The Journal of Supercomputing (2025) 81:452
https://doi.org/10.1007/s11227-025-06963-y

CPU-GPU co-execution through the exploitation of hybrid technologies via SYCL

Nozal Raúl[1] · **Jose Luis Bosque**[1]

Accepted: 17 January 2025

Abstract

The performance and energy efficiency offered by heterogeneous systems are highly useful for modern C++ applications, but the technological variety demands adequate portability and programmability. Initiatives such as Intel oneAPI facilitate the exploitation of Intel CPUs and GPUs, but not NVIDIA GPUs, which are present in systems of all kinds and are necessarily leveraged by CUDA technology. Frequently, only GPUs are used, leaving the CPU for management tasks, with the consequent loss of energy and system utilization. In this work, the CoexecutorRuntime system design and API are extended to transparently integrate backends of diverse technologies, unifying offloading mechanisms under a consistent co-execution API and scheduling runtime. Moreover, CPU-GPU co-execution of hybrid technologies is enabled to ensure performance portability. Experimental results show performance improvements for all programs studied, achieving average efficiencies of 0.91 and speedups of 1.31 over using only the GPU.

Keywords Heterogeneous computing · Hybrid parallel computing · Co-execution · SYCL · OpenCL · CUDA · OneAPI · Performance portability · LLVM · Usability · Load balancing

Nozal Raúl and Bosque Jose Luis have contributed equally to this work.

✉ Jose Luis Bosque
joseluis.bosque@unican.es

Nozal Raúl
raul.nozal@unican.es

[1] Department of Computer Engineering and electronics, Universidad de Cantabria, Avda. los castros, s/n, 39.005 Santander, Spain

Published online: 31 January 2025

1 Introduction

Through the popularization and exploitation of heterogeneous systems, unprecedented levels of performance and energy efficiency are being achieved. These benefits generate a trend toward the incorporation of new and powerful programming models, with the aim of being able to use specific accelerators by a whole amalgam of applications and languages. However, technologies such as oneAPI or CUDA, despite being able to efficiently exploit the devices, are constrained by the manufacturers themselves, limiting their use in a cooperative manner and between different architecture types and vendors.

In this situation, it is essential to provide tools that make life easier for programmers, offering code portability and programmability advantages. Considering the complexity of the business logic of modern C++ applications, the more abstraction, flexibility and maintainability given to these proposals, the better. Languages such as OpenCL are powerful and portable, but their low level of abstraction and verbosity keep them far from the program domain. In addition, their performance is not as good as with native technologies, which complicates their use. Faced with this situation, proposals and standards of a higher level of abstraction have emerged, such as SYCL, which attempt to favor a single language for specifying execution kernels for heterogeneous devices. However, it does not allow the full potential of the machine to be exploited, since co-execution, load distribution and performance portability mechanisms are absent.

For this reason, software architectures and designs, such as CoexecutorRuntime, allow to take advantage of the simultaneous execution of kernels on several devices [1, 2]. It is a runtime system which provides a high-level API to abstract heterogeneous execution, maintaining compatibility with SYCL while incorporating flexible scheduling algorithms. Originally designed for oneAPI, it enables simultaneous kernel execution across multiple devices and offers an extensible architecture for performance and energy optimization. Its extensible scheduling system favors specific load balancing proposals without distracting the programmer from the intricacies of the business logic. In this work, CoexecutorRuntime is used as API and co-execution system on which to design and build a proposal that supports hybrid technologies, leveraging CUDA for NVIDIA GPUs and oneAPI for the system CPU. In this way, high maintainability and performance portability are assured.

Considering other related works, none has been found that achieves something similar to what is presented in this proposal, but they can be classified considering three aspects addressed. On the one hand, there are works related to the use of SYCL and NVIDIA, performing transformations and code portability, but they do not involve co-execution [3–11]. On the other hand, there are efforts to perform execution through the specialization of kernels or the use of explicit hybrid technologies [12–22]. And finally, there are proposals that address the problem of CPU-GPU co-execution in SYCL, albeit involving the same execution technology for all devices [1, 2, 23].

Our work offers a runtime with a high-level API, part of the program domain, facilitating the integration in real-world C++ applications, mimicking the API

provided by CoexecutorRuntime, but completely compatible with SYCL, encapsulating the inner technology to increase its flexibility. Therefore, the co-execution system supports a new hybrid mode of execution, exploiting CUDA and oneAPI programming models. It allows using NVIDIA GPUs and Intel CPUs simultaneously, achieving performance portability through its scheduling system, benefiting from a set of load balancing algorithms. Experimental results show that the co-execution is worthwhile compared with the fastest device, the GPU. It achieves speedups of up to 2.41 for irregular programs, and 1.12 for regular ones, obtaining average efficiencies of 0.91 with the best scheduling configuration.

The main contributions of this work are:

- Providing a high-level SYCL runtime system and a compatible API that is based on CoexecutorRuntime, but abstracted from the original oneAPI design, allowing kernel offloading to NVIDIA GPUs and Intel CPUs.
- Integrating and enhancing the co-execution system to support hybrid technologies, computing simultaneously with oneAPI on the CPU and CUDA on the GPU, leveraging the system with highly optimized load balancing algorithms.

The rest of the paper presents in Sect. 2 a motivation of this work, while Sect. 3 exposes the design of the novelties of the co-execution architecture to support the new backend and the hybrid co-execution mode. Section 4 details the methodology and the experimental results to validate the proposal and its enhancements. Finally, Sect. 5 presents the most relevant related works while Sect. 6 highlights the most important conclusions and future work.

2 Motivation

With the emergence and popularization of heterogeneous systems, a new world of possibilities opens up in terms of performance and energy efficiency. However, all this variety of devices and architectures exposes a complexity in programming models and standardization of computing techniques that makes the use of open, powerful and reliable technologies indispensable. One of the most recent ones, Intel oneAPI, provides the mechanisms to efficiently exploit modern C++ applications through the SYCL language. This programming model offloads the most computationally intensive regions to all types of Intel devices.

