The Journal of Supercomputing (2023) 79:1065–1080
https://doi.org/10.1007/s11227-022-04671-5

Mashing load balancing algorithm to boost hybrid kernels in molecular dynamics simulations

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

Accepted: 20 June 2022 / Published online: 21 July 2022

Abstract
The path to the efficient exploitation of molecular dynamics simulators is strongly driven by the increasingly intensive use of accelerators. However, they suffer performance portability issues, making it necessary both to achieve technological combinations that allow taking advantage of each programming model and device, and to define more effective load distribution strategies that consider the simulation conditions. In this work, a new load balancing algorithm is presented, together with a set of optimizations to support hybrid co-execution in a runtime system for heterogeneous computing. The new extended design enables the exploitation of custom kernels and acceleration technologies altogether, being encapsulated for the rest of the runtime and its scheduling system. With this support, Mash algorithm allows to simultaneously leverage different workload distribution strategies, benefiting from the most advantageous one per device and technology. Experiments show that these proposals achieve an efficiency close to 0.90 and an energy efficiency improvement around 1.80 over the original optimized version.

Keywords Load balancing · Co-execution · Hybrid programming models · HPC · Molecular dynamics · OpenMP · OpenCL · C++ · CPU-GPU-MIC · Accelerators

1 Introduction

The heterogeneous architectures enables new ways of exploiting HPC problems, mainly due to their performance and energy efficiency properties. Molecular dynamics simulators are among the most relevant scientific softwares, being optimized for years, aiming to squeeze the multi-core architectures.

However, porting the software to accelerators reveals some performance portability issues. Technologies such as OpenCL cannot cope with highly optimized codes, being necessary to provide technology combination mechanisms to use the most

✉ Raúl Nozal
raul.nozal@unican.es

[1] Universidad de Cantabria, Santander, Spain

appropriate programming models for each case. Furthermore, an additional problem arises, since load balancing algorithms are not adequately adapted to the simulation conditions, wasting opportunities to exploit the heterogeneous system to solve the molecular dynamics computations.

There have been different works related to combining heterogeneous programming models and technologies [1–6], but they usually provide explicit code inputs, isolation of technologies by tasks, focus only on CPU-GPU distribution or use non-OpenCL-based languages. Some works focus on providing load distribution for HPC simulation environments [1, 7–13], but most focus on distributed technologies in combination with shared memory. And those that include accelerators are centered on host-device models or task-based parallelism. However, none of these works focuses on a combination of scheduling algorithms and the use of hybrid technologies to perform co-execution.

This work addresses the cooperative execution to solve molecular dynamics computations, relying for this purpose on three different architectures and considering a real simulator, ls1-MarDyn [14, 15]. Experimental validation shows that the runtime optimizations and its new native *execution core* improve energy efficiency by up to 2x with respect to the OpenCL version. When exploiting the whole system composed of CPU-GPU-MIC and the new Mash scheduler is applied with the hybrid co-execution mode, average improvements of 1.29x are obtained with respect to the next best balancing algorithm.

To overcome the above issues, the major contributions of this work include: (1) Mash, an algorithmic proposal to distribute the workload, applicable to the simulation conditions of ls1-MarDyn; (2) optimizations of the EngineCL runtime and its API to exploit co-execution using different acceleration technologies and programming models, providing a new hybrid co-execution mode along with a native *execution core*.;

The rest of the paper is organized as follows. Section 2 describes the motivation of this work, while Section 3 details the algorithmic proposal along with the optimizations performed to EngineCL. Then, after describing the API usage in Section 4, the methodology and experimental validation are exposed in Sections 5 and 6. Finally, Section 7 highlights the main conclusions.

2 Motivation

Computationally intensive scientific applications, such as the ls1-MarDyn molecular dynamics simulator, have generally been run in homogeneous multi-core clusters [16]. With the advancement of heterogeneous nodes and the popularization of accelerators, more efficient solutions are becoming available. However, they give rise to different main challenges: programming complexity, device performance portability issues as the programming model varies, and inefficiency in balancing between CPUs and accelerators.

Firstly, with the emergence of technologies such as OpenCL, the execution of kernels on these devices is possible, offering code portability but not always performance portability. This model represents a drastic change in the way of

programming these devices, placing them under a host-device paradigm. The main problem is that these are low-level languages a9756723nd APIs that make it difficult to be applied in complex software architectures, generally present in simulators. In addition, it is a multi-purpose objective, not only in the applicability and maintenance, but also guaranteeing performance portability, offering advantages over the original solutions.

Secondly, although these programming models are multi-architecture, a common practice is to use accelerators intensively and leave the CPU in charge of device management, work distribution and synchronization. This practice facilitates programming but involves a misuse of the energy consumed and a potential loss of performance, since the CPU could use that time to compute a region of work.During the period that the CPU is not computing, it is time that could be used to compute a region of work and has to be assigned to the accelerators. In addition, the static energy of the system continues to occur since the CPU, even if it is idle, keeps consuming in order to be able to operate. Thus, it is convenient to co-execute the problem, that is, to compute a kernel simultaneously by all the devices and computing units available in the system.

