Performance portability of generated cardiac simulation kernels through automatic dimensioning and load balancing on heterogeneous nodes

Vincent Alba, Olivier Aumage, Denis Barthou, Raphaël Colin, Marie-Christine Counilh, Stéphane Genaud, Amina Guermouche, Vincent Loechner, Arun Thangamani

Performance portability of generated cardiac simulation kernels through automatic dimensioning and load balancing on heterogeneous nodes

Vincent Alba*, Olivier Aumage*, Denis Barthou†, Raphaël Colin‡, Marie-Christine Counilh*, Stéphane Genaud‡, Amina Guermouche*, Vincent Loechner‡, Arun Thangamani‡

* University of Bordeaux, CNRS, Bordeaux INP, Inria, LaBRI, Talence, France

† Bordeaux INP, Talence, France

‡ ICube Lab., University of Strasbourg, CNRS, Inria, Strasbourg, France

***Abstract*—Electrophysiology simulation applications, such as the community-developed OPENCARP framework for in-silico experiments, involve applying a broad range of ionic model kernels with different computational weights and arithmetic intensity characteristics. Efficiently processing such kernels on modern heterogeneous architectures necessitates to accurately dimension the set of computing resources to use and to actively balance the load on the available computing units, to account for discrepancies in kernel duration and distinct computing unit speeds. We thus propose the following contributions: 1) the adaptation of an existing load-balancing algorithm to transparently manage the mapping of these ionic model kernels onto the heterogeneous units of a computing node; 2) a resource dimensioning heuristic that constraints the number of devices that should be used to maximize efficiency, according to the selected ionic models' computational weight; 3) the integration of these mechanisms in OPENCARP, building on prior work that took advantage of LLVM's MLIR framework to generate multiple device-specialized variants of kernels from ionic models expressed in OPENCARP's high-level DSL; 4) a thorough experimentation of the mechanisms on a comprehensive series of 30 ionic models provided by OPENCARP. The experiments show that when using the combination of the load-balancing algorithm and the resource dimensioning heuristic to compute each ionic model, the geometric mean of speedup is $9.97\times$ with respect to the original multi-threaded code on an architecture with two A100 GPUs and $2\times$ 32-cores AMD Zen3 CPUs.**

***Index Terms*—Heterogeneous architecture, dynamic load-balancing, resource selection, task parallelism, task aggregation**

I. INTRODUCTION

Cardiac electrophysiology is a medical specialty in which the research community has long been using computational simulation. Understanding the heart beat behavior (and in particular cardiac diseases involving arrhythmia) requires to model the ionic flows between the muscular cells of cardiac tissue. Such models, called *ionic models*, describe the way an electric current flows through the heart cell membranes.

The widespread practice in this field is for experts to describe their ionic model in a domain-specific language (DSL), which essentially enables to model the current flow by ordinary differential equations. The OPENCARP[1] [19] simulation framework has been created to promote the sharing of the cardiac simulation efforts from the electrophysiology community. To describe ionic models, it offers a DSL named EasyML [24], from which a code generator can derive `C/C++`.

The next major advances in cardiac research will require to increase by several orders of magnitude the number of cardiac cells that are simulated. The ultimate goal is to simulate the whole human heart at the cell level [20], that will run several thousands of time steps on a mesh of several thousands of billions elements.

In its initial version, the OPENCARP code generator could produce `C/C++` code suited for CPU and parallelize the computations using OpenMP. However, simulations at large scale require to efficiently exploit all the capabilities of modern supercomputers. In order to tackle the heterogeneous nature of today's supercomputers, recent work on OPENCARP has improved the original code generator with efficient SIMD code generation for multicore CPUs [22] and for GPU [23]. Although this work has brought significant performance enhancement, it is limited to target a specific accelerator at code generation time. Thus, in order to introduce a level of abstraction with respect to the hardware architecture, we use the STARPU task-based runtime system [1] as the foundation to build the heterogeneity-enabled OPENCARP to make the kernel launches and data transfers device-independent for the application. Our first contribution was therefore to augment the code generator [23] to generate systematically both vectorized multicore code and GPU code, with STARPU runtime code.

Running computations on a heterogeneous architecture may however introduce load imbalance. As a matter of fact, all computing units may not get the same amount of work. Furthermore, due to the lightweight nature of OPENCARP ionic models kernels and the wave-oriented layout of their execution, instead of a pure runtime approach balancing tasks among devices such as STARPU, we propose an hybrid

[1] https://opencarp.org

compiler/runtime approach. Thus, we adapted the load balancing algorithm proposed by Huchant *et al.* [12], [13] to match OPENCARP needs. At each computation iteration, this algorithm adapts the workload chunk size given to each device (if needed) according to the execution time of the previous iteration. Moreover, since the OPENCARP ionic models exhibit a variety of computational requirements, we developed an algorithm that adapts the number of used devices, depending on the actual needs of the selected ionic model.

The contributions of this paper are as follows:

- automatic generation of StarPU code (as *codelets*),
- implementation of a load-balancing algorithm,
- automatic resource dimensioning algorithm,
- integration of all of the above in a real use-case application (OPENCARP).

We implemented and evaluated our proposal in a real large-scale code, on 30 different ionic models in OPENCARP, on up to 8 GPUs per node. Depending on the ionic model, the performance gain on multi-GPU architectures reaches up to $60\times$ speedup, with a geometric mean speedup of $13\times$ compared to the multicore CPU vectorized and parallel version. On an hybrid architecture, the geometric mean speedup reaches $9.97\times$.

The paper is organized as follows: Section II presents the OPENCARP application. Sections III and IV describe the main contributions of the paper. Section V presents experiment results on real use-cases. Section VI discusses related works, and Section VII concludes the paper.

II. OPENCARP ARCHITECTURE

OPENCARP is an open cardiac electrophysiology simulator for in-silico experiments. It offers single-cell as well as multiscale simulations from the ion channel level to the whole organ level. OPENCARP uses a mathematical-oriented domain-specific language (DSL), called *EasyML*, that provides a convenient way for cardiology scientists to describe ionic models in a natural formalism. These ionic models are at the core of the simulation. They model the ion transfers through the cells membranes that result in the heart contraction.