However, many heterogeneous nodes have discrete graphics cards that are not usable with this technology, such as NVIDIA, which is present in both commodity nodes and large HPC clusters. Moreover, with the popularization of CUDA technology for all kinds of applications and areas of knowledge, in many cases one has to make the uncomfortable decision of choosing between computing using the GPU or the CPU. In these situations, the former is favored, due to the popularization of the CUDA language and the efficiency of these accelerators [4–6, 12], leaving the CPU to perform the management tasks, which is typical in the host-device programming model. This facilitates programming, as it is not involved in the program computation. However, in most cases these are powerful CPUs that continue consuming

energy without contributing anything to the computation. Therefore, it is convenient to co-execute the problem to void harming both the performance and energy consumption of the system [24, 25]. The co-execution allows all devices, including the CPU, to operate on the same problem, consuming less time and energy to solve it. This technique has proved that it can achieve very good results, maximizing the performance and efficiency of heterogeneous systems [26].

In order to take advantage of these devices, it is necessary to provide mechanisms so that the computing modes of both devices can operate simultaneously. Since it is necessary to maintain software portability and facilitate the implementation of business logic for the programmers, it is required to use the same language. Using OpenCL for CPU and GPU takes the programmer away from the problem domain, due to its verbosity and low level compared to the high-level C++ applications.

It is not sufficiently maintainable, it is error prone and, above all, it does not achieve an appropriate performance compared to the native technology in some of these devices, such as OpenCL or OpenMP compared with CUDA, for the NVIDIA GPUs. For this reason, the choice is to provide a co-execution system based on SYCL, but capable of exploiting the full potential of CUDA for the GPU.

However, this approach introduces yet another challenge, and that is to ensure performance portability, an issue that has to be addressed transparently to the programmer. Not only does it have to allow working with a single code that implements the kernel, but it has to be efficient in distributing the load between CPU and GPU, as well as allowing to exploit each device with the appropriate compute technology.

3 CoexecutorRuntime/SYCL

Below are the key decisions and enhancements made to the CoexecutorRuntime runtime and its original proposal. This section focuses on the novelties, affected components and new design decisions that enable the use of CPUs and NVIDIA GPUs, as well as efficient co-execution with hybrid technologies, while maintaining a compatible API.

3.1 Mixing technologies via LLVM and SYCL

CoexecutorRuntime is based on performing an abstraction on Intel oneAPI technology, entirely focused on Intel devices, specifically to perform co-execution of a CPU and an integrated GPU. This abstraction provides a higher-level API with the goal of facilitating the programmability and maintainability of modern C++ applications, so that programmers can focus on business logic. But, at the same time, it provides an exposition of the oneAPI primitives, with the goal of being easily extensible and having a smooth porting of a traditional host-device application to the co-execution scheme offered by CoexecutorRuntime. In addition, some of the oneAPI functionalities have been integrated into CoexecutorRuntime and exposed through its API, such as the possibility of using pointers and memory regions of the running application.

This proposal extends its use to multivendor devices, specifically for NVIDIA and Intel. The original architecture was constrained to the oneAPI technology for Intel CPUs and integrated GPUs, but the runtime needs to be extended to support NVIDIA discrete GPUs. Thus, a conversion of the runtime core is performed to accommodate other technologies and its support to simultaneously mix them during the application execution. The encapsulation software layers and the different technologies involved are shown in Fig. 1. The difficulties to support NVIDIA GPUs could be bypassed by using OpenCL for the NVIDIA GPU, but as demonstrated in previous studies, it is not an efficient solution and would penalize the maintainability and performance. This approach would require the OCL interoperability layer of oneAPI, involving multiple kernel codes. This layer provides an OpenCL interoperability API so that OpenCL code can be used from oneAPI to offload kernels to an OpenCL-compliant device. Therefore, the designed runtime system is built on top of SYCL, as an abstraction that encapsulates the behaviors and optimizations provided by the SYCL programming model implementations, satisfying the original design decisions of CoexecutorRuntime. To this end, the system is built by combining two fundamental technologies, oneAPI for the CPU and NVIDIA CUDA for the GPU.

The fundamental idea is to provide a programming model compatible with SYCL, avoiding the creation of a new language to interact with the accelerators, while using efficient technologies to compute on the devices involved. Therefore, hybrid technologies are used to compute, but are not exposed to the programmer. The ease of use is given by the single source code property, and the performance is given by exploiting the best technology for each device, even though they are based on different programming models. In this proposal, CoexecutorRuntime/SYCL has been designed, implemented and evaluated using the LLVM compiler, building it with support for the experimental NVIDIA CUDA backend and the OpenCL/DPC++ backend for CPUs, using a modern version of the Khronos OpenCL ICD to match symbols. Fat binaries generated using this system require both the SYCL libraries provided directly by oneAPI, providing SPIR-V binaries (spir64) for the CPU, as well as CUDA and OpenCL libraries provided by the NVIDIA CUDA

