Load balancing and precision analysis for cardiac simulation M2 Internship report

Vincent Alba

To cite this version:

Vincent Alba. Load balancing and precision analysis for cardiac simulation M2 Internship report. Computer Science [cs]. 2022. hal-03784546

HAL Id: hal-03784546

https://inria.hal.science/hal-03784546v1

Submitted on 28 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Load balancing and precision analysis for cardiac simulation

M2 Internship report

Vincent Alba

Internship supervisors: M. Denis Barthou, Mrs. Marie-Christine Counilh

Academic year: 2021 – 2022

Contents

1 Presentation of the team and establishment **3**
1.1 Inria . . . 3
1.2 Team STORM . . . 3
1.3 Microcard project . . . 3

2 Internship topic & problematic **4**
2.1 OpenCARP . . . 4
2.1.1 Overview . . . 4
2.1.2 Limpet . . . 5
2.1.3 Gestion of ionic models in Limpet . . . 6
2.2 Problematics . . . 7
2.2.1 Running ionic models computation on heterogeneous architectures . . . 7
2.2.2 Precision analysis . . . 8

3 Conducted Work **8**
3.1 Methods and tools used . . . 8
3.1.1 Code generation with MLIR . . . 8
3.1.2 Handling heterogeneity . . . 9
3.1.2.1 Different method of handling heterogeneity . . . 9
3.1.2.2 StarPU . . . 11
3.1.3 Handling precision Analysis . . . 12
3.1.3.1 Introduction to floating point errors . . . 12
3.1.3.2 Planned experiments . . . 13
3.1.3.3 FPTaylor . . . 14
3.1.3.4 Verificarlo . . . 14
3.1.3.5 Monte Carlo Arithmetic (MCA) in Verificarlo . . . 15
3.2 Planning . . . 16
3.3 Integration of the new tools into the OpenCARP framework . . . 18
3.3.1 Adding StarPU . . . 19
3.3.2 Adding Verificarlo . . . 19
3.3.3 Final version . . . 19
3.4 Implementation of Limpet's heterogeneous version . . . 20
3.4.1 Multi-Tasks CPU version . . . 21
3.4.2 CPU Version experiments . . . 23
3.5 Precision analysis experiments . . . 25
3.5.1 Precision modification on an entire model . . . 25
3.5.2 Precision modification on part of a model . . . 30
3.5.3 Precision modification depending on time steps . . . 32
3.5.4 Perspectives for optimizations . . . 36

4 Remaining works **36**

5 Conclusion **38**

Acknowledgement

This report wouldn't have been possible without the help and support of several individuals who, in one way or another, helped me with their guidance to complete my internship.

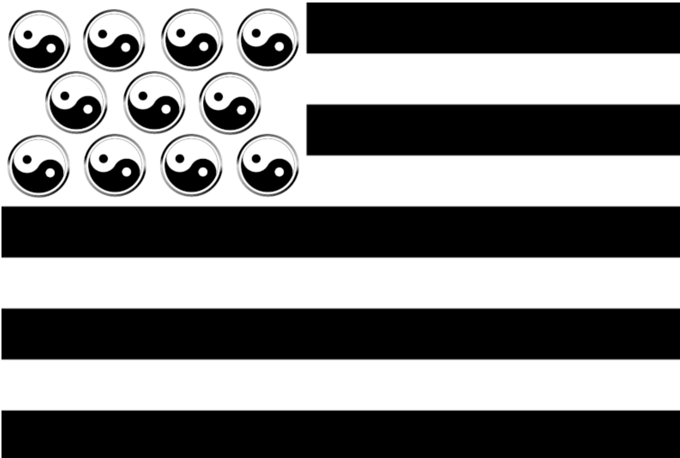
I would like to express my deepest gratitude to my supervisors, who helped me greatly during this internship, each in their own way. *Denis*, thanks for taking the time out to hear and guide me in spite of being extraordinarily busy. Thank you, *Marie-Christine* for the countless discussions when I needed an outside view on my work, it helped me a lot when I was stuck. And I would like to thank both of you for allowing me to pursue this internship.