These previous drawbacks are addressed by using EngineCL, a high-level co-executor runtime for heterogeneous computation tested on multiple architectures. It offers a layered and optimized design that enables high usability without penalizing performance. However, two problems have been found to be solved, both related to performance portability. On the one hand, OpenCL technology is not always suitable for computing all types of problems. One of the key points of the performance of a device and the associated OpenCL programming model is determined by the quality of the driver and the optimizations performed by each vendor. This has been a serious problem encountered when working with the ls1-MarDyn simulator. The Intel Xeon processor requires a degree of optimizations not achievable by its driver regarding these molecular dynamics kernels, causing a performance penalty. Thus, this drawback not only penalizes the exploitation of the CPU, but also of the possibility of working cooperatively with any other device.

On the other hand, the existing algorithms do not fully benefit from this situation. It is necessary to leverage the best programming models and optimizations to solve the previous problem, independently of the major technology used in the runtime. Furthermore, it should be improved how the workload is distributed among the devices, as part of the simulation process. There are situations in which an algorithm performs better in one type of problem or device, and in other cases another one behaves much better. For example, an integrated GPU that supports compute-communication overlap via multiple queues, when faced with a program behavior like NBody, can benefit from algorithms that divide the load into many small packets [5, 17], while a discrete accelerator faced with the execution of many short-lived kernels generally cannot amortize the management overhead, and is better suited to algorithms that exploit very large packets [18–20]. For this reason, it is necessary to provide an appropriate and more sophisticated load balancing algorithm that take into account the context of the simulation and the runtime system. Hence, including these new strategies as an integral part of the process enable the heterogeneous

computing exploitation, benefiting from different types of architectures and leveraging the system resources, all with high usability for the simulator programmers.

3 Mash scheduler with EngineCL

This section details the algorithmic proposal to improve the execution of kernels with hybrid technologies. For this purpose, firstly, it is briefly presented the runtime system along with the changes introduced, in order to allow the co-execution when using a combination of computing technologies. Secondly, the simulator execution context and the fundamentals of the new load balancing algorithm are explained.

3.1 EngineCL optimizations

The optimizations focus on enhancing EngineCL [17] with more functionality, without compromising its usability and applicability. The runtime has experienced innovations with the main goal of providing support for hybrid heterogeneous computing model, which means combining different computing technologies for co-execution.

The runtime system has been modified to support two types of *execution cores*, that is, the computing technologies that are managed and exploited by EngineCL internally. This has required an internal transformation, including the generation of new interfaces to encapsulate the distinct implementations of its behavior. The first one makes it possible to continue executing with OpenCL technology, preserving the already validated functionality. However, a new *execution core* enables binary kernels on the CPU, as part of a native execution. In addition, to enable multiple devices to be used simultaneously, the software architecture has been extended to support hybrid co-execution, mixing native and OpenCL *execution cores*. Hence, the same kernel is computed simultaneously by two independent technologies, being EngineCL in charge of synchronization, workload distribution and resource management, regardless of the *execution cores* involved.

These enhancements decouple the OpenCL technology from the runtime. Figure 1 shows the compilation and execution model of EngineCL once the modifications have been made. At the top is the API for programmers, where they simply have to set the source and binary codes to the engine and the program to be executed, both from EngineCL Tier-1. Internally, these codes are processed by different modules, in order to prepare the provided kernels. By means of compilers and linkers, it is possible to encapsulate the kernels, containing more efficient and hand-optimized programming models, including pre-compiled binaries, as long as they preserve the signatures imposed by EngineCL. After all kernels are built as binary objects and normalized with a common internal specification, the runtime is able to orchestrate their execution, providing them to the execution cores, OpenCL and Native. Thanks to these abstractions and the usage of interfaces in the internal software architecture, it is possible to enable a new hybrid co-execution mode that allows reusing existing schedulers with both types of execution kernels.

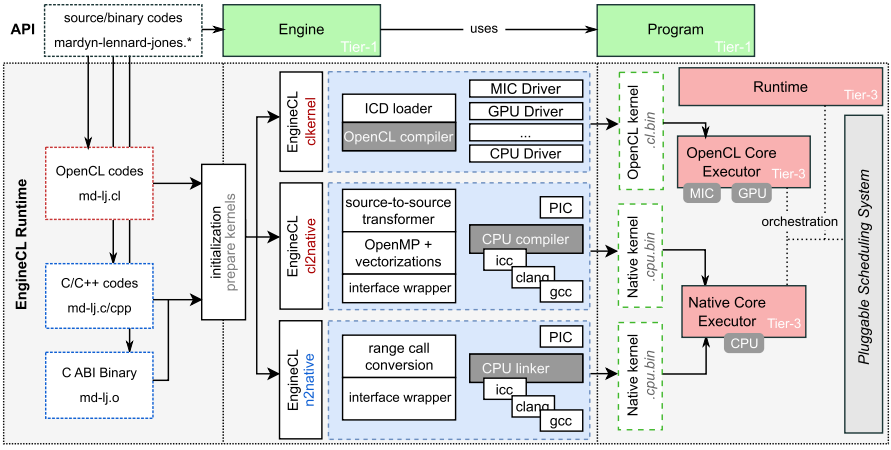

Fig. 1 EngineCL compilation-execution model showing the optimizations to enable different kernel origins via its new Core Executors functionality