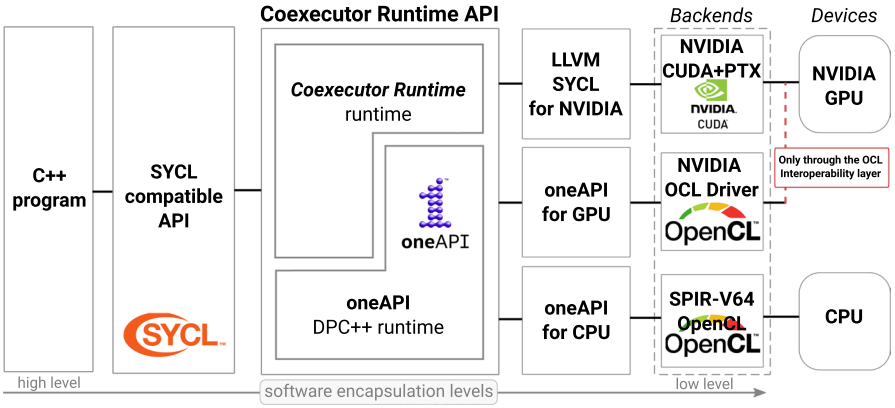

Fig. 1 Architectural layers of the CoexecutorRuntime to enable CPU and NVIDIA GPU devices

runtime (nvptx64-nvidia-cuda). One particularity of this technological combination is that the possibility of efficiently co-executing using the SYCL fallback host device is lost, since it lacks multithreading support. However, this is not a disadvantage since the system is provided with oneAPI for CPU, and in cases where it is not supported, thanks to the new software architecture, it is possible to provide, for example, a backend based on C++ STL threads.

3.2 Hybrid co-execution with NVIDIA GPUs

One of the advantages of decoupling the CoexecutorRuntime architecture is that it allows interacting with the various components in near isolation, facilitating their extension. Since the goal of the runtime is to facilitate programmability and allow to take advantage of various type of devices on heterogeneous nodes, it is necessary to satisfy the needs of co-execution. Considering that this is a hybrid execution model, the software architecture of the system has been affected in the modules in charge of building and launching work packages and kernels (`dispatcher_interface`) as well as in the co-execution units that encapsulate the hardware devices (`CoexecutionUnit`).

Figure 2 shows the general overview of the CoexecutorRuntime/SYCL scheduling system, updated to represent the variation produced in this proposal with respect to the work performed by the load balancing algorithms. Thanks to the encapsulation of the system, all the components of the system have been ported without modifications, such as `Director`, `Scheduler` or `CommanderLoop`, despite variations in the `CoexecutionUnit`, the SYCL runtime with its backends and the dispatcher mechanism. In this way, all original design decisions are preserved, so there is no impact on the design of the load distribution system, the synchronization and monitoring, the collection of results and other necessary stages during the co-execution cycles. This is a fundamental aspect to achieve an efficient offloading when co-executing, as will be seen in the experimental evaluation.

CoexecutionUnits are now provided with mechanisms to detect, filter and configure devices, adapting to the requirements of the programmer, allowing the specialization of kernels, buffers or data transfers per device or backend, such as applying certain optimizations in the case of the GPU. These variations have been prepared with runtime extensibility in mind, as this proposal is focused on the co-execution of

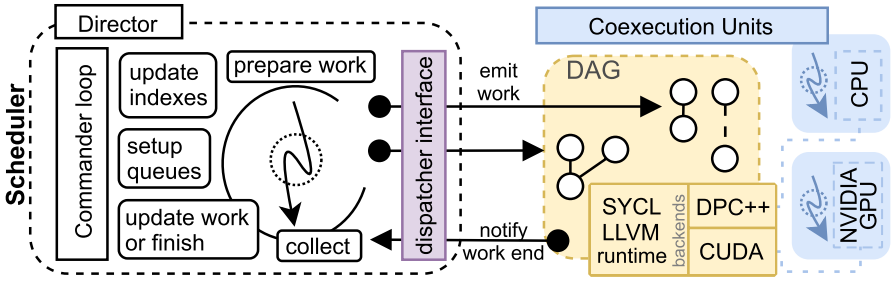

Fig. 2 Scheduling system overview and its encapsulated interaction with the CoexecutionUnits

Intel CPUs with an NVIDIA GPU, but more diverse configurations could be accommodated in the future.

Having different backends in the SYCL runtime facilitates the construction of the application and the necessary toolchains, as well as the execution through a single-entry point to the application. This is fundamental due to all the implications it entails in terms of portability and in order to exploit optimizations and co-execution mechanisms, without the need for multiple compilation steps or costly inter-process calls and memory mappings.

Variations in the dispatcher interface have affected the way kernels are built and in the case of certain extensions, since backends other than SYCL are being used. It has been required to generate kernels based on the backend and the scheduling algorithm used, something that was not affected when only oneAPI was used. This implication is given by the different passes made in the compilation process and the translation units. Therefore, kernels are compiled for different architectures and bundled in the final binary. For example, for the NVIDIA GPU used in Methodology Sect. 4.1, the final PTX has been built using CUDA 11.6 for the SM60 architecture.

Finally, one of the fundamental points of this proposal is to be compatible with the CoexecutorRuntime high-level co-execution API, and with SYCL in terms of the kernel to be executed. Listing Fig. 3 shows a code snippet using the simple API mode exposed by CoexecutorRuntime, but with the mechanisms introduced by this proposal and offered to programmers. Lines 5 to 9 enable the selection of the co-execution units involved in the computation, being the CPU and any NVIDIA GPU.

Thanks to these mechanisms, the same filters can be applied during the co-execution process, inside of the `launch` region (lines 10 to 27). For instance, in this example the GPU and CPU exploit the new feature of kernel specialization, altering the computation based on specific properties of each device or backend involved (lines 12 and 27). The final code remains as maintainable as in the original proposal and being compatible with any SYCL kernel (lines 20 to 24), but benefiting from the simultaneous execution on a CPU and a NVIDIA GPU. Therefore, thanks to these decisions, performance portability is still ensured, leveraging the node performance through the exploitation of any of the different load balancing algorithms provided (lines 2 to 4).

3.3 Load balancing algorithms

To achieve efficient co-execution, one of the fundamental aspects is to distribute the workload proportionally to the computational capacity of the devices, CPU/GPU. The way to do this is to divide the dataset to be processed into a series of *packages* and assign them correctly to the devices. The CoexecutorRuntime currently supports three well-known load balancing algorithms which are described briefly below [24, 27]. The programmer should decide which one to use in each case, depending on the characteristics and knowledge he has of the architecture.