Each step of the main OPENCARP simulation time-loop is composed of two stages: (i) the computation of the electrical potential on the membrane using a solver and (ii) the computation of the ion flow that crosses the membrane. While the former can be executed in a distributed manner (involving several compute nodes), the latter is always run on a single node (that can be heterogeneous). In this paper we focus on the ionic models (the second stage), that are used to compute the current that flows in and out of each cell. These models are accessed through the *LIMPET* (Library of Ionic Models & Plug-ins for Electrophysiological Theorization) library, which defines a common interface to them. The work presented in this paper more specifically focuses on this library.

To run experiments using LIMPET without any call to the solver part, we use *bench*, a tool provided by OPENCARP. Most of the computations of the *bench* execution is contained inside a main loop. This loop revolves around a structure called `timer_manager`, which handles the time discretization into timesteps.

Inside the main loop, at each timestep, two types of computations are repeated :

- the computation of the membranes ion flow using *LIMPET*'s ionic models,
- additional operations (e.g stimuli, I/O operations), known as *events*.

Events are stored in the `timer_manager`: they are known at the start of the execution and are characterized by the timestep ts when the event occurs and its duration d in timesteps. At each timestep ts', the main loop calls the `timer_manager` during the event handling phase to check if any of the stored events should occur at ts'. If so, it will execute the function associated to each event that will run during ts'. Note that events can last more than a single timestep, in which case they are called *persistent events*.

Listing 1. Pseudo-code of the main loop of the bench executable

```
1  for timestep in time_manager(){
2      event_phase()
3      model->compute()
4  }
5
6  function model::compute(){
7      for I in mesh_domains{
8          // ionic computation
9      }
10 }
```

For a given timestep, as seen in Listing 1, the main loop first computes the events (if any) and then runs the ionic computation. The ionic computation has its own loop that iterates over a list of *mesh elements*: these elements are an abstraction of either one or multiple cells or even sub-cellular states. The loop runs the ionic model compute function over each *mesh element*, and is referred to as a *kernel*.

III. RUNNING OPENCARP ON HETEROGENEOUS NODES

The exploitation of simulation applications such as OPENCARP on modern heterogeneous computing nodes requires to be able to generate specialized kernels for the target computing units and to manage the efficient execution of these kernels on these units. Task parallelism is a natural way to achieve this goal, as it brings the necessary flexibility in handling kernel specialization and work mapping on such architectures. We describe below how we rely on the LLVM MLIR framework [16] to generate the kernel specializations for each target device (Section III-A), and on specially tailored granularity management techniques to achieve load balancing on heterogeneous computing units (Section III-B).

A. Compiling OPENCARP kernels for heterogeneous nodes

OPENCARP uses EasyML as a DSL to represent diverse ionic models. The models are described as mathematical equations. EasyML can be translated from/to many other equivalent languages used to describe ionic models in this community (CellML, SBML, MMT). OPENCARP's LIMPET library provides an EasyML compiler through the `limpet_fe` python

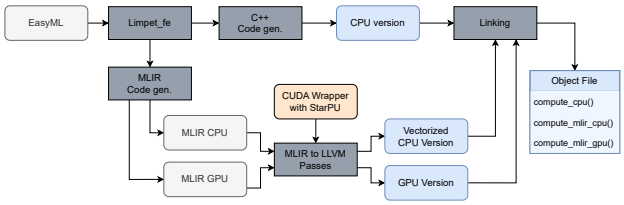

Fig. 1. Compilation steps and StarPU integration

script. Figure 1 represents an overview of the code generation using `limpet_fe` on an EasyML model.

The original `limpet_fe` code could only produce the CPU version with OpenMP parallelization on the mesh elements loop of the kernel. Recent works used MLIR to develop LimpetMLIR [23], an extension of `limpet_fe` able to generate both efficient vectorized OpenMP CPU code (for SSE, AVX2, and AVX512 vector units) and GPU code (using CUDA for Nvidia GPUs and ROCm for AMD GPUs). LimpetMLIR allows to compile EasyML models for different hardware to aim for higher performance.

During compilation, LimpetMLIR produces multiple versions of each ionic model compute function:

- CPU: the baseline version, runs on CPU, with basic compiler optimization, plus OpenMP multi-threading.
- MLIR-CPU: efficient SIMD version for CPU, with MLIR vectorized mesh elements computation, plus OpenMP multi-threading.
- MLIR-CUDA: MLIR generated GPU version for Nvidia GPU.
- MLIR-ROCm: MLIR generated GPU version for AMD GPU.

In this paper, we will refer to those different versions as *targets*. By default, LimpetMLIR will compile all of the available targets on the system (i.e. only compile the MLIR-CUDA target when Nvidia GPUs are available). Each of these kernels are then linked to different private functions inside each model's object file. To adapt to the StarPU execution environment, we wrote wrapper functions to those target-specific kernels, to be able to select the target dynamically as StarPU codelets.

For our implementation, we also need to control the GPUs: we use StarPU to manage CPU-GPU data transfers, to handle GPU streams and to select on which CPU or GPU a kernel will run. To address this we modified the CUDA and ROCm wrappers generated by LimpetMLIR to let the StarPU runtime handle those aspects.

B. Executing openCARP on heterogeneous nodes

Once the kernel versions have been compiled, they need to be run onto the available computing units. This process is particularly challenging for the various ionic model kernels of openCARP. Indeed, the models exhibit widely distinct characteristics regarding their computational weight and arithmetic intensity, resulting in varying relative efficiencies on

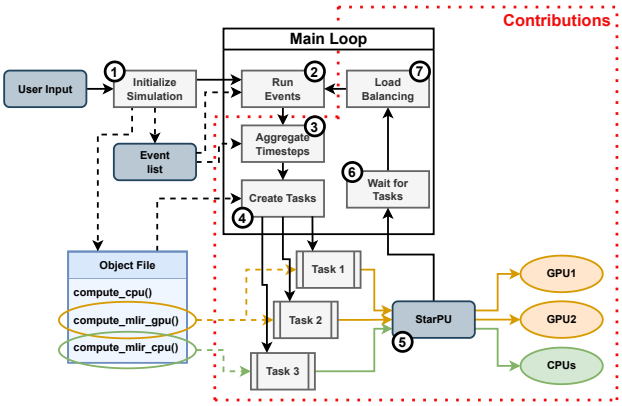

Fig. 2. Execution flow of bench

accelerated units and on CPU cores. These behavioral differences incur load imbalance and resource under-subscription. Note that the execution of a kernel handles independent mesh elements, that can thus be freely reordered.