Special thanks to *Amina* for her numerous help and her precious guidance throughout the preparation of my internship defense. I would also like to thank my other colleagues of the STORM team for their help and advice, but also for all the delicious cakes and chocolates throughout those six months. I think of *Emmanuelle*, *Laércio*, *Nathalie*, *Mihail*, *Olivier*, *Pierre-André*, *Raymond*, *Sabrina*, *Samuel* and *Scott*.

I would also like to offer my thanks to *Pablo* from UVSQ for his help with Verificalo and for taking his time to work with me, and of course to *Tiago* and *Vincent* from the University of Strasbourg for welcoming me and helping me work with the MLIR versions of Limpet. I could not have made this much progress without their help.

Of course, thanks to all my colleagues at the OpenSpace for the fun times and the (too) many breaks. *Maxime* for his valuable lesson on the size of many historical monuments of Bordeaux, I will never forget the size of the stone bridge ever again. *Chiheb* for his charismatic presence, *Celia* for her kindness (and chocolates), *Thomas* for fixing the coat hanger, *Charles* and *Charles* for being Charles, *Pierre-Antoine*, *Diane* and *Baptiste* for the long coffee breaks, *Gwenolé* for always being ready to help, *Van Man* for the sauces, *Alice* for keeping Raymond busy, but also *Edgard*, *Radja*, *Pélagie*, *Kun*, *Bastien* and *Romain*.

And finally thanks to *Panagiotis*, my friend, for helping me with the redaction of this report.



Abstract

This document is a report written as part of an end-of-study internship. This internship took place in the Inria STORM team. The goal of this internship is to adapt a code generator for cardiac simulation to modern heterogeneous architectures. To achieve this we switched the code structure to use task-based parallelism using StarPU to enable better handling of heterogeneity. This led us to lay the ground for the future heterogeneous version of the cardiac simulation. We also performed precision analysis with Verificarlo on the generated code to analyze the possibilities of using mixed precision for better performance and energy consumption. We explored the effects of switching floating-point precision on the accuracy of the simulation using Verificarlo, either by changing the precision on the whole simulation, only some function, or only during some iteration. We found out that some cardiac ionic models may be converted to lower precisions without impacting accuracy too much, but also that we could change precision only on the less active part of the computation to reduce the impact on accuracy. We also proposed a way to choose the resolution of lookup tables using precision analysis.

1 Presentation of the team and establishment

1.1 Inria

The internship took place at the Inria Bordeaux - Sud-Ouest research center in Talence within the STORM team from 02 February to 29 July 2022. Inria (National Institute for Research in Digital Science and Technology) is composed of 3900 researchers and engineers divided into 200 project-teams and 9 research centers all located in the heart of various French university campuses. It is notably involved in numerous projects such as the development of the OCaml language or the Coq proof assistant, as well as the development of various free software.

1.2 Team STORM

STORM (Static Optimizations and Runtime Methods) is a joint project-team between Inria, CNRS, the University of Bordeaux, and Bordeaux INP. The team, located at the Inria Bordeaux - Sud-Ouest research center, is made up of 30 members who are also part of the LaBRI (Laboratoire Bordelais de Recherche en Informatique).

The main themes of the team are related to the field of high-performance computing and all the team members share expertise in parallel languages, code optimization, and parallel task scheduling.

Those expertises have led the team to develop various tools, including StarPU, a runtime system for parallel architectures used in this internship which will be described later in section 3.1.2.2.

1.3 Microcard project

Microcard[16] is a European research project funded by EuroHPC. The project goal is to build software able to simulate cardiac electrophysiology on a whole heart using a sub-cellular resolution. The program should be able to run on future exascale supercomputers.

The project involves multiple partners from France, Italy, Germany, Austria, Norway, and Switzerland. The project is split into ten different work packages.

This internship is related to work package 2: Task-based parallelization. The objective of this work package is to provide a parallel and task-based version of the application.

Part of this internship also applies to work package 6: Code generation for heterogeneous architectures. The objective of this work package is to develop tools to automatically generate optimized code (such as vectorized code) adapted to heterogeneous architectures.