3.2 Mash algorithm

As a result of the feasibility analysis when porting and co-executing ls1-MarDyn kernels with EngineCL, it has been shown that there is not a load balancing algorithm that is the best for all cases. After a detailed analysis of performance and co-execution, one of the main drawbacks of accelerators in this type of applications have been determined. In addition to the complexity of use that degrades performance portability, the time to offload kernels, data transfers and synchronization of packages are highlighted, being penalized in the face of adaptive strategies. For this reason, there are situations where a static algorithm may be desirable, and minimal management and overhead may be decisive for such situations. On the other hand, in more computationally intensive or molecule-intensive problems, where offloading compensates, adaptive algorithms and accelerators become more relevant.

For these reasons, a load balancing algorithm, called *Mash*, was devised. Its philosophy is based on the combination of load distribution strategies, but specifically designed to facilitate its use and solve the problems encountered in this type of simulations. In order to do this, it is first necessary to briefly present how the simulator is executed.

The simulation process is composed of `epochs`, and each of these, in `phases`, both being configurable in number, size and properties involved. However, at the beginning and at certain points of the simulation there are periods of setup and initialization, as well as secondary operations and *maintenance* simulation tasks. In addition to initialization and shutdown, it is in these other `phases` where the simulation slows down momentarily to save states, perform checkpoints, inter-node data migrations or plugin executions, among others. For this reason, the algorithm has small periods of time where it can benefit from runtime operations, performing certain tasks to obtain relevant information that can be used later.

To assist in the explanation of the algorithm *behavior*, Figure 2 is provided, where the simulator execution process is shown. It represents the *behavior* later

analyzed in Section 6, thereby involving three devices, the CPU and two accelerators, MIC and GPU. Above, horizontally, the molecular processing `stages` are shown, starting at *Stage 1*. The algorithm has three consecutive phases: profiling, setup and co-execution. The profiling analyzes the *behavior* of the devices. The setup establishes the appropriate parameters and load distribution mechanisms for the subsequent phase. And finally, the co-execution use all the devices in the system to compute the molecular sets, benefiting from configuration and profiling phases.

The *profiling* phase represented in the figure with a downward projection, which begins in the CPU of `Stage 1`, determine the *behavior* of the devices with respect to the number of work packages launched. There are as many profiling stages as devices are being used in the co-execution. However, each stage of the profiling phase is performed by a single device at a time.

Considering profiling `Stage 1`, performed by the CPU, it can be seen that it has 3 `steps`, identified as A, B and C to help in the explanation. In the first one, `Step A`, a work package is received with a given problem size W, which represents the iteration space, since the computation to be performed is of data-parallel style. In this `step`, the problem size is duplicated or divided, depending on the selected *behavior* by the simulator programmers.

By default or when the programmers indicate that the problem may present irregular *behavior*, the work package is duplicated, obtaining two packages that are identical. Irregular *behavior* represents those problems that for the same work size require different times, even when computed on the same device. However, if they choose a regular pattern, this profiling phase is slightly optimized, since the package is divided into two equal halves. In any case, at the end of `Step A`, two packages are available.

At the start of `Step B`, these two packages are assigned to two *offloading modes*, the single W_s and the multiple W_m. In this step, a consecutive execution is performed under these modes. Programmers have established two types of

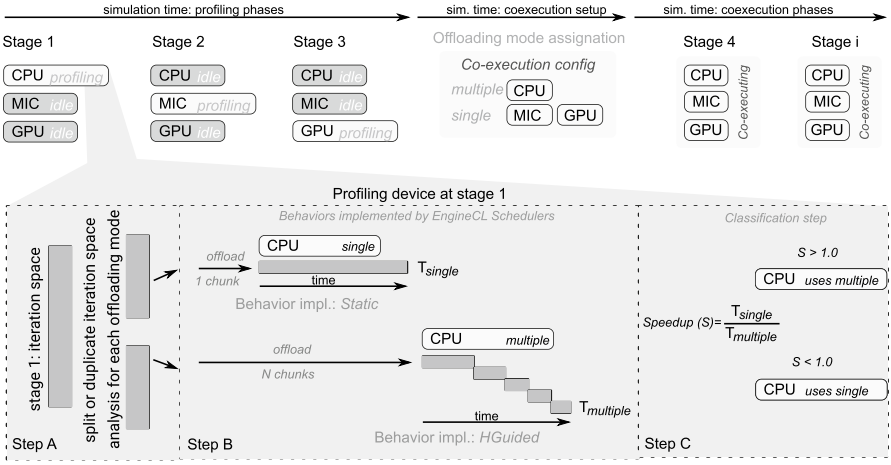

Fig. 2 Load balancing steps during the simulation: profiling, setup and co-execution phases

behaviors when using Mash, one when using single-chunk offloading, and another one involving multiple chunks. For example, out of the algorithms provided by EngineCL, the `Single` and `Static` schedulers fit in the former category, as well as `Dynamic` and `HGuided` as in the latter one. In any case, Mash only specifies that the *behaviors* have to fulfil these two offloading modes. It does not specify further restrictions or specific implementations. The figure shows how the complete package is first executed by performing the offload with a single chunk on the CPU, obtaining the time T_{single}. Once this has finished, the other package provided by the `Step A` is chunked. For example, the figure depicts a *behavior* with progressively smaller chunks. At the end, the time $T_{multiple}$ is obtained.

