.pdf (accessed on 15 August 2021).
22. Da Silva, H.C.; Pisani, F.; Borin, E. A Comparative Study of SYCL, OpenCL, and OpenMP. In Proceedings of the 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Los Angeles, CA, USA, 26–28 October 2016; pp. 61–66, doi:10.1109/SBAC-PADW.2016.19.
23. Szuppe, J. Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL. In Proceedings of the 4th International Workshop on OpenCL, Vienna, Austria, 19–21 April 2016; doi:10.1145/2909437.2909454.
24. Corporation, I. *Intel® oneAPI Programming Guide*; 2020. Available online: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html (accessed on 15 August 2021).
25. Ashbaugh, B.; Bader, A.; Brodman, J.; Hammond, J.; Kinsner, M.; Pennycook, J.; Schulz, R.; Sewall, J. Data Parallel C++: Enhancing SYCL Through Extensions for Productivity and Performance. In Proceedings of the International Workshop on OpenCL, Munich, Germany, 27–29 April 2020; doi:10.1145/3388333.3388653.
26. Zhang, F.; Zhai, J.; He, B.; Zhang, S.; Chen, W. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 905–918, doi:10.1109/TPDS.2016.2586074.
27. Shen, J.; Varbanescu, A.L.; Lu, Y.; Zou, P.; Sips, H. Workload Partitioning for Accelerating Applications on Heterogeneous Platforms. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 2766–2780, doi:10.1109/TPDS.2015.2509972.
28. Nozal, R.; Bosque, J.L.; Beivide, R. Towards Co-execution on Commodity Heterogeneous Systems: Optimizations for Time-Constrained Scenarios. In Proceedings of the 2019 International Conference on High Performance Computing & Simulation (HPCS), Dublin, Ireland, 15–19 July 2019; pp. 628–635.
29. Nozal, R.; Bosque, J.L. Exploiting Co-execution with OneAPI: Heterogeneity from a Modern Perspective. In *Euro-Par 2021: Parallel Processing*; Sousa, L., Roma, N., Tomás, P., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 501–516.
30. Dávila Guzmán, M.A.; Nozal, R.; Gran Tejero, R.; Villarroya-Gaudó, M.; Suárez Gracia, D.; Bosque, J.L. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *J. Supercomput.* **2019**, *75*, 1732–1746, doi:10.1007/s11227-019-02768-y.

31. Bosque, J.L.; Perez, L.P. Theoretical scalability analysis for heterogeneous clusters. In Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), Chicago, IL, USA, 19–22 April 2004; IEEE Computer Society: 2004; pp. 285–292, doi:10.1109/CCGrid.2004.1336579.
32. Jin, Z. *The Rodinia Benchmark Suite in SYCL*; Technical Report; Argonne National Lab. (ANL): Lemont, IL, USA, 2020.
33. Christgau, S.; Steinke, T. Porting a Legacy CUDA Stencil Code to oneAPI. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops, New Orleans, LA, USA, 18–22 May 2020; pp. 359–367.
34. Jin, Z.; Morozov, V.; Finkel, H. A Case Study on the HACCmk Routine in SYCL on Integrated Graphics. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops, New Orleans, LA, USA, 18–22 May 2020; pp. 368–374, doi:10.1109/IPDPSW50202.2020.00071.
35. Aktemur, B.; Metzger, M.; Saiapova, N.; Strasuns, M. Debugging SYCL Programs on Heterogeneous Intel® Architectures. In Proceedings of the International Workshop on OpenCL, Munich, Germany, 27–29 April 2020; doi:10.1145/3388333.3388646.
36. Tibrewala, S.; Faria, A.D.O. Making Banking Secure via Bio Metrics Application Built Using OneAPI and DPC++ Based on SYCL/C++. In Proceedings of the International Workshop on OpenCL, Munich, Germany, 27–29 April 2020; doi:10.1145/3388333.3388671.
37. Constantinescu, D.A.; Navarro, A.G.; Corbera, F.; Fernández-Madrigal, J.A.; Asenjo, R. Efficiency and productivity for decision making on low-power heterogeneous CPU+GPU SoCs. *J. Supercomput.* **2021**, 77, 44–65, doi:10.1007/s11227-020-03257-3.