To address this challenge, we devised the use of a task-based parallel programming approach based on the StarPU runtime system. However, the computing cost of individual ionic kernel calls is often too lightweight for each call to be wrapped in its own StarPU task. This is a typical problem with simulation applications where the natural expression of the problem does not reach a sufficient grain to efficiently use the underlying architecture and compensate the runtime overhead [21]. As a matter of fact, the minimal StarPU task granularity is $\approx 100\mu$s while the duration of a timestep kernel can be as low as $\approx 11\mu$s. We therefore developed an aggregation stage that merges the computation of several consecutive timesteps by taking advantage of the opportunities offered by the simulation characteristics. We then integrated a heterogeneity-aware load-balancing algorithm to map the work on the available CPU and GPU units. Yet, even with such an aggregation stage, the resulting computational weight of each task may be too low for some of the ionic models to justify the use of all the computing units on platforms with multiple accelerators per node, especially with respect to the data transfer costs. We thus designed a heuristic to automatically constrain the set of targeted computing units to maintain a high level of efficiency.

IV. Parallel Execution

In this section, we detail our contributions to openCARP regarding load balancing, data partitioning and resource selection.

A quick overview of the parallel execution of `bench` is described on Figure 2. First, during the initialization (1) the program recovers the user-given parameter, the ionic models and events to be computed. Then, for every timestep, (2) we run the available events, if there are any. After that, (3) we group, using the event schedule in the `timer_manager`, all timesteps between two events in a single task. Next, (4) we create the next batch of tasks using the different implementations of the ionic model (at the beginning, the mesh

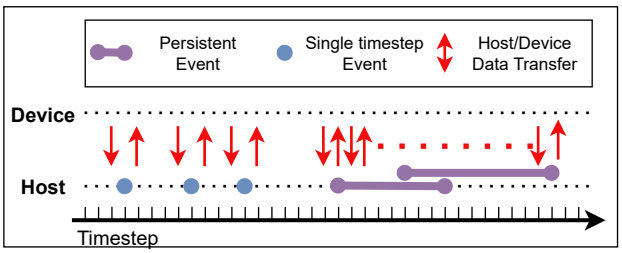

Fig. 3. Host-device data transfer for the global data vectors depending on the events. During persistent events, data is transferred at each timestep: the figure only shows 3 couples of transfers but the rest is represented as "..."

elements are distributed equally between each task). Then (5) the tasks are executed on the different devices with their data. (6) The main loop will wait for each task to finish and then, on hybrid executions, if there is a significant load imbalance between CPUs and GPUs, (7) compute a new load distribution from the execution time of the current task batch. This process is repeated until all timesteps are computed.

A. Data Management

As described in Listing 1, the ionic model computation consists of running the kernel loop on every mesh element of a mesh element list L_m of size nb. In the code, L_m is implemented as multiple data parallel vectors of size nb in a similar fashion to a structure of array. Each mesh element has an entry to two types of vectors:

- The *state variable* vector which stores all data that is specific to the ionic model. This vector is unique and is never used outside of the ionic model.
- The *global* data vectors which store data that are used both inside and outside the ionic models computation. They are used, for example, in most events. Note that there are several instances of global data vectors.

When running in parallel, the mesh elements have to be distributed across the different computing units. As a consequence, when running on a hybrid architecture, since the state variable vector is local to an ionic model, the only moment where data from this vector may be transferred is during the mesh distribution phase. On the other hand, when computing any mesh element on GPU, the global data vectors require a data transfer each time an event is triggered since events are always executed on CPU.

Figure 3 shows how events incur data transfers on GPU-based execution. Since ionic model computation modifies the global data vectors at every timestep, any data partition that is on GPU needs to be transferred to CPU before computing an event. Similarly, since events can modify the global data vectors, we need to send the data back to the GPU. As a consequence, every non-persistent event requires 2 data transfers. For persistent events, since they trigger the same operations at each timestep for their entire duration d, they require $2 * d$ transfers. In other words, they incur back-and-forth CPU-GPU data transfers at each timestep.

Since we know when every event will trigger before the execution, we also know when we will need to transfer the global data vector in advance. Therefore we only need to transfer data when we know an event will happen, avoiding useless data transfers.

B. Load Balancing

On hybrid architectures, we also have to address how to handle workload distribution between the different types of processing units. Since task granularity varies during the execution depending on the event schedule and on the ionic model, it is not possible to find a one-size-fits-all chunk size that performs good in an application-agnostic way.

To distribute the work to each processing unit, we rely on an adaptive partitioning algorithm [12]. The algorithm implemented in OPENCARP consists in the following steps:

- CPU cores are grouped together in a first pool; GPUs are grouped in a second pool.
- Initially, each pool is assigned a data partition corresponding to a fixed percentage of 50%. Each device of a pool is then assigned a uniform sub-partition of the pool.
- At each iteration, the ionic model kernel is applied once by each device on its own assigned piece of data, with the kernel version specifically compiled for that device (e.g. MLIR-CPU, MLIR-CUDA, ...).
  Such an application of the ionic model kernel by a single device on its own piece of data at a single iteration corresponds to a StarPU task. Recall that one iteration can be either one individual simulation timestep, or a consecutive series of merged timesteps.
- At each successive iteration, the partitioning ratio is adapted by taking into account the execution time of each individual task of the previous iteration considering the size of the piece of data it received, so as to minimize the imbalance in the next iteration.
  Note that all CPU cores get the same ratio, and all GPUs get the same ratio in our current implementation.

To make the CPUs sub-partitions more SIMD-friendly, we align all chunk sizes to the SIMD vector size, with the potential remainder assigned to the last CPU core.

Since re-partitioning can be costly because of the data transfers it incurs, we choose to re-partition only if the load imbalance (i.e. the relative difference in execution time of the two partitions) is greater than a given threshold t. Choosing a large t provides less accurate partitioning but reduces the number of re-partitioning and incurs less data-transfer overhead. The optimal threshold t varies from model to model. We empirically found 10% to be the best average threshold over all the models, therefore we set the default value of t to 10%, but the user has the option of selecting t as an optimization parameter.