Then, the last step of the *profiling phase*, `Step C`, is performed. At this point, the acceleration between both offloading modes is computed, determining the best strategy to be used with the device being profiled:

$$S_{offloading\_mode} = \frac{T_{single}}{T_{multiple}}$$

Thus, values higher than 1.0 indicate that the device has a beneficial *behavior* when facing workload splitting strategies, allowing an increase in throughputs by taking advantage of multiple command queues, overlap between computation and communication as well as appropriate interleaving between management and computation, as demonstrated in previous studies [17, 19, 21–24]. And therefore, values lower than 1.0 indicate that it suffers penalization for device management and chunk synchronization, sharing of CPU usage with the simulator itself or other tasks and even an indication of very short execution times, where the generation of multiple chunks is usually counterproductive.

Mash allows setting a threshold to apply a default offloading mode, configurable by programmers. This prevents the application of an unfavorable offloading mode with respect to the other one, knowing that this could lead to severe load balancing problems. Thus, when the threshold is not exceeded, if irregular patterns have been indicated, the default mode is multiple. Otherwise, with regular ones, it is single. The default mode is considered if the calculated speedup is in the threshold range:

$$(1.0 - threshold) < S_{offloading\_mode} < (1.0 + threshold)$$

Once `Step C` is completed, the execution of `Stage 1` is finished. The process is then repeated for each of the remaining devices, but using the next `stages` of the simulation. In the figure, `Stage 2` and `Stage 3` are performing the MIC and GPU *profiling phases*, respectively.

The co-execution configuration phase is performed when the profiling is finished. This phase is really fast and does not require any simulation execution, since only the configuration of the *behaviors* assigned to Mash by the programmers is performed. That is, the same *behaviors* that were set for the profiling phase, `Static` and `HGuided`, for example, are now configured for the co-execution. This offloading mode assignment is done by device grouping, as

long as the *behavior* supports it, since, at a minimum, it has to support a single device. Figure 3 shows an example of assignation, using `HGuided` and `Static` as schedulers. From the best times obtained by each device in the profiling phase $T_{mode_{dev}}$, the throughputs th_{dev} are computed and the relative computational powers are obtained, by means of the following equation:

$$p_i = \left\lfloor \frac{th_i}{\sum_{j=1}^{n} th_j} \right\rfloor$$

Where n is the number of total devices, regardless of the offloading mode. Subsequently, the computational powers relative to each offloading mode *op* are configured, where the above equation is used again but with n being the number of devices involved in that offloading mode. In the example used, the GPU is the only one that requires single mode, so it is assigned the 100% of the 30.7% of the total problem (w). In contrast, the CPU and MIC use multiple mode, as it is the most effective, so the assignation of the computational powers are 66.6% and 33.3%, respectively. This is important, since Mash does not impose the scheduler that implements each *behavior* the size that should be assigned to each device, but the computational power, and it will be the scheduler that determines how it distributes the work. In the case of `Static`, by using only the GPU with 1.0 of computational power, it will generate a single chunk for it. On the other hand, with `HGuided`, the values 0.66 and 0.33 are used to divide the chunks progressively, but being a very efficient adaptive algorithm, they may end up delivering different work sizes. This is another advantage of Mash, since it does not impose to its internal algorithms how they should act, but performs the partitioning taking into account all the devices involved, based on throughputs and relative computational powers.

Finally, once the internal schedulers have been configured and the setup phase is finished, the co-execution phase itself begins. From this moment on, the following stages of the simulation will be computed according to the established configurations. It should be noted that the first two phases are usually executed in those times of secondary operations indicated above, in order to avoid penalizing the simulation. Even so, the work to be done is duplicated, as long as an irregular pattern has been indicated, during as many phases as there are devices, so that it is a negligible proportion with respect to the complete simulation. The advantage of these periods of maintenance and configuration of the simulator itself is that they can be used to perform profiling again without the minimum cost for the simulation. In this way, periodic profiling can be performed, both every certain number of computed stages,

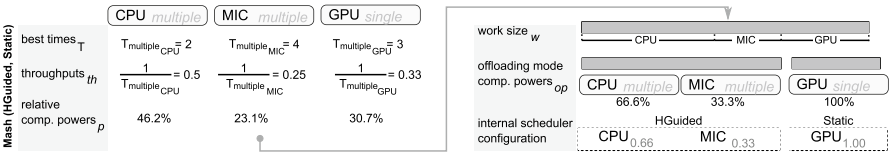

Fig. 3 Mash co-execution configuration phase using HGuided (multiple) and Static (single) as mashing schedulers

and every time Mash is notified to recompute the profiling and configuration phase, so that in addition to the adaptive algorithms used internally, a higher level adaptive adjustment phase is performed.