2 Internship topic & problematic

To understand cardiovascular diseases, computer models are used to represent and understand the behavior of the heart and these diseases. The goal of the Microcard project is to create software able to run such simulation on a whole heart cell by cell. But currently, we are only able to run whole heart simulation by representing a large group of cells as one entity. If we wanted to use a cellular resolution we would be confronted with a problem that is, at last, a 10000× larger and also harder to solve. In other words, we can only hope to reach the computing power we need to run such a simulation on the future exaflopic super-computers.

However, simply porting the current version of cardiac simulation to such computers is not enough. This is because the current trend of architectures for supercomputers is to use highly heterogeneous architectures using a lot of accelerators. Writing code that runs efficiently on such architectures comes with a lot of challenges that are often very different from the ones we can encounter on standard architectures. The goal of this internship is to study and solve some of the challenges we will describe in this section.

2.1 OpenCARP

2.1.1 Overview

OpenCARP[15] is an open cardiac electrophysiology simulator for in-silico experiments. It is composed of multiple tools to allow for: simulation, visualization, and developing scripts to automate experiments to allow for reproducibility. Since it's both open source and focuses on state-of-the-art simulations, it is the main focus of the Microcard Project, and therefore the software we will focus on during this internship to deal with heterogeneity issues.

It should be mentioned that openCARP is composed of multiple tools that create a large ecosystem as we can see in figure 1. However, our work doesn't concern the entirety of this framework, as it relates directly to the openCARP simulator which is the heart of the openCARP ecosystem. This solver is responsible for the computation of *Ionic models* (See section 2.1.3) which are mathematical models that represent the electrical events that give rise to cardiac action potential at a cellular level. This solver is roughly composed of two parts:

- The Limpet ionic model library which handles the compilation and computation of those ionic models
- A solver, used to compute the cells membrane potential from the transmembrane current (See section 2.1.3).

This internship focuses on the code of Limpet, while the solver is handled by another work package of the Microcard project.

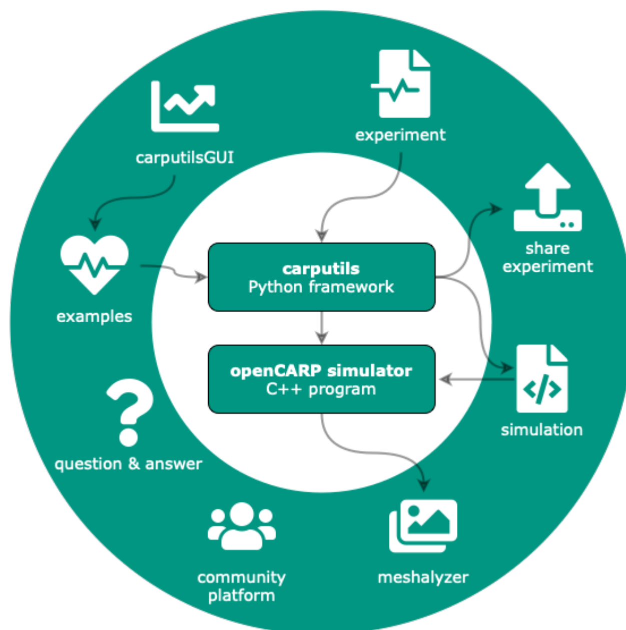


Figure 1: Plan of the OpenCARP ecosystem

2.1.2 Limpet

Limpet stands for Library of Ionic Model & Plug-ins for Electrophysiological Theorization, it's the library that manages everything related to ionic models in OpenCARP. Limpet is used both for the compilation of the ionic models and for manipulating them during the simulation.

The language used by Limpet to represent ionic models is EasyML, which is a simple modeling language able to describe markup equations. It is unable to use loops, functions, or any form of flow control, but uses markup to give different instructions to the compiler. Those instructions are either used to give different properties to some variables for the simulator, or for optimization purposes (for example flagging some variables as needing to be precomputed).

For compilation, Limpet takes into input the EasyML models and converts them into a library to be used by the OpenCARP simulator. It uses a parser written in python to generate C/C++ code for each model.