C. Data Transfers

Whenever a synchronization triggered by an event happens, the global data computed by the ionic models on the CPUs and GPUs are gathered on the host and updated before the

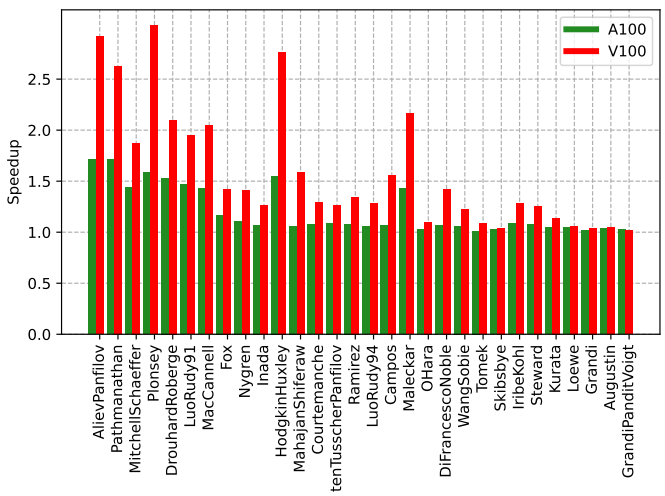

Fig. 4. Speedup of StarPU memory management over CUDA unified memory on Nvidia V100 and A100 GPU

next time step, and sent back to the devices. Due to this dependence, no prefetch can be used and data transfers cannot be overlapped with computations.

The StarPU runtime system [1] provides application programmers with a task-oriented parallel programming model targeting multicore nodes and accelerators. It proposes a Distributed Shared Memory (DSM) abstraction, which relies on data dependencies to transparently manage data transfers, replications and prefetching between the main memory and the memory spaces of the accelerator devices. The memory consistency between the multiple replicates of a piece of data is enforced through an MSI (Modified/Shared/Invalid) algorithm-based distributed shared-memory module within StarPU[2]. We thus have two options to handle data transfers, either use the automatic mechanisms provided by StarPU to handle dependencies, or handle such data transfers outside of StarPU. We experimented with the two approaches, one resorting to data copies managed by StarPU, and the other relying on the unified memory mechanism from CUDA.

Previous works comparing explicit data copies and CUDA unified memory [4], [9], [14] for Nvidia V100 and A100 GPUs show that explicit memory management usually outperforms implicit updates with unified memory, at the cost of a more difficult code to write. Unified memory is on par or outperforms explicit copies only if data can be prefetched and an overlap with other computations is possible.

For our application, Figure 4 shows the speedup obtained by using StarPU for handling data transfers compared to unified memory on a V100 and an A100 GPU. Ionic models on the X-axis are sorted by their execution time on the baseline *CPU* target. For the sake of the discussion, we further classify these models into 3 duration categories according to the makespan of an 8M elements mesh and 10k timesteps execution: *small* models for a makespan lower than 1 min (models from *AlievPanfilov* to *Nygren*), *medium* for a makespan between 1-5 min (*Inada* to *Tomek*) and *large* for a makespan above 5 min (*Skisbye* and above).

On V100, the speedup can reach up to $3\times$ on smaller models but is less noticeable on larger models. This can simply be explained by the fact that large models have much higher operational intensity, and for this reason the difference in memory transfer times are less noticeable. The difference in performance is much smaller on the A100 GPU and only benefits very small models. However, the speedup can reach up to $1.6\times$, and no slowdown is observed on any model. Since we observed that using StarPU provides better results on average and no slowdown, we rely on StarPU for data transfers in the remainder of the paper.

D. Resource Selection

One of the main issues we had to address while working on the optimization of this application was how to deal with the very different sets of computational intensities among ionic models and possible experiments. While using all resources for computation-heavy ionic models is generally a good idea, it is not always the case for models with low computational intensity. In such a case, it is often possible to obtain better performance using fewer resources. We thus implemented a greedy resource optimization heuristic to automatically and transparently adapt the number of resources used during the execution, which computes a preferred number of computing units to use concurrently to approach the best possible performance for the selected model.

We define a task wave as a set of tasks across the computing units processing either the same standalone timestep or the same contiguously aggregated timesteps. We rely on the StarPU on-line performance profiler to compute the execution time of each task wave to estimate the performance on a given configuration. We enforce a threshold of 300 timesteps to ensure measurement stability even on small models. Since the execution times are collected at the task wave boundaries, the actual number of timesteps measured can be larger, depending on the merged timesteps layout. In such a case, we normalize the measurements to 300 timesteps.

Our heuristic applies a simplified dichotomic search to elect a preferred number of resources to use, with one iteration of the search applied every few timesteps. The heuristic can choose any number of GPUs (from the available set of GPUs) but selects either all or no CPU at all. This is because the CPUs have less impact on the overall performance compared to the GPUs.

The function used for our heuristic can be divided into two steps : (i) find the number of GPUs and (ii) decide whether using CPUs is beneficial.

In order to compute the number of GPUs, the algorithm first measures the execution time of 300 timesteps on all GPUs and no CPU. Then, at each iteration, the execution time of another 300 timesteps is measured and compared against the

[2]While the StarPU runtime system also comes with hybrid task scheduling and performance modeling modules, these optional services are not employed here, to give precedence to the Dynamic Task Granularity management presented here.

current best solution. The number of GPUs is then updated according to the dichotomic search algorithm. Note that after the algorithm finds a smaller configuration better than the previous one, it will not test the configurations in between. For instance, if using 4 GPUs is better than 8 GPUs, but also better than 2 and 3 GPUs, it will assume that the best configuration is 4 GPUs and will not test for 5, 6 and 7 GPUs. The same way, if 2 is better than 4, then 3 will never be tested. We chose this method in order for the algorithm to converge fast to a good solution.

Once the number of GPUs is computed, the algorithm compares the impact of using CPUs along with this number of GPUs on the execution time. The configuration with the best performance is then set.

V. Experiments

The implementation is evaluated on two architectures:

- **Plafrim's sirocco 22-25** with 2x A100 GPUs, 2x 32-cores AMD Zen3 (AVX2) and 512 GB of RAM. This configuration will be called `Sirocco` in the following.
- **Grid5000's Gemini** with 8x V100 GPUs, 2x 20-cores Intel Xeon E5-2698v4 (AVX2) and 512 GB of RAM. This configuration will be called `Gemini` in the following.

Our work is implemented on top of the openCARP source from the git repository. Our experiments run on every standalone model (except models using the Rosenbrock function) using the `bench` executable with $819,200$ cells and $10,000$ timesteps (equivalent to 100 ms of cardiac activity)[3]. Events take place every 100 timesteps, this way the execution will contains 100 task waves of equal size (100 timesteps). The number of cells is close to what is expected for cardiac simulation experiments. However, we use $10,000$ consecutive timesteps to better evaluate the overhead of load balancing on short executions whereas real experiments usually use more timesteps.