4 Programmability and API usage

Listing 1 shows an example of the EngineCL API to compute LennardJones potentials for sets of molecules. The novelty is that it exploits the new load balancing algorithm, Mash, presented in Section 3.2, while using different kernel sources. It shows one of the computations performed in the experiments of Section 6. It uses the CPU with the new native *execution core* described in Section 3, and two accelerators, MIC and GPU, via OpenCL technology. To enable cooperation between these devices, the new hybrid co-execution mode is employed, transparently for the programmers.

Listing 1 EngineCL API using the Mash algorithm with the hybrid co-execution mode to compute LennardJones potentials.

```
using ecl::DeviceMask; using ecl::Device; using ecl::io;
using ecl::Scheduler; using ecl::Scheduler::MashOpt;
ecl::EngineCL engine;
ecl::Program program;

void hook_setup_simulation(/* simulation context */){
  /* binary and source code file readers */
  auto kernel = io::file_read("mardyn-lennard-jones.cl");
  auto phi_kernel = io::file_read("md-lj.phi.cl.bin");
  engine.use({ DeviceMask::CPU,
  Device::BlobLoader("md-lj.hand-opt-cpu.bin") },
  Device(2, 1, phi_kernel), Device(1, 0));
  engine.scheduler(Scheduler::Mash(
  MashOpt::Profiler(MashOpt::Irregular),
  /* scheduling mashing:        Multiple,              Single */
  MashOpt::Behaviors(Scheduler::HGuided, Scheduler::Static)));
  program.kernel(kernel, "LennardJonesPot");
}

void stage_simulation_iterator(md_flt4* in_vel, md_dbl4* out_pt
/* + simulation ctx & vars */){
  /* omitted for brevity: pos-vel init, mesh, fields and pots
     */
  auto gws = molecules; auto lws = 64;
  ljpot_setup(molecules, /* ... + args */ in_vel, out_pt);
  engine.work_items(gws, lws);

  program.in(in_vel);
  /* rest of the application domain arguments */
  program.out(out_pt);
  program.args(in_vel, /* ..., */ out_pt, molecules);
  engine.program(&program);
  engine.run();
}
```

The listing is divided into three sections, from top to bottom. The parent scope, where the two main Tier-1 API classes, `engine` and `program`, is declared in lines

3-4 (*L3-4*). Subsequently, two functions to be called during the simulation process are defined, `hook_setup_simulation` (*hss*) and `stage_simulation_iterator` (*ssi*). Both functions provide access to the simulator context, where the rest of the simulation parameters and other modules of its software architecture can be accessed, such as the plugin system, the safeguard checkpoints, profiling information or the execution epochs.

The *hss* function is called during the initialization stages of the simulator, taking advantage of the hook ports to configure the runtime and to initialize structures and technologies. After reading the source and binary codes (*L7-8*), the `engine` is cued to use the native *execution core* by means of a special binary kernel load helper (*L9-12*). In addition, two other devices are used, the GPU with the generic kernel and the Xeon Phi with a specialized binary kernel (*L13,14*). Subsequently, the `Mash` scheduler is assigned to the runtime (*L15-19*). To do this, two parameters are set. First, the type of profiler to be used, indicating that it is a kernel with irregular pattern (*L16*). Second, the behaviors to be associated to the mashing, both when choosing the multiple and the single offloads, commented in line 17 to show the assignation order. In this case, the programmer has selected to associate them to the schedulers provided by EngineCL, `HGuided` and `Static` (*L18*). Finally, the program is configured with the generic kernel, indicating the input function to compute (*L20*). It is worth mentioning that programmers have the periodic profiler at their disposal, and they would only have to change line 16 to, for example, `MashOpt::PeriodicProfiler(MashOpt::Irregular, 20)`, to adjust the performances of each device every 20 stages of the simulation.

On the other hand, the *ssi* function is launched for each stage of the simulation, providing the execution ranges, variables and primitives needed to compute. To simplify the example, only two variables are listed, being of packed floating point types, defined by the simulator. After setting the kernel parameters and properly configuring what is needed to compute the set of molecules (*L22-26*), the execution range is provided to the runtime (*L27*), as well as the input (*L28*), output (*L30*) and application domain parameters (*L31*). This is, in EngineCL terminology, the application to be computed, isolated from the execution engine. Finally, the `program` is assigned to the `engine`, and the co-execution of the work supplied by the stage is performed (*L32,33*). In this way, this function will continue to execute as the simulation progresses, and the runtime itself will be in charge of internally managing the *profiling*, *assignment* and *co-execution phases*.

The proposal presented in this work has only modified the external API in two regions, the utilization of the native execution core for the CPU (*L9-12*), as well as the incorporation of a more sophisticated type of scheduler (*L15-19*), which is a composite type. The rest of the regions and forms of use are maintained, preserving the advantages of the originally validated API design.