Static This algorithm works before the kernel is executed by dividing the workload in as many packages as devices are in the system. The division relies on knowing the relative computational capacity of each device, in advance. Then

```
// C++ containers and values from the application domain (before)
coexecutor_runtime<dyn> runtime; // instantiating the runtime
runtime.config(CoexecutorSet::Scheduler, // Dyn with 80 Packages
               coexecutor_runtime::dist(80));
runtime.config(CounitSet::DeviceFilter, [&](coexecutor_unit *cu){
  return (cu->is(CounitSet::Mask::GPU) && (cu->is("NVIDIA"))) ||
         (counit->is(CounitSet::Mask::CPU));
         /* can also expose sycl::device to operate with */
});
runtime.launch(data.size(), [&](coexecutor_unit *cu, package p){
  if (cu->is(CounitSet::Mask::GPU)){ /* GPU specialization */
    sycl::buffer<int, 1> buf_input(data.data() + p.offset,
    sycl::range<1>(p.size));
    // more buffers and offsets...
    cu->dispatch([&](sycl::handler &h) {
      auto R = sycl::range<1>(pkg.size);
      auto input = buf_input
                   .get_access<sycl::access::mode::read_write>(h);
      // more ranges and accessors...
      h.parallel_for(R, [=](sycl::item<1> it) {
        auto tid = it.get_linear_id();
        input[tid] = input[tid] * datav;
        // rest of the kernel with the business logic...
      });
    });
  } else { /* other conditions for the CPU */ }
});
// co-execution has finished, all results in data containers
```

Fig. 3 CoexecutorRuntime API is preserved while adding extensions to interact with NVIDIA devices and CoexecutionUnit behavior specialization

the execution time of each device can be equalized by proportionally dividing the workload among the devices. It minimizes the number of synchronization points; therefore, it performs well when facing regular loads with known computing powers that are stable throughout the workload execution. However, it is not adaptable, so its performance might not be as good with irregular kernels, where the execution time of each package varies along the execution (for instance, in sparse computing).

Dynamic It divides the workload in a given number of equal-sized packages. The number of packages is well above the number of devices in the heterogeneous system. During the execution of the kernel, the *Scheduler* is in charge of assigning packages to the different devices, including the CPU. This algorithm adapts to the irregular behavior of some applications. However, each completed package represents a synchronization point between the device and the host, where data is exchanged and a new package is launched. This overhead has a noticeable impact on performance if the number of packages is high. The dynamic algorithm takes the size of the packages as a parameter.

HGuided This algorithm offers a variation over the dynamic by establishing how the workload is divided. The algorithm makes larger packages at the

beginning and reduces the size of the subsequent ones as the execution progresses. Thus, the number of synchronization points and the corresponding overhead is reduced, while retaining a small package granularity toward the end of the execution to allow all devices to finish simultaneously. Since it is an algorithm for heterogeneous systems, the size of the packets is also dependent on the computing power of the devices. The size of the package for device i is calculated as follows:

$$\text{packet\_size}_i = \lfloor \frac{G_r P_i}{k\, n \sum_{j=1}^{n} P_j} \rfloor$$

where k is a constant, for which, in previous works [25], it has been empirically obtained that the best results are obtained with values of k equal between 2 or 3. The smaller the k constant, the faster decreases the packet size. Tweaking this constant prevents too large packet sizes when there are only a few devices, with cases such as giving half the workload in the first packet to a device, unbalancing the load.

G_r is an integer number that stores the amount of workload (threads) pending to be executed in each new assignment, and it is updated with every package launch. The parameters of the HGuided are the computing powers and the minimum package size of the devices to be used. P_i is the computational power of the device i, and the $\sum_{j=1}^{n} P_j$ represents the total computational power of the system, where n is the number of devices in the system. The computational power value for each device (P_i) is obtained by normalizing the benchmark execution time on each device by the time of the most powerful device. Thus, the P_i of the most powerful device will be 1.0, and the values for the rest of the devices will be between 0 and 1. In this way, for example, a total computational power ($\sum_{j=1}^{n} P_j$) equal to 1.5 means that the whole system has the computational power equivalent to 1.5 times that of the most powerful device. The minimum package size is a lower bound for the $packet\_size_i$ and the minimum package sizes are usually dependent on the computing power of the devices, being bigger package sizes in the most powerful devices.

4 Evaluation

This section presents the methodology and experimental results that determine the behavior of the proposal, CoexecutorRuntime/SYCL, using CPU and GPU with two different technologies (oneAPI and CUDA), performing the co-execution for a set of applications with respect to the most efficient device, the GPU.

4.1 Methodology

The experiments are carried out on a computer with 16 GiB of RAM composed of an Intel Core i7-10700 CPU and an NVIDIA GeForce GT 1030 GPU with 2 GiB of GDDR5. The CPU has 8 cores and 2 threads per core, employing OpenCL 3.0 to provide 16 compute units at 2.9GH, while the GPU uses

the Compute Capability 6.1 of the CUDA backend, powered by the NVIDIA driver version 510, to expose 3 compute units at 1468 MHz, as it has 3 Streaming Multiprocessors.

Five programs have been selected to evaluate the behavior of the runtime [28]. Gaussian and matmul present regular behavior, and Rap, Rayc and Ray are irregular, where each part of the problem involves different computational complexities when executed on the same device. Rayc and Ray are Raytracing computations, but Rayc uses C++ classes to package its data types (vectors) and involves many more lights and objects. These have been chosen as they validate the co-execution behavior for both regular and irregular problem types, showing sufficiently diverse offloading pattern types and with good results, as demonstrated in previous studies [24].