We ran two kinds of experiments where the GPUs are always used but the CPUs are not. In what we call the multi-GPU experiments (Section V-A) the CPUs are only used for the events computation, while in the hybrid version (Section V-B) the CPUs can also be used for the kernel computations.

Previously published work [23] presented the performance in terms of Gflops/s of the MLIR-optimized ionic model kernels. The results showed that their performance reach up to 8% of the peak performance of the target machine because of their low compute intensity. For these reasons, we only focus on the possible parallelization improvements of the models compared to the current solution and show speedup plots.

A. Multi-GPU performance

The results of our benchmark for the multi-GPU version on `Gemini` are shown in Figure 5. We plot the speedup obtained using 1, 2, 4, and 8 GPUs compared to an execution using the **CPU** target with 40 CPU cores. Models are sorted by ascending execution time on the **CPU** target from left to right.

With 8 GPUs the multi-GPU version can reach up to 64.37 speedup, compared to 16.3 with 1 GPU. We notice that while GPU numbers scale well on large models, many of the smaller models lack sufficient computational intensity to take advantage of 8 GPUs. For most of the smaller models, the best number of GPUs is rather 4 among the configurations tested. We observe that the geometric mean of the speedups when considering all the models is 13.37 with 4 GPUs, while it is only 12.28 with 8 GPUs. When considering only the large models (starting from *IribeKohl*) the geometric mean of the speedups is instead 23.31 for 4 GPUs and 38.01 for 8 GPUs. Thus, using all available GPUs unconditionally may result in a worse average performance than using just a limited number of them.

Figure 5 also shows the performance of our resource selection heuristic (with hybrid disabled) labeled as *Auto*. We can see that for most kernels, the performance with Auto GPUs are close to the best observed performance. To further observe whether the results of our heuristic are accurate, we check the distance from the best solution, among the solutions tested, as shown in Table I. In this Table, for each model, we rank all the resource configurations for a given ionic model from worst to best, including configurations with $3, 5, 6$, or 7 GPUs. We then compute the distance between the rank of the configuration found by *Auto* and the rank of the best configuration among those tested (i.e. if *Auto* finds the 3rd best configuration, the distance is 2). Table I shows that *Auto* manages to find the best configuration for 12 of the 30 models and the second best one for 10 of them. Interestingly, the vast majority of cases where the heuristic only finds the second best solution is due to the dichotomic search skipping the best configuration: as explained in Section IV-D, if the best configuration is 5 GPUs, but 4 GPUs is better than 8, configurations with $5, 6$ or 7 GPUs will be skipped. This comes down to a compromise between accuracy and reducing the overhead from the research time. Since the performance between the second and first best solution tend to be close, the impact is limited. For 8 of the 30 ionic models, however, *Auto* results in a distance of 2 or worse. We identified two different patterns in those 8 models:

- In a few cases, when the difference in configuration performance is very small, the heuristic randomly picks one configuration based on small timing variations. This is the case on *Maleckar* for example. Thankfully in this scenario, the impact on performance is negligible.
- In most cases, this is due to the ionic models relying heavily on conditional instruction, making the use of a greedy heuristic not well adapted.

Since our heuristic needs $1,200$ out of $10,000$ timesteps (4 configurations to test for 300 timesteps each) to find the best number of GPUs, the results of *Auto* are always slightly slower than directly using the best configuration. Note that on longer experiments, the research time of our heuristic is still $1,200$ timesteps, making the performance gap less significant.

[3]bench parameters are: `-a 100 -n 819200 --numstim=0`

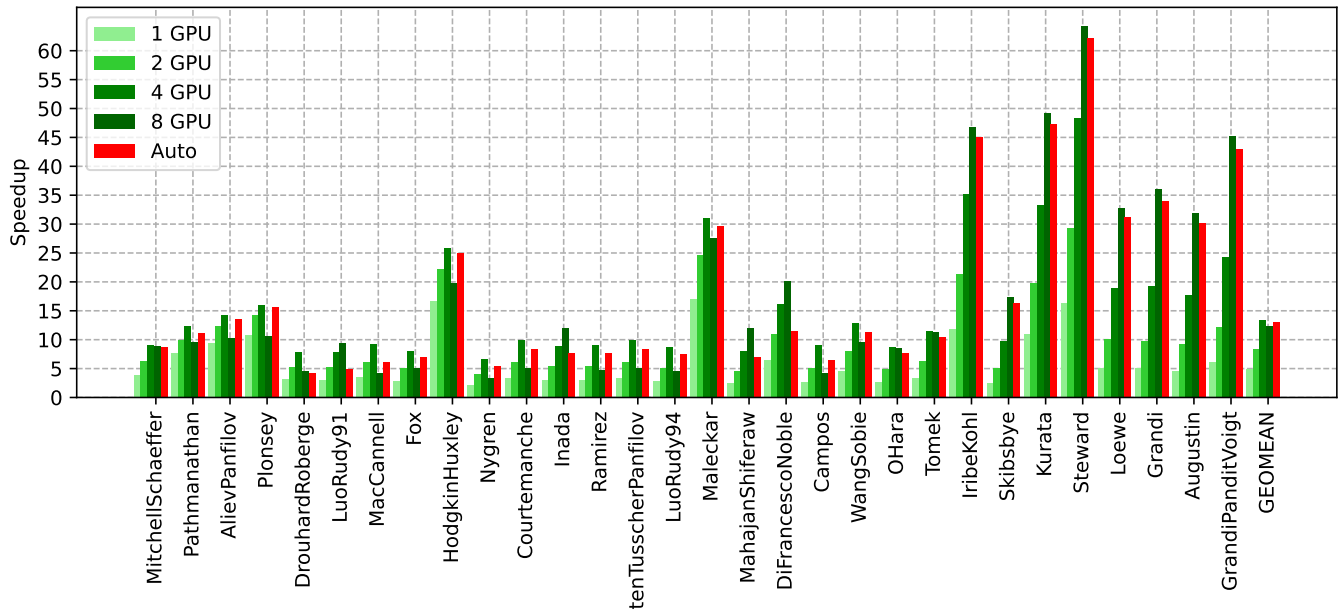

Fig. 5. Scalability on one Gemini node (eight V100) with different number of GPUs. The baseline is the **CPU** version (40 cores)

Overall, the geometric mean of speedups of *Auto* is 13.08 on all models, and 36.29 on large models. This indicates that the heuristic manages to provide similar performance to the best average on all models. As a conclusion, our heuristic seems to alleviate the need for the users to choose the number of GPUs on the multi-GPUs version while still maintaining good performance.

	best	2nd-best	3rd-best	4th-best	5th-best
Gemini	12	10	2	3	3
Sirocco	25	2	0	3	N/A

TABLE I
SUCCESS OF THE HEURISTIC TO MATCH THE BEST CONFIGURATION (OR THE X-BEST) ON THE DIFFERENT MODELS (EG. ON GEMINI THE HEURISTIC FOUND THE 2ND-BEST SOLUTION ON 10 MODELS).