This enables the user to easily add models to OpenCARP by either describing them in EasyML or converting another modeling language to EasyML (some language conversions like CellML[5] to EasyML are supported by limpet). However, this means that any modification to the computation of ionic models needed for this internship has to be done through modification of the code generator.

For simulations Limpet can be used for two kinds of experiments :

- On single cell or tissue (multi cells) simulation with one model: This is done using the **bench** tool which is contained inside limpet. Such simulations are run on one cell for a single-cell simulation or multiple cells for tissue simulation.
- On tissue simulation with multiple models: Tissue simulation can also be run with multiple models, for this, it's necessary to define a mesh (which is handled by another part of OpenCARP) with multiple regions. Each region can have different models, or have the same model, but with different initial values. The communication between regions is handled by the solver.

To better understand what we changed during this internship let's start with a quick summary of how the OpenCARP simulator worked before our contributions. In figure 2 we can see the workflow of an OpenCARP simulation. Everything inside the "Limpet" frame on the figure is done only once during compilation. First, the ionic model compiler takes all the EasyML models the user wants to add to the simulator. The list of models that the compiler will take into account is specified in a file by the user. The compiler generates C/C++ codes for each of the ionic models, which are then compiled by a C/C++ compiler. Once all the models are compiled, they are all added to an ionic model library along with all the precompiled codes (all the other libraries used or provided by Limpet) needed for their computation. This library is then linked to the OpenCARP simulator for it to access the ionic model during simulation.

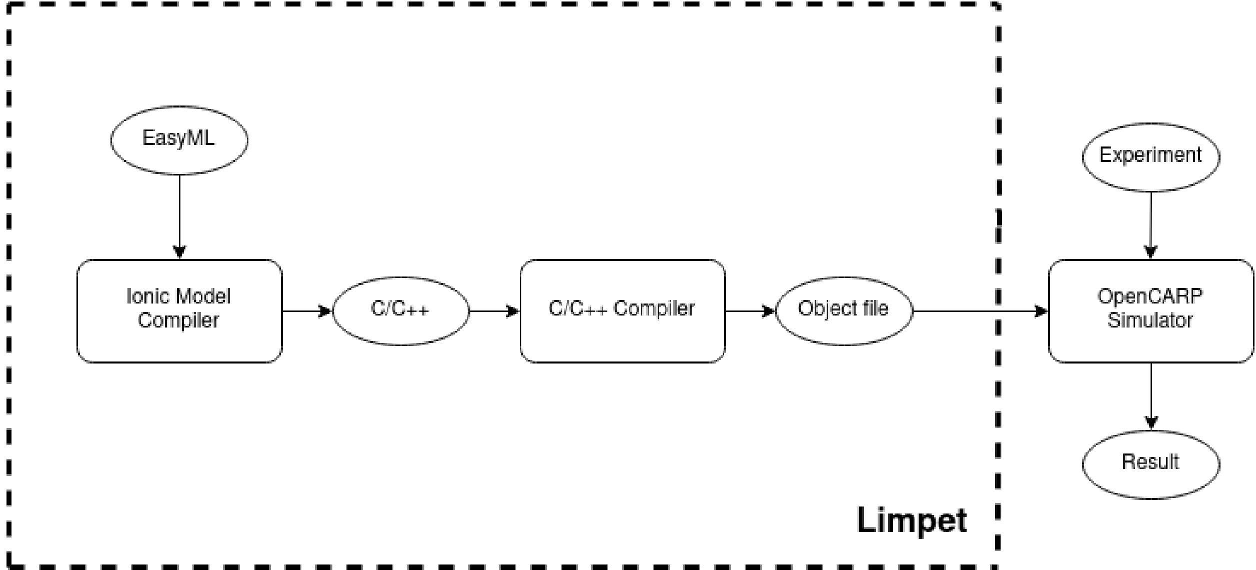


Figure 2: Workflow of Limpet in OpenCARP before our modification

2.1.3 Gestion of ionic models in Limpet

In this section, we will give a quick introduction of Ionic models (definition from [7]) and how they work in Limpet. This is not intended to be an explanation of the physics computed by those models, but instead an explanation of how the computation of ionic models works from a programmer's viewpoint.