To guarantee integrity of the results, the values reported are the arithmetic mean of 25 executions, discarding a previous first one to avoid warm-up penalties. The standard deviation is not shown because it is negligible in all cases. To measure the energy consumption, another 25 executions are performed to avoid introducing time delays due to the sampling overheads.

The validation of the proposal is done by analyzing the performance and energy efficiency of the new hybrid CPU-GPU co-execution mode compared with the fastest and most energy efficient device, the GPU. The total response time is measured, including kernel computing and data transfer. Two CoexecutorRuntime scheduling configurations are evaluated when co-executing, static and HGuided (an improved version of dynamic) [29].

Three metrics are used to evaluate the proposal, speedup, heterogeneous efficiency and energy efficiency [29]. The speedup is calculated as $S = \frac{T_{\text{GPU}}}{T_{\text{co-exec}}}$, being T_{GPU} and $T_{\text{co-exec}}$ the execution times for the GPU and the co-execution, respectively. Due to the heterogeneity of the system and the different behavior of the benchmarks, the maximum achievable speedups depend on each program. As a result of this, the heterogeneous efficiency has been computed as the ratio between the empirically obtained speedup and the maximum achievable speedup, for each benchmark [24]. The maximum speedup is computed as the sum of the computational power of each of the devices in the system, since, as explained in section 3.3, this value represents the computational capacity of the complete system, and therefore the maximum acceleration that can be obtained:

$$Sp_{\max} = \sum_{i=1}^{n} P_i$$

Finally, energies are measured using RAPL counters for the CPU and RAM, and nvidia-smi along a real-time power usage monitor to measure the whole system and the GPU, giving the total consumption in Joules. The energy-delay product (EDP) is used to evaluate the energy efficiency, measured in Js [30]. Since the values obtained have a very wide range, this metric is provided normalized with respect to the EDP of the GPU execution.

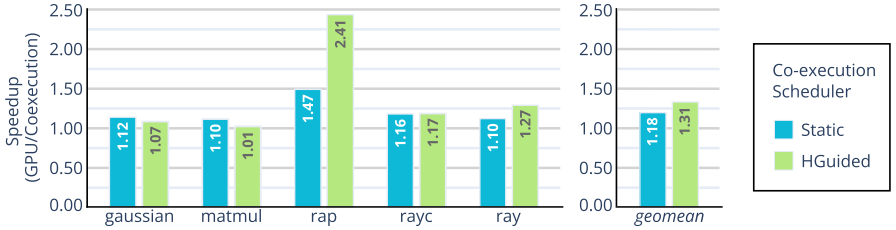

Fig. 4 Speedup when co-executing CPU-GPU compared with the GPU

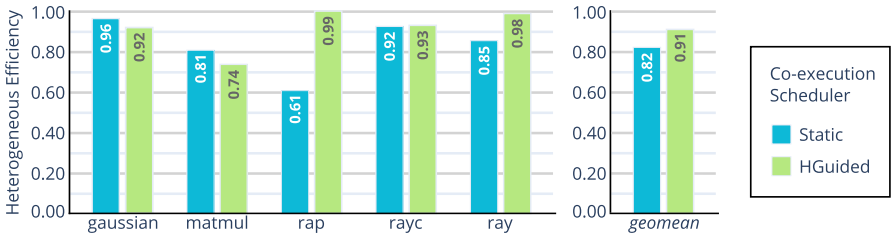

Fig. 5 Heterogeneous efficiency when co-executing CPU-GPU compared with the GPU

4.2 Experimental results

This section presents the experimental results with respect to the performance, the load balancing and the energy efficiency of the proposal.

Regarding the performance of the CoexecutorRuntime, Figs. 4 and 5 show, respectively, the speedup and the heterogeneous efficiency obtained, for all the benchmarks and the two load balancing algorithms evaluated, static and HGuided, together with the geometric mean, in the platform described in the previous section.

The main conclusion that can be drawn from the speedup results is that co-execution is profitable in all the cases studied, since it provides speedup results always higher than 1.0. Looking at the average values, it can be seen that the gains obtained are 1.31x with the HGuided algorithm and 1.18x with the static algorithm, which can be considered very good results considering the performance difference between the CPU and GPU devices. Analyzing the results of the benchmarks individually, it can be seen that the static algorithm provides slightly higher speedups in regular benchmarks (Gaussian and matmul), but HGuided presents much better results in irregular ones, reaching a gain of even 2.41x in *rap*.

The results of the heterogeneous efficiency, presented in Fig. 5, provide an idea about the utilization of the system as a whole, reaching a value of 1.0, when both devices are used at 100% in the co-execution. This metric reaches mean values of 0.82 for static and 0.91 for HGuided, which means a very good utilization of both devices in all benchmarks. These results can be considered very good, taking into account that we are working with a very heterogeneous platform, i.e., the computational capacity of both devices is very different and with irregular benchmarks.

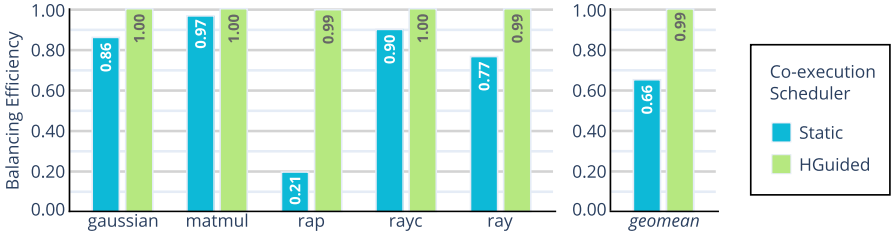

Fig. 6 Balancing efficiency when co-executing CPU-GPU

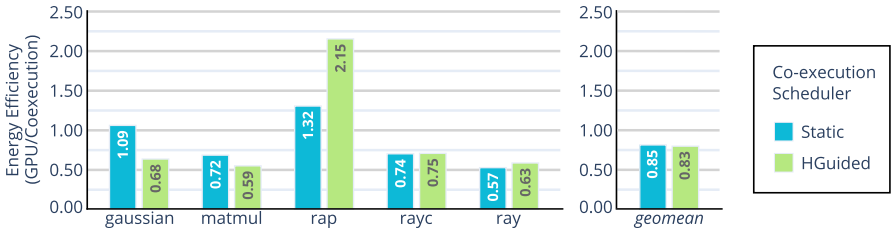

Fig. 7 Energy efficiency improvement of the co-execution over the GPU

The benefits are higher for irregular benchmarks (RAP, Rayc and Ray) than for regular ones, where the efficiency rises to 0.96. Gaussian has a time-consuming initialization phase of the data structures on both devices, which cannot be parallelized. Therefore, in co-execution this phase has to be performed by both devices.