B. Hybrid performance

We ran our benchmark for the hybrid version on Sirocco. The results are shown in Figure 6. Here we compare the speedup obtained using 1 and 2 GPUs with and without CPUs compared to an execution using the **CPU** target with 64 CPU cores. Hybrid performance are shown only if they lead to a speedup, otherwise only the GPU performance are shown. Models are sorted by their makespan on the **CPU** target in ascending order from left to right.

Figure 6 shows that the hybrid version can reach up to 33.45 speedup with 2 GPUs and 64 CPU cores, against 31.64 with only 2 GPUs. The geometric mean of speedups is 9.84 for the hybrid version against 9.69 for 2 GPUs. When using only one GPU, most models benefit from using the hybrid versions. However, when using 2 GPUs, some of the smaller models perform better without CPU cores. These models tend to be the models that also obtain better performance with 1 GPU compared to 2 GPUs suggesting that those models lack computational intensity in the first place. This is the case for models such as *HodgkinHuxley* or *Plonsey*.

However some smaller models such as *LuoRudy91* seem to benefit from an increase from 1 to 2 GPUs but a loss in performance when we add CPUs. To further investigate this behavior, we measured the execution time of each task wave of *LuoRudy91* on CPU and GPU separately as shown in Figure 7. During load balancing, we use the default threshold of 10% to check if we need to re-partition. We can see on Figure 7 that a single task wave execution for this model is very short (around $18ms$), making small deviations in execution time significant enough to trigger re-partitioning, resulting in multiple additional CPU-GPU data transfers thus lowering performance. This indicates that the 10 % threshold is too small in this case. Another situation where hybrid computation brings poor results is with models where the GPUs compute the kernel much faster than the CPUs, like the *Maleckar* model.

Larger models, however, tend to perform better. Looking at the results for the *Skisbye* models on Figure 8, models that perform well tend to have timesteps large enough for load balancing not to be easily swayed by minor variations in execution time. In the example of *Skisbye*, timesteps are around 10 times bigger than in the *LuoRudy91* model, resulting in a rapid stabilization of the load balance. In short, the load balancing algorithm, while able to find a good load balance, needs sufficient workload to be efficient.

In Figure 6, we also evaluate our resource selection heuristic, labeled as *Auto*, this time enabling the option to take CPUs into account. Again, we rank each configuration and evaluate

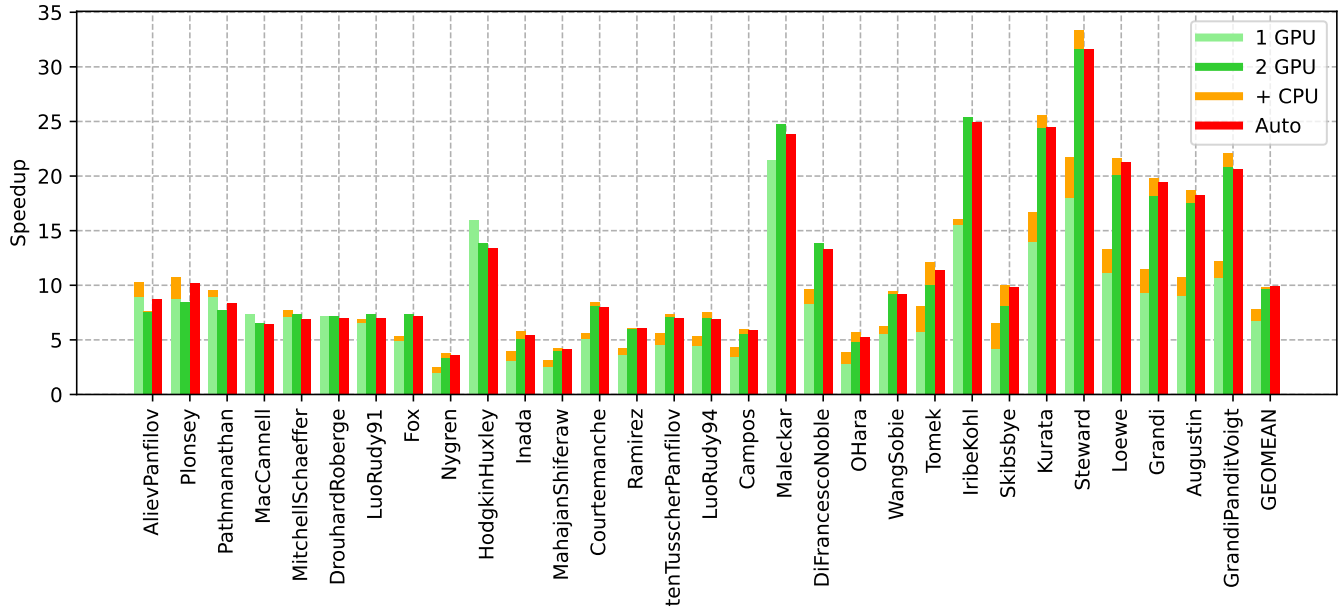

Fig. 6. Speedup of hybrid versions on one Sirocco node (two A100). The baseline is the **CPU** version (64 cores). The speedup reached by the addition of the CPU to 1 or 2 GPUs is shown by the orange bar

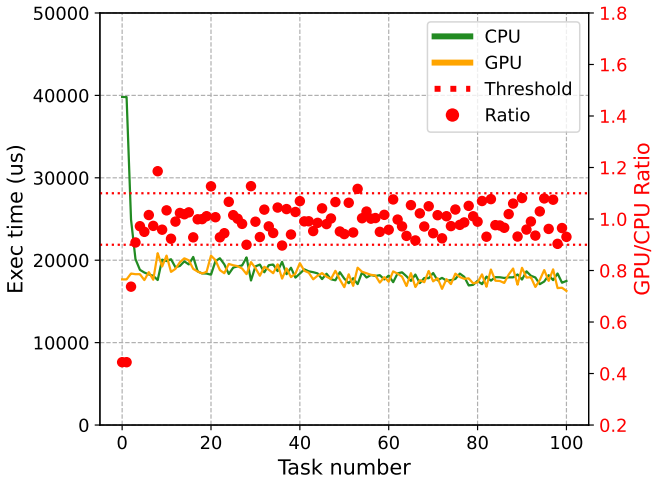

Fig. 7. Execution time of the CPUs and GPUs for each task wave on the *LuoRudy91* model. GPU/CPU ratio at each timestep is represented with red dots, the red lines represent the load-balancing threshold