Ionic models are the electrical representation of a cell based on the movement of ions across the membrane. Those movements result in a change of voltage across the membrane, this is what we call the transmembrane current. These currents, in turn, causes changes to the membrane potential (V_m). In OpenCARP ionic models represent the computation of the transmembrane currents for one timestep. OpenCARP's ionic models communicate with the rest of the simulation through variables called global variables, which correspond to the membrane potential V_m and the sum of the transmembrane currents I_{ion}. In short, each timestep the ionic models compute I_{ion} for each cell, the simulation recovers I_{ion} and computes the values V_m for each cell using a solver to solve the differential equation:

$$\frac{\Delta V_m}{\Delta t} = \frac{I_{ion}}{C_m}$$

Where t is the time, and C_m is the membrane capacitance. The new value of V_m is then passed to the ionic model which affects the transmembrane currents.

From the user point of view, to run the simulation, the following parameters need to be provided:

- Number of nodes: correspond to the number of cells computed by the ionic model.
- Duration: The duration of the simulation in timesteps (dt).
- Step duration: The duration of a timestep in microseconds

By default, the ionic model simulation produces one stimulus at the start of the simulation. These stimuli are a representation of electrical cardiac stimulation. They are parametrable by the user, it's possible to change different parameters such as the number of stimuli, when they start, their duration, or how strong they are.

From a program point of view, the computation of a model consists of three nested loops. The first loop iterate on the timesteps. This is an iterative loop where the computation of each step depends on the

previous ones. Therefore this loop is entirely sequential and cannot be parallelized. The second loop iterates over the ionic models and is only used for multi-region simulation, and is also sequential. The third loop iterates over the nodes of the models, this is where I_{ion} is computed for each node. In the base version of OpenCARP, this loop is already parallelized for CPU using OpenMP[12].

In terms of memory consumption, each node needs to store one value for each global variable (V_m and I_{ion}) and one value for each state variable (which are variables of the simulation that are specified in the model to be stored on a node by node basis). So we need to store $(sv + g) \times n \times size$ bits of data, where sv the number of state variables, g the number of global variables, n the number of nodes, and $size$ the size in bytes of a variable. Since the computation is in double precision, $size = 8$.

Finally, since we use them in some experiments, we will define here how lookup tables (or LUT) are defined and used in ionic models. It's possible for each variable, to specify whether or not it uses a lookup table. Those lookup tables exist to pre-compute for a given variable var its output for an array of input values. The pre-computed input values are specified by the user by giving an upper (ub) and lower bound (lb), and a resolution (r). The lookup tables then pre-compute the output of var for each input between lb and ub with a step of r. The tables are then used to obtain the output during the simulation. If we pass an input not pre-computed for a given var during the simulation, the output is linearly interpolated from the two nearest values. Lookup tables enable for better performance by avoiding redoing the same computation multiple times. However, if the value of r is too high, interpolation will provide an inaccurate result. In a way, Lookup tables are a compromise between performance and accuracy.

2.2 Problematics

2.2.1 Running ionic models computation on heterogeneous architectures

Currently, the code generated by Limpet is designed for homogeneous architectures. But if we want ionic model computation to run on supercomputers we need to adapt this code generation for heterogeneous architectures. First, Limpet needs to generate not only a CPU version of ionic model computation but also a GPU version.

Once a GPU version is available the next goal for handling heterogeneity is to take into account the different performance of each device. By this, we mean that because each different device may have different performances, naively distributing workload equally between devices will lead to suboptimal performance. This is because the slowest device will still have works to do while the other device will be idle. We can see an example of this phenomenon in Figure 3, where we can see a problem over N nodes on architectures with CPUs and 2 different GPUs. The problem is equally split into 3 tasks of $N/3$ nodes. However, because of its low performance compared to GPUs, the multi-core CPU task takes a much larger amount of time to complete than the 2 GPUs. This also means that the two GPUs have wasted computation time by idly waiting while there was still computation to be done. For this reason, it is important to achieve load balancing so that each device works for the same amount of time.