As for load balancing algorithms, Fig. 6 presents the balancing efficiency, defined as the ratio between the execution time of both devices. Therefore, the optimal value of this metric is 1.0, where both devices finish simultaneously without idle times. 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 *CoexecutorRuntime*, as the host.

It is worth highlighting the good results obtained in this metric, especially for the HGuided algorithm, with a value above 0.99 in all cases. However, being a dynamic algorithm, it introduces a slight overhead that penalizes the total execution time. This overhead is more detrimental in regular benchmarks, so the performance in those cases is better with the static algorithm.

Another very interesting metric is 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 Fig. 7. Therefore, values higher than 1.0 indicate that the co-execution is more energy efficient than the GPU.

The results show that only in some cases (Gaussian with static and RAP in both cases) it is worthwhile to co-execute in terms of energy efficiency, compared with just using the GPU. This is reasonable since the GPU is a much more energy efficient device than the CPU, specifically a low-profile GeForce GPU. Thus, the time reduction achieved with co-execution cannot compensate, in many cases, the

additional energy cost of using two devices. However, the differences are not so significant (15% on average), and are acceptable for the performance improvements achieved. Future work plans to use an integrated GPU as a device, which is likely to improve these energy efficiency results.

5 Related work

There is a clear interest in the scientific community in the study of performance portability between different heterogeneous platforms and programming models. In this sense, two kinds of work can be distinguished.

Firstly, those that carry out comparative analyses of benchmarks or mini-apps on different architectures and with different programming models. Among them, [31] compares the performance of the NIVIDIA V100 GPU using SYCL and CUDA, with three applications: BabelStream, Mixbench and Tiled Matrix-Multiplication. The study analyzes the performance in terms of DRAM bandwidth, kernel execution time, compilation time and throughput. The main conclusion is that, the performance of SYCL and CUDA is similar, but in some cases, the latter outweighs the former.

Meanwhile, Breyer et al. compare the different competing programming frameworks OpenMP, CUDA, OpenCL and SYCL, paying special attention to the two SYCL implementations hipSYCL and DPC++ [32]. The paper investigates the different frameworks with respect to their usability, performance and performance portability on a variety of hardware platforms from different vendors, i.e., GPUs from NVIDIA, AMD, and Intel and CPUs from AMD and Intel. The authors evaluate the performance of the least squares support vector machines for scientific computing. They point out that performance portability has not yet been fully achieved by any SYCL implementation.

Finally, Homerding and Tramm [5] evaluate the performance of benchmarks and mini-apps having both SYCL and CUDA implementations on a NVIDIA V100 GPU. While there is missing functionality support, the performance of running SYCL is competitive with using CUDA directly. The authors find that many of the performance differences are due to the ordering and choices of memory accesses.

Secondly, a lot of work has been done on the migration of different applications from CUDA to SYCL. For instance, [33] describes the experience of converting a CUDA implementation of a high-order epistasis detection algorithm to SYCL. The paper performs a detailed description of migration paths between CUDA and SYCL that can be useful to application and compiler developers. Evaluating the CUDA and SYCL applications on an NVIDIA V100 GPU, concluding that successful migration requires a good understanding of the two programming models. Christgau and Steinke [9] use both the compatibility tool *dpct* of oneAPI, as well as SYCL extensions for the CUDA base code of the easyWave simulator. They compare the performance of the original code running on Xeon processors using OpenMP as well as CUDA with the performance of the DPC++ counterpart on multicore CPUs as well as integrated GPUs. Jin and Vertter migrate representative kernels in bioinformatics applications from CUDA to SYCL, evaluate their performance on an NVIDIA GPU

and explain the performance gaps through performance profiling and analyses [34]. The results indicate that whether the performance of running SYCL is competitive with using CUDA directly depends on the applications and how they are optimized.

On the other hand, there are also many studies in the literature that propose the co-execution of tasks in different devices. For example, [35] offers co-execution evaluation but uses a static work-sharing technique, so the programmer must know the assignation per device in advance, not adapting to irregular problems. Furthermore, it offers an independent runtime applied for a real-world application, giving support for hybrid native code on the CPU combined with OpenCL code on GPU. The authors evaluate the scalability and expose the trade-offs between performance and power consumption. Proposals like SkelCL [36] and SkePU [37] provide data management and composable primitives and skeletons to build parallel applications, but the programmer is responsible of using their own data containers. The authors of [38] apply fuzzy neural networks to the task distribution problem. MultiCL [39] is an OpenCL runtime based on storing execution information for each kernel-device pair for future kernel launches. Finally, Unicorn [40] presents a parallel programming model based on a work-stealing task Scheduler. They are all based on OpenCL and do not allow the use of the SYCL programming model, as CoexecutorRuntime does.

Some authors also propose the use of hybrid technologies. As for instance, Hofmann et al. does a hybrid CPU/GPU implementation of particle simulations using OpenCL, but they compute different kernels per device [41]. They do near-field interactions in the GPU and far-field interactions in CPU, but they do not do co-execution among the devices. Furthermore, [16] address the same problem offering from another perspective. Our work uses a high-level API for C++, proposes a transformation of OpenCL code at compile time and offers two hybrid execution models to offer simplicity or flexibility, allowing the programmer to use also hand-made custom native parallelized/vectorized code.