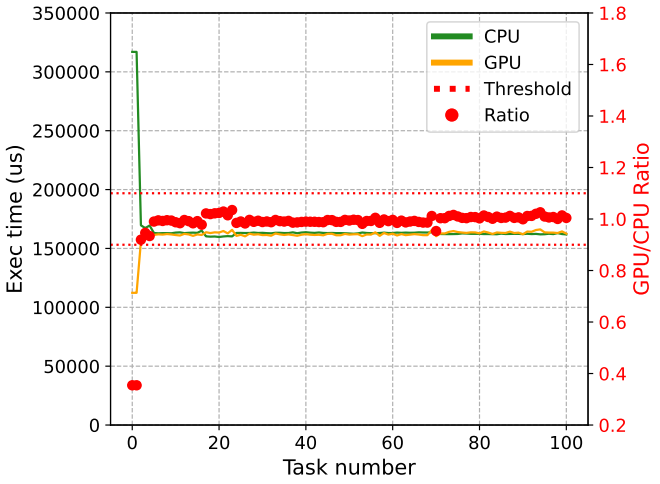

Fig. 8. Execution time of the CPUs and GPUs for each task group on the *Skibsbye* model. GPU/CPU ratio at each timestep is represented with red dots, the red lines represent the load-balancing threshold

the distance between the rank of the configuration found by *Auto* and the best configuration, the resulting distances are shown in Table I. On 25 out of 30 cases, *Auto* manages to find the best configuration. However, *Auto* finds the 2^{nd} best solution for 2 models. This is because *Auto* finds the best number of GPUs but not of CPUs. For instance, `LuoRudy94` kernel starts at a high computational intensity. The heuristic decides to use the CPU, but then, the arithmetic intensity drops after the decision is made (and does not increase afterward). Using a CPU is no longer needed, but the heuristic already decided otherwise. Regarding the last models shown in Table I, the heuristic selects the worst configuration on 3 of the 30 models. For all those models this is due to the arithmetic intensity varying during execution, conflicting with the greedy nature of our heuristic leading to bad decisions. Despite this issue, the geometric mean of speedup on all models with the *Auto* setting is 9.97, which on average, is better than simply using all resources every time.

VI. Related Works

The literature contains a numbers of efforts to port cardiac electrophysiology simulation to parallel architectures. Myokit [5] is a tool designed for modeling and simulating cardiac cellular electrophysiology. It supports model loading from its native format and other formats such as CellML [6], and offers export to CUDA, and OpenCL. Myokit does not provide support for multi-device architectures but could benefit from some of the implementations presented in our work.

Campos et al. [3] propose a GPU implementation of the Lattice Boltzmann method for cardiac electrophysiology simulation. G-Heart. Zhang et al. [27] also propose a GPU-based system for electrophysiological simulation. Both studies include ionic model computation but only focus on single GPU implementation. Langguth et al. [15] implement a simulator for 3D tissue of the human cardiac ventricle on the supercalculator Tianhe-2. This implementation, however, is highly optimized for Tianhe-2 usage of Intel Knight Corner many-cores, which follow a markedly different paradigm than accelerators such as Nvidia and AMD GPUs.

In term of dynamic load-balancing for hybrid architectures, an extensive amount of work has already been done on the subject. One common approach is to use task scheduling to distribute works the different devices. Youness et al. [26] propose a task-based load-balancing algorithm for the non-blocked Householder transformation. HyGraph [11] is a job-based dynamic load-balancing algorithm for CPU-GPU Hybrid architectures for graph processing. Other works focus on distributing data over a set of resources according to their processing powers. UIIMF [2] is a load-balancing framework that provides multi-objective functions to target both performance and energy consumption. Zhong et al. [28] offer a functional performance model approach to data partitioning for hybrid CPU/GPU plateforms. These studies are complementary to our work because they focus on computing multiple kernels on different devices instead of using all available devices to compute a single data-parallel kernel.

Regarding task granularity management, some works aggregate multiple fine-grained task into one coarse-grained task [21]. Other studies take the opposite approach of starting from coarse-grained and splitting them into finer-grained task when appropriate [8], [25]. Granularity management, starting from a DSL and generating code using StarPU runtime have been studied in the particular context of linear algebra [7]. Splitting one task into as many tasks as devices has been studied [13] as a compiler optimization and analysis, adapting dynamically the subdomain given to each task in order to tackle heterogeneity. Our approach is similar to this work, starting from EasyML DSL and extending it with dynamic GPU/CPU allocation.

EngineCL [18] [10] is an OpenCL framework that transparently handles co-execution for heterogeneous architectures. It allows both static and dynamic load-balancing, using the HGuided scheduler, for performance or energy consumption and supports CPU+GPU+FPGA Hybrid architectures. SKMD [17] is a framework that allows dynamic load balancing for CPU-GPU co-execution of a single kernel by assigning subsets of data-parallel workload over multiple CPUs and GPUs. OmpSS [15] also comes with a scheduler to allow transparently using all devices of a heterogeneous architecture to compute a single kernel, it also comes with an auto-tuned version of the HGuided scheduler. All these approaches only consider using all available devices for computation, while our implementation tries to use fewer resources when it would lead to better performance.

VII. Conclusion

This paper presents a method to efficiently and transparently run EasyML ionic models for heterogeneous architectures on the OpenCARP framework. To perform well even on models lacking computational intensity, we aggregate multiple kernel calls into a single task and employ a dynamic load-balancing strategy to distribute the workload. In addition, we propose a resource selection heuristic to alleviate the need to select a good number of computing resources for a given ionic model.

We evaluate our implementations on two architectures. Our experimental results demonstrated that our method, when using only GPUs, finds the best or second-best resource configuration in 80% of the cases on 8 GPUs and reaches a speedup of up to 62 compared to the CPU implementation. The experiments on hybrid executions show that our method finds the best or second-best resource configuration on 90% of the models and reaches a speedup of up to 32.