5 Methodology

The experiments are carried out on a computer composed of an Intel Xeon E5-2620 with 24 threads, an AMD Rx5700XT GPU with 40 compute units and an Intel Xeon Phi 7120P Knights Corner Many-Integrated Core (MIC) with 240 compute units. The first technology involved is the current ls1-MarDyn implementation, labeled *CPU-icc*. It is parallelized with OpenMP, vectorized and compiled with the Intel compiler. *MIC*, *GPU* and *CPU-ocl*when OpenCL drivers are used for the Xeon Phi MIC, Rx5700XT GPU and Xeon CPU devices. Finally,the new hybrid mode and its native *execution core* for the CPU, labeled *CPU-hy*.

Five kernels related to the computation of particles and their interactions have been selected as part of the computational core of ls1-MarDyn [15]. Two of them, *md_dist* and *md_distn2* are related to the computation of distances between molecules. The former offers a flow-based interaction with low computational load, while the latter performs calculations based on indirections over all cells. The program *md_diststar* handles the minimum image convention while computing the distance between molecules. Finally, *md_bin* computes the associated indices for a set of cells in streaming mode, while *md_lj* obtains the potential and evaluates the force for the Lennard Jones 12-6 potential.

The validation of the proposal is performed taking into account two simulation scopes. On the one hand, contrasting the behavior of the new *execution core* with respect to the current mode used for CPU (*CPU-icc*), performing a complete offload of the work for each device. On the other hand, taking into account the whole heterogeneous system, so that all devices cooperate to solve the problem simultaneously, demonstrating the impact of the new load balancing algorithm as well as the hybrid co-execution mode, using both *execution cores CPU-ocl* and *CPU-hy*. In addition to the Mash algorithm (*Mh*), two scheduling algorithms included in EngineCL are evaluated, since these stood out for the kernels studied, being HGuided (*Hg*) and Static (*St*).

In both scopes the total response time is measured. Then, two metrics are used to evaluate the coexecution, heterogeneous efficiency (efficiency, henceforth) and energy efficiency. The efficiency is obtained from the speedups and the maximum speedup. The speedup is calculated as $S = \frac{T_{CPU\text{-}icc}}{T_{co\text{-}exec}}$, being $T_{CPU\text{-}icc}$ and $T_{co\text{-}exec}$ the execution times for the current CPU implementation and the coexecution, respectively. Due to the heterogeneity of the system and the different behavior of the kernels, the maximum achievable speedups depend on each program These values are computedas follows: Then, the efficiency, in Equation 3, is computed as the ratio between the empirically obtained speedup and the maximum achievable speedup, for each program [19].

Finally, energies are measured using the *sauna* tool along with RAPL counters and *sysfs* system drivers, giving the total consumption in *Joules*. The Energy-Delay Product (EDP) combines performance (time) and energy (power consumption), being used to evaluate the energy efficiency, measured in *Js*.

$$S = \frac{T_{CPU\text{-}icc}}{T_{co\text{-}exec}} \tag{1}$$

$$S_{max} = T_{CPU_{icc}} \sum_{i=1}^{n} \frac{1}{T_i} \tag{2}$$

$$HE = \frac{S}{S_{max}} \tag{3}$$

6 Validation

Two types of experiments are carried out. The first evaluates the scalability of each device and the benefits offered by the new native execution core and its optimizations. The second shows the impact of the Mash algorithm when leveraging all devices, using only OpenCL and the hybrid co-execution mode.

The execution times when using a single device to compute the whole problem are depicted in Fig. 4, showing how each device scales as the problem size is increased. Hence, the granularity of execution is increased, producing simulation tasks that are more computationally intensive. In this way, each technology and device is evaluated to compute a set of molecules, thereby comprising one of the stages of the simulation.

For all the kernels, the *CPU-ocl* obtains the worst results, making it pointless to use OpenCL on the CPU. These results are so poor that it limits the co-execution, penalizing the runtime management itself and preventing it from being competitive with respect to the *CPU-icc* version.

However, the new execution core offers very similar performance to the CPU-optimized ls1-MarDyn version, as shown by *CPU-hy* and *CPU-icc*. On the other hand, the GPU obtains computation times close to these last two CPU modes, although being slightly slower except in the case of the *md_distn2* kernel. It computes 2.64 times faster than the best version of the CPU, when calculating the distances between one million molecules. On the other hand, the MIC has worse performance than the GPU, only outperforming the CPU versions when using *md_distn2* with large problem sizes. It is a device that benefits from the use of multiple queues as well as dynamic scheduling algorithms, so it is limited when using the host-device offloading model to compute a single large work package. These kernels are highly optimized for the CPU, taking advantage of the memory hierarchy and vectorizations. Thus, the GPU, despite being several generations newer than the CPU and offering many more cores, is not the fastest device, as has been the case in many other classical kernels. The MIC, moreover, is using OpenCL, so it could suffer from the same drawbacks as the *CPU-ocl* when it comes to efficiently vectorizing these molecular dynamics kernels.

For now on, the results obtained for the complete heterogeneous system (CPU, GPU and MIC) are analyzed. Figure 5 shows the heterogeneous efficiency values obtained, for all benchmarks and the three load balancing algorithms evaluated, Static, HGuided and Mash, together with the geometric mean for each of them. A clear conclusion is that the hybrid model greatly outperforms the OpenCL model on CPU, with an average gain of around 20%.