Finally, it should be noted that LLVM-based compiler for SYCL (open source effort led by Intel) provided support for NVIDIA GPUs via CUDA backend, since its performance was inadequate on OpenCL backend. Previously, SYCL implementations targeted NVIDIA GPUs via the OpenCL backend. However, the problem with such an approach was that NVIDIA's OpenCL drivers offered limited support in terms of features to its GPUs. Also, both Intel and CodePlay have announced support for SYCL code to run on NVIDIA GPUs. However, the big advantage of CoexecutorRuntime is that it allows you to co-execute applications in different applications using the SCYL programming model on different devices, including NVIDIA GPUs.

6 Conclusions

Proposals such as CoexecutorRuntime provide high-level APIs that facilitate the programmability and maintainability of modern C++ applications. In addition, thanks to their efforts in providing a co-execution system with various load

balancing algorithms, they squeeze heterogeneous nodes appropriately, contributing to performance portability.

However, it is based on oneAPI technology, limiting its applicability to Intel devices. Since there are many applications and systems that focus on exploiting NVIDIA GPUs, it is necessary to offer solutions that take advantage of devices from both types of manufacturers. For this reason, this work designs a system that allows taking advantage of oneAPI for CPUs and CUDA for GPUs, based on the CoexecutorRuntime foundations, thus providing a compatible API. These native technologies leverage these devices optimally, so efforts have been focused on providing an efficient co-execution system that enables load distribution among devices and hybrid technologies. In this way, any programmer can benefit from sophisticated scheduling algorithms to further improve the performance of their applications, regardless of the business logic and problem domain, facilitating maintainability.

Experimental validation has been performed by evaluating 5 types of programs, always comparing the co-execution with respect to the fastest and most energy efficient device, the GPU. Speedups of up to 2.41 have been obtained, considering a specific program and scheduling configuration, but reaching average speedups and efficiencies of 1.31 and 0.91, respectively. As it is a very energy efficient GPU, only a few applications have been found in which co-executing with the CPU has compensated, but in these cases, they achieve improvements of up to 2.15 with respect to the GPU.

In the future, behavioral studies will be performed regarding multidevice executions and the exploitation of different memory models, including the co-execution support for various accelerators along with the CPU.

Author contributions All authors contributed equally to the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work has been supported by the Spanish Science and Technology Commission under contract PID2022-136454NB-C21, the Ministerio de Ciencia e Innovación; Proyectos de Transición Ecológica y Digital 2021 under grant TED2021-131176B-I00 and the European HiPEAC Network of Excellence.

Data availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no conflict of interest.

References

1. Nozal R, Bosque JL (2021) Exploiting co-execution with oneapi: heterogeneity from a modern perspective. In: European Conference on Parallel Processing, pp. 501–516
2. Nozal R, Bosque JL (2021) Straightforward heterogeneous computing with the oneapi coexecutor runtime. Electronics 10(19):2386
3. Peccerillo B, Bartolini S (2019) Phast—a portable high-level modern c++ programming library for GPUs and multi-cores. IEEE Trans Parallel Distrib Syst 30(1):174–189. https://doi.org/10.1109/TPDS.2018.2855182
4. Alekseenko A, Páll S, Lindahl E (2021) Experiences with adding sycl support to gromacs. In: IWOCL, pp. 17–1
5. Homerding B, Tramm J (2020) Evaluating the performance of the hipSYCL toolchain for HPC kernels on NVIDIA v100 GPUs. In: Proceedings of the International Workshop on OpenCL, pp. 1–7
6. Crisci L, Salimi Beni M, Cosenza B, Scipione N, Gadioli D, Vitali E, Palermo G, Beccari A (2022) Towards a portable drug discovery pipeline with SYCL 2020. In: International Workshop on OpenCL, pp. 1–2
7. Castaño G, Faqir-Rhazoui Y, García C, Prieto-Matías M (2022) Evaluation of intel's DPC++ compatibility tool in heterogeneous computing. J Parallel Distrib Comput 165:120–129
8. Reyes R, Brown G, Burns R (2020) Bringing performant support for NVIDIA® hardware to SYCL. In: Proceedings of the International Workshop on OpenCL, pp. 1–1
9. Christgau S, Steinke T (2020) Porting a legacy cuda stencil code to oneapi. In: Proc. of IPDPSW, pp. 359–367
10. Shin W, Yoo K-H, Baek N (2020) Large-scale data computing performance comparisons on SYCL heterogeneous parallel processing layer implementations. Appl Sci 10(5):1656
11. Alpay A, Soproni B, Wünsche H, Heuveline V (2022) Exploring the possibility of a hipSYCL-based implementation of oneapi. In: International Workshop on OpenCL, pp. 1–12
12. Breyer M, Van Craen A, Pflüger D (2022) A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. In: International Workshop on OpenCL, pp. 1–12
13. Baratta I, Richardson C, Wells G (2022) Performance analysis of matrix-free conjugate gradient kernels using SYCL. In: International Workshop on OpenCL, pp. 1–10
14. Doumoulakis A, Keryell R, O'Brien K (2017) Sycl c++ and opencl interoperability experimentation with triSYCL. In: Proceedings of the 5th International Workshop on OpenCL, pp. 1–8
15. Nozal R, Niethammer C, Gracia J, Bosque JL (2021) Feasibility study of molecular dynamics kernels exploitation using enginecl. In: Euro-Par 2021: Parallel Processing Workshops
16. Moreton-Fernandez A, Gonzalez-Escribano A, Llanos DR (2019) Multi-device controllers: a library to simplify parallel heterogeneous programming. Int J Parallel Prog 47(1):94–113
17. Joó B, Kurth T, Clark MA, Kim J, Trott CR, Ibanez D, Sunderland D, Deslippe J (2019) Performance portability of a wilson dslash stencil operator mini-app using kokkos and sycl. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 14–25
18. Shin W, Yoo KH, Baek N (2020) Large-scale data computing performance comparisons on sycl heterogeneous parallel processing layer implementations. Appl Sci 10:1656
19. Jin Z, Morozov V, Finkel H (2020) A case study on the haccmk routine in sycl on integrated graphics. In: Proceedings of IPDPSW, pp. 368–374. https://doi.org/10.1109/IPDPSW50202.2020.00071
20. Wang Y, Zhou Y, Wang QS, Wang Y, Xu Q, Wang C, Peng B, Zhu Z, Takuya K, Wang D (2021) Developing medical ultrasound beamforming application on gpu and fpga using oneapi. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 360–370. IEEE
21. Ashbaugh B, et al (2020) Data parallel c++: enhancing sycl through extensions for productivity and performance. In: International Workshop on OpenCL. IWOCL. ACM. https://doi.org/10.1145/3388333.3388653
22. Tibrewala S, Faria ADO (2020) Making banking secure via bio metrics application built using oneapi and dpc++ based on sycl/c++. In: International Workshop on OpenCL. IWOCL '20. ACM. https://doi.org/10.1145/3388333.3388671
23. Constantinescu DA, Navarro AG, Corbera F, Fernández-Madrigal JA, Asenjo R (2020) Efficiency and productivity for decision making on low-power heterogeneous CPU+GPU SOCS. J Supercomput. https://doi.org/10.1007/s11227-020-03257-3