As a future improvement, we would like to consider energy consumption as an optimization target for our load-balancing algorithm and resource optimization heuristic. Another area of improvement we would like to work on is to be able to transparently find the best load balancing threshold for each ionic model.

Acknowledgment

Experiments presented in this paper were carried out using the **PlaFRIM** experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr).

Experiments presented in this paper were carried out using the **Grid'5000** testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr)

This work was supported by the European High-Performance Computing Joint Undertaking EuroHPC under grant agreement No 955495 (**MICROCARD**) co-funded by the Horizon 2020 programme of the European Union (EU), the French National Research Agency ANR, the German Federal Ministry of Education and Research, the Italian ministry of economic development, the Swiss State Secretariat for Education, Research and Innovation, the Austrian Research Promotion Agency FFG, and the Research Council of Norway.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

[2] A. Cabrera, A. Acosta, F. Almeida, and V. Blanco. A dynamic multi–objective approach for dynamic load balancing in heterogeneous systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2421–2434, 2020.

[3] J. Campos, R. Oliveira, R. dos Santos, and B. Rocha. Lattice boltzmann method for parallel simulations of cardiac electrophysiology using gpus. *Journal of Computational and Applied Mathematics*, 295:70–82, 2016. VIII Pan-American Workshop in Applied and Computational Mathematics.

[4] S. Chien, I. Peng, and S. Markidis. Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57, 2019.

[5] M. Clerx, P. Collins, E. de Lange, and P. G. Volders. Myokit: A simple interface to cardiac cellular electrophysiology. *Progress in Biophysics and Molecular Biology*, 120(1):100–114, 2016. Recent Developments in Biophysics and Molecular Biology of Heart Rhythm.

[6] M. Clerx, M. T. Cooling, J. Cooper, A. Garny, K. Moyle, D. P. Nickerson, P. M. F. Nielsen, and H. Sorby. Cellml 2.0. *Journal of Integrative Bioinformatics*, 17(2-3):20200021, 2020.

[7] A. Duchâteau, D. A. Padua, and D. Barthou. Hydra: Automatic algorithm exploration from linear algebra equations. In *Code Generation and Optimization*, pages pp.1–10, Shenzhen, China, Feb. 2013.

[8] M. Faverge, N. Furmento, A. Guermouche, G. Lucas, R. Namyst, S. Thibault, and P.-a. Wacrenier. Programming Heterogeneous Architectures Using Hierarchical Tasks. *Concurrency and Computation: Practice and Experience*, 35(25), 2023.

[9] M. González and E. Morancho. Multi-gpu systems and unified virtual memory for scientific applications: The case of the nas multi-zone parallel benchmarks. *Journal of Parallel and Distributed Computing*, 158:138–150, 2021.

[10] M. A. D. Guzmán, R. Nozal, R. G. Tejero, M. Villarroya-Gaudó, D. S. Gracia, and J. L. Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 75:1732 – 1746, 2019.

[11] S. Heldens, A. L. Varbanescu, and A. Iosup. Dynamic load balancing for high-performance graph processing on hybrid cpu-gpu platforms. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pages 62–65, 2016.

[12] P. Huchant, D. Barthou, and M.-C. Counilh. Adaptive Partitioning for Iterated Sequences of Irregular OpenCL Kernels. In *SBAC-PAD - 30th International Symposium on Computer Architecture and High Performance Computing*, Lyon, France, Sept. 2018.

[13] P. Huchant, M.-C. Counilh, and D. Barthou. Automatic OpenCL Task Adaptation for Heterogeneous Architectures. In *Euro-Par*, Euro-Par 2016: Parallel Processing, pages 684 – 696, Grenoble, France, Aug. 2016.

[14] M. Knap and P. Czarnul. Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus. *The Journal of Supercomputing*, 75, 11 2019.

[15] J. Langguth, Q. Lan, N. Gaur, X. Cai, M. Wen, and C.-Y. Zhang. Enabling tissue-scale cardiac simulations using heterogeneous computing on tianhe-2. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 843–852, 2016.

[16] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[17] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 245–255, 2013.

[18] R. Nozal, J. L. Bosque, and R. Beivide. Towards co-execution on commodity heterogeneous systems: Optimizations for time-constrained scenarios. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 628–635, 2019.

[19] G. Plank, A. Loewe, A. Neic, C. Augustin, Y.-L. Huang, M. A. Gsell, E. Karabelas, M. Nothstein, A. J. Prassl, J. Sánchez, G. Seemann, and E. J. Vigmond. The openCARP simulation environment for cardiac electrophysiology. *Computer Methods and Programs in Biomedicine*, 208:106223, 2021.

[20] M. Potse, E. Saillard, D. Barthou, and Y. Coudière. Feasibility of whole-heart electrophysiological models with near-cellular resolution. In *2020 Computing in Cardiology*, pages 1–4, 2020.

[21] C. Rossignon, H. Pascal, O. Aumage, and S. Thibault. A NUMA-aware fine grain parallelization framework for multi-core architecture. In *PDSEC - 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing - 2013*, Boston, United States, May 2013.

[22] A. Thangamani, T. T. Jost, V. Loechner, S. Genaud, and B. Bramas. Lifting code generation of cardiac physiology simulation to novel compiler technology. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, page 68–80, New York, NY, USA, 2023. Association for Computing Machinery.

[23] T. Trevisan Jost, A. Thangamani, R. Colin, V. Loechner, S. Genaud, and B. Bramas. Gpu code generation of cardiac electrophysiology simulation with mlir. In *Euro-Par 2023: Parallel Processing: 29th International Conference on Parallel and Distributed Computing, Limassol, Cyprus, August 28 – September 1, 2023, Proceedings*, page 549–563, Berlin, Heidelberg, 2023. Springer-Verlag.

[24] E. Vigmond. EasyML. https://opencarp.org/documentation/examples/01_ep_single_cell/05_easyml, 2021.

[25] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical DAG Scheduling for Hybrid Distributed Systems. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015)*, Hyderabad, India, May 2015.

[26] H. Youness, M. Osama, and a. Tarek. Load balancing on cpu-gpu heterogeneous system. 12 2012.

[27] L. Zhang, K. Wang, W. Zuo, and C. Gai. G-heart: A gpu-based system for electrophysiological simulation and multi-modality cardiac visualization. *Journal of Computers*, 9(2):360–367, 2014.

[28] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications. In *2012 IEEE International Conference on Cluster Computing*, pages 191–199, 2012.