Taking into account the geomean values, it can be stated that Mash is the one that best exploits the computational capacity of the system, with an average efficiency close to 0.9, which translates into a speedup of 1.9, followed by Static and finally HGuided, although both of these with very close values.

Analyzing the performance of each benchmark in more detail, it can be seen that in all cases the Mash algorithm is the best, except for the md_distn2 kernel, which is slightly outperformed by HGuided. However, HGuided offers very poor efficiency values in some kernels, such as md_bin. This is because these kernels execute Stages of small molecular sets, making the execution shorter, so that it is not able to adapt correctly in such interval, being penalized by its management and distribution overhead. Finally, the Static algorithm offers more stable values, but is by no means the best. These efficiency values allow the performance of the heterogeneous system to be superior to the best version of the CPU, whenever the hybrid model and the Mash algorithm are used. The average speedup value is 1.90, reaching a value of up to 5.03 in md_distn2, which is the one that needs more execution time.

It is important to note that in md_distn2, Mash has not managed to impose itself as the best since in the profiling phase the MIC was set with a single offloading mode, resulting in a high CPU occupation. This is due to the OpenCL driver of the MIC, as it requires high management, penalizing the computation of the CPU device when there is no possibility to make trade-offs in CPU usage. In the rest of the cases, the MIC and the CPU work using the multiple offloading mode, speeding up the execution notably.

Finally, a very important metric is the Energy Efficiency, presented in Fig. 6. It is studied through the EDP improvement over the baseline, with respect to the heterogeneous system. Values above 1.0 indicate better energy efficiency. From

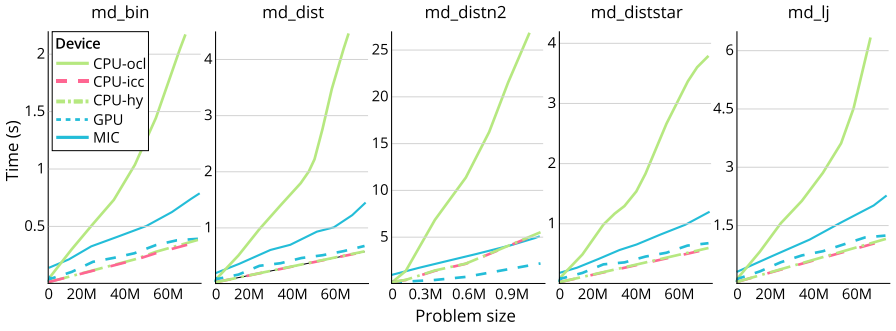

Fig. 4 Scalability when launching the whole kernel computation in a single device

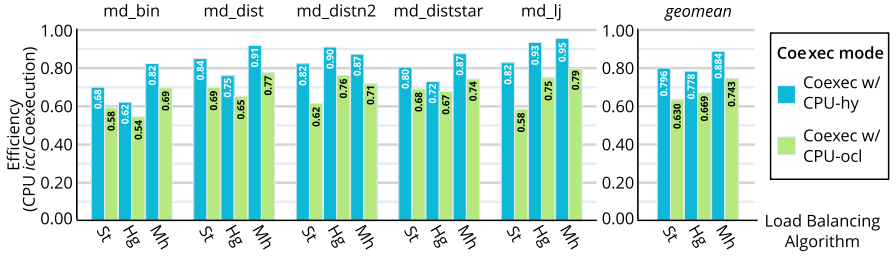

Fig. 5 Heterogeneous Efficiency of the heterogeneous system vs. CPU execution

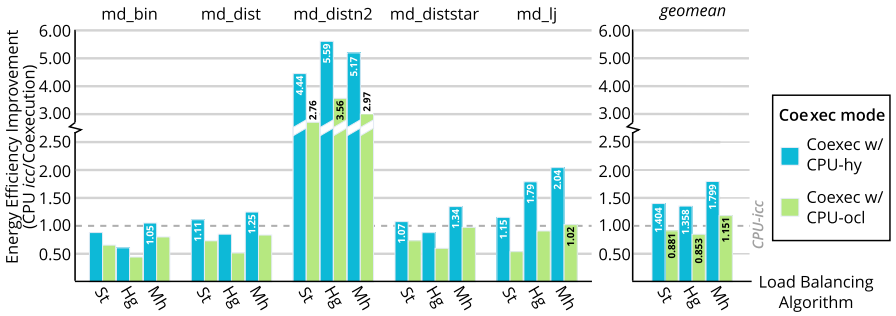

Fig. 6 Energy Efficiency of the heterogeneous system vs. the CPU execution

these results, it is important to highlight that, for all the benchmarks analyzed, it is necessary to use the two new proposals of this article, the hybrid model and the Mash algorithm, in order to achieve high energy-efficient co-execution in the heterogeneous system.

Otherwise, it can be seen that there is a full correlation between the performance results, shown in Fig. 5, and the energy efficiency results, presented in Fig. 6. For this reason, with respect to energy efficiency, the same conclusions already mentioned for the case of performance can be confirmed.

In summary, the results presented show two clear conclusions. Firstly, that the Hybrid model is clearly superior to the model using OpenCL in both performance and energy efficiency. Secondly, that the Mash algorithm is the one that best exploits in most cases both the computational capacity, obtaining the best performance, but also the energy efficiency of the heterogeneous system.