24. Nozal R, Bosque JL, Beivide R (2020) Enginecl: usability and performance in heterogeneous computing. Future Gen Comp Syst 107(C):522–537. https://doi.org/10.1016/j.future.2020.02.016
25. Nozal R, Bosque JL, Beivide R (2019) Towards co-execution on commodity heterogeneous systems: Optimizations for time-constrained scenarios. In: 17th International Conference on High Performance Computing and Simulations HPCS, Ireland,, pp. 628–635. IEEE, https://doi.org/10.1109/HPCS48598.2019.9188188
26. Zhang F, Zhai J, He B, Zhang S, Chen W (2017) Understanding co-running behaviors on integrated CPU/GPU architectures. IEEE Trans Parallel Distrib Syst 28(3):905–918. https://doi.org/10.1109/TPDS.2016.2586074
27. Pérez B, Stafford E, Bosque JL, Beivide R (2021) Sigmoid: an auto-tuned load balancing algorithm for heterogeneous systems. J Parallel Distrib Comput 157:30–42. https://doi.org/10.1016/j.jpdc.2021.06.003
28. Becker M, Brown C (2024) Heterogeneous Offloading Benchsuite. Last accessed Dec. https://github.com/EngineCL/Benchsuite
29. Nozal R (2022) Optimizing Performance and Energy Efficiency in Massively Parallel Systems. PhD thesis, Universidad de Cantabria
30. Nozal R, Pérez B, Bosque JL, Beivide R (2019) Load balancing in a heterogeneous world: CPU-XEON phi co-execution of data-parallel kernels. J Supercomput 75(3):1123–1136. https://doi.org/10.1007/S11227-018-2318-5
31. Reddy Kuncham GK, Vaidya R, Barve M (2021) Performance study of gpu applications using sycl and cuda on tesla v100 gpu. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. https://doi.org/10.1109/HPEC49654.2021.9622813
32. Breyer M, Van Craen A, Pflüger D (2022) A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. In: International Workshop on OpenCL. IWOCL'22. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3529538.3529980
33. Jin Z, Vetter JS (2022) Performance portability study of epistasis detection using sycl on nvidia gpu. In: Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics. BCB '22. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3535508.3545591
34. Jin Z, Vetter JS (2022) Understanding performance portability of bioinformatics applications in sycl on an nvidia gpu. In: 2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp. 2190–2195. https://doi.org/10.1109/BIBM55620.2022.9995222
35. LaKomski D, Zong Z, Jin T, Ge R (2015) Optimal balance between energy and performance in hybrid computing applications. In: 2015 Sixth International Green and Sustainable Computing Conference (IGSC), pp. 1–8. IEEE
36. Steuwer M, Kegel P, Gorlatch S (2011) SkelCL - A portable skeleton library for high-level GPU programming. In: IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (May 2011), pp. 1176–1182 https://doi.org/10.1109/IPDPS.2011.269
37. Enmyren J, Kessler CW (2010) SkePU: a multi-backend skeleton programming library for multi-gpu systems. In: Proceedings of 4th International Workshop on High-Level Parallel Programming and Applications
38. Zhang F, Zhai J, He B, Zhang S, Chen W (2017) Understanding co-running behaviors on integrated CPU/GPU architectures. IEEE Trans Parallel Distrib Syst 28(3):905–918. https://doi.org/10.1109/TPDS.2016.2586074
39. Aji AM, Peña AJ, Balaji P, Feng W-C (2016) Multicl: enabling automatic scheduling for task-parallel workloads in OpenCL. Parallel Comput 58:37–55. https://doi.org/10.1016/j.parco.2016.05.006
40. Beri T, Bansal S, Kumar S (2017) The unicorn runtime: efficient distributed shared memory programming for hybrid CPU-GPU clusters. IEEE Trans Parallel Distrib Syst 28(5):1518–1534. https://doi.org/10.1109/TPDS.2016.2616314. (**Cited by: 9**)
41. Hofmann M, Kiesel R, Leichsenring D, Rünger G (2018) A hybrid cpu/gpu implementation of computationally intensive particle simulations using opencl. In: 17th IEEE International Symposium on Parallel and Distributed Computing, pp. 9–16