7 Conclusions

This paper proposes a novel load balancing algorithm, Mash, to exploit heterogeneous execution in molecular dynamics simulators, such as ls1-MarDyn. This algorithm allows taking advantage of known behaviors of existing algorithms and include them

as part of its internal mash, but favoring a higher level adaptive phase and with high usability. For this purpose, it has been implemented in EngineCL, a runtime system for heterogeneous computing.

The optimizations made allow the use of different parallel programming technologies and hand-optimized kernels, all of them in cooperation with the different schedulers provided, enabling support for hybrid co-execution. This, together with the new scheduling algorithm evaluated in the simulation, allows reaching average speedups of 1.90, efficiencies up to 0.95 and energy efficiency improvements close to 1.80 with respect to the original optimized version.

Acknowledgements This work has been supported by the Spanish Ministry of Education (FPU16/03299 grant), the Spanish Science and Technology Commission under contract PID2019-105660RB-C22 and performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action (H2020). The author gratefully acknowledges the support of the SPMT group, part of HLRS.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availability the datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

References

1. Luk C-K, Hong S, Kim H (2009) Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 45–55
2. Ravi VT, Ma W, Chiu D, Agrawal G. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In: Proceedings of the 24th ACM Int. Conference on Supercomputing, pp. 137–146
3. Gummaraju J, Sander B, et al. (2010) Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: 19th Int. Conference on Parallel Architectures and Compilation Techniques (PACT)
4. Ding H, Huang M (2014) A unified opencl-flavor programming model with scalable hybrid hardware platform on fpgas. In: Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig14), pp. 1–7
5. Nozal R, Bosque JL (2021) Exploiting co-execution with oneapi: heterogeneity from a modern perspective. In: European Conference on Parallel Processing, pp. 501–516
6. Scogland T et al. (2012) Heterogeneous task scheduling for accelerated openmp. In: 2012 IEEE 26th Int. Parallel and Distributed Processing Symposium, pp. 144–155
7. Hamidouche K, Falcou J, Etiemble D (2010) Hybrid bulk synchronous parallelism library for clustered smp architectures. In: Proceedings of the 4th Int. Workshop on High-level Parallel Programming and Applications, pp. 55–62

8. Kylasa SB et al. (2016) Reactive molecular dynamics on parallel heterogeneous architectures. IEEE Transactions on Parallel and Distributed Systems, 202–214
9. Nozal R, Bosque JL (2021) Straightforward heterogeneous computing with the oneapi coexecutor runtime. Electronics 10(19):2386
10. Bergen B K et al. (2010) A hybrid programming model for compressible gas dynamics using opencl. In: 2010 39th Int. Conf. on Parallel Processing Workshops, pp. 397–404
11. LaKomski D et al. (2015) Optimal balance between energy and performance in hybrid computing applications. In: 6th Int. Green and Sustainable Comp. Conf., pp. 1–8
12. Rabee F, Liao Y, Yang M, Liu J, Zhu G (2014) Global hybrid multi-core-gpus-openmps-resources platform in hard real time system. In: IEEE 17th Int. Conf. on Computational Science and Engineering, pp. 845–850
13. Feng, Jie et al. (2022) Heterogeneous computation and resource allocation for wireless powered federated edge learning systems. IEEE Trans Commun
14. Seckler S textitet al. (2021) ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning. J Comput Sci **50**
15. Nozal R, Niethammer C, Gracia J, Bosque JL (2022) Feasibility study of Molecular Dynamics kernels exploitation using EngineCL. In: Euro-Par 2021: Parallel Processing Workshops
16. Tchipev NP (2020) Algorithmic and implementational optimizations of molecular dynamics simulations for process engineering. PhD thesis, Technische Universität München
17. Nozal R et al. (2020) Enginecl: Usability and performance in heterogeneous computing. Future Gen Comput Syst, pp. 522–537
18. Dávila Guzmán M A et al. (2019) Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. J Supercomput 75(3), 1732–1746
19. Nozal R, Bosque JL, Beivide R (2019) Towards co-execution on commodity heterogeneous systems: Optimizations for time-constrained scenarios. In: 2019 Int. Conference on High Performance Computing & Simulation (HPCS), pp. 628–635
20. Rodriguez-Canal Gabriel et al. (2021) Efficient heterogeneous programming with fpgas using the controller model. J Supercomput 77(12), 13995–14010
21. Morenń K, Göhringer D (2022) Graphcl: A framework for execution of data-flow graphs on multi-device platforms. In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 116–121
22. Nozal R (2022) Optimizing Performance and Energy Efficiency in Massively Parallel Systems. PhD thesis, Universidad de Cantabria
23. Moreton-Fernandez A, Gonzalez-Escribano A, Llanos DR (2019) Multi-device controllers: a library to simplify parallel heterogeneous programming. Int J Parallel Program 47(1):94–113
24. Jääskeläinen P, Korhonen V, Koskela M et al (2019) Exploiting task parallelism with opencl: a case study. J Signal Process Syst 91(1):33–46