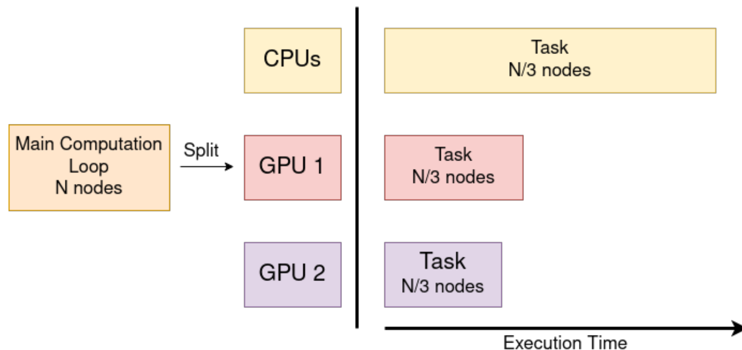


Figure 3: Naive load distribution on CPU and 2 different GPU for N nodes

However, how fast each device will perform a task doesn't entirely depend on clock speed. CPUs and GPUs for example handle computation and parallelism very differently. This means that while GPUs tend to be faster there are some types of tasks that run faster on CPU than on GPU. Therefore the work distribution may be different for each kind of task.

We also have heterogeneity to handle in Limpet's multi-region simulation, since different regions can hold different models. Since each model can have a very different computation cost, this means that we have to take into account those costs when doing load balancing on multi-region simulations.

The first goal of this internship will be to change Limpet's code to be able to handle heterogeneity. The generation of the GPU version isn't handled by us, but by the Inria Camus team from University of Strasbourg. The contribution of this internship is to work on the generation of the task-based parallel code able to automatically handle the heterogeneity of the hybrid version(CPU + GPU)

2.2.2 Precision analysis

Making cardiac simulation more efficient at running on heterogeneous architectures is the primary goal of the Microcard project. But with the current issue of climate change and global warming, it is also important to propose solutions that are also efficient at energy consumption. Future exascale computers are expected to consume several tens of MegaWatt, therefore we also need to aim at reducing the energy consumption of our codes.

One of the solutions for raising performance while reducing energy consumption is to employ mixed precision in our computations. Mixed precision is when we run part of our code in double floating-point precision and switch some parts to single or half floating-point precision. This usually leads to better performance but also better memory and energy consumption.

However mixed precision comes with a compromise. Every part of our code we switch to a lower precision will affect the accuracy of our result. And if we reduce precision too aggressively, we won't obtain results accurate enough to be trustworthy. Therefore we have a compromise between speed and accuracy to take into account. To correctly handle this compromise we have to run precision analysis on the code generated by Limpet to measure the effect of lowering precision on the accuracy of our results.

The second objective of this internship is not to implement a mixed precision version of the code but to run various precision analysis experiments to find how we can implement mixed precision correctly.

3 Conducted Work

3.1 Methods and tools used

As explained in section 2, this internship focuses on two different problems:

- Adapting Limpet to generate code able to handle the heterogeneous architectures of the future exascale supercomputer.
- Running code analysis on Limpet's generated code to find the correct way to implement mixed precision on those same codes.

Therefore in this section, we will present the methods and tools we choose for answering those problems.

3.1.1 Code generation with MLIR

As we said before, the INRIA Camus from the university of Strasbourg also working on the Microcard project will address the lack of a GPU version in Limpet during this internship. To produce a GPU version of the code of each model they used a tool named MLIR[10], which is a sub-tool of the LLVM[1] compilation infrastructure. MLIR is a compiler infrastructure that is intended to handle software that depends on multiple infrastructures with very different requirements. MLIR work by layering multiple Intermediate

representations to represent each of those requirements. This allows targeting one of those Intermediate Representation with code transformation and optimization that are adapted to them.

The Camus team uses MLIR to generate a GPU version directly from the code of each ionic model, using the same method they also generate a vectorized version of the CPU code of Limpet (See [11]). This optimization also optimizes the data structures used in the vectorized version to better handle vectorization. It's important to note that while the vectorized version of Limpet was ready before the start of this internship, the GPU version was in development during the internship and the first version was only released in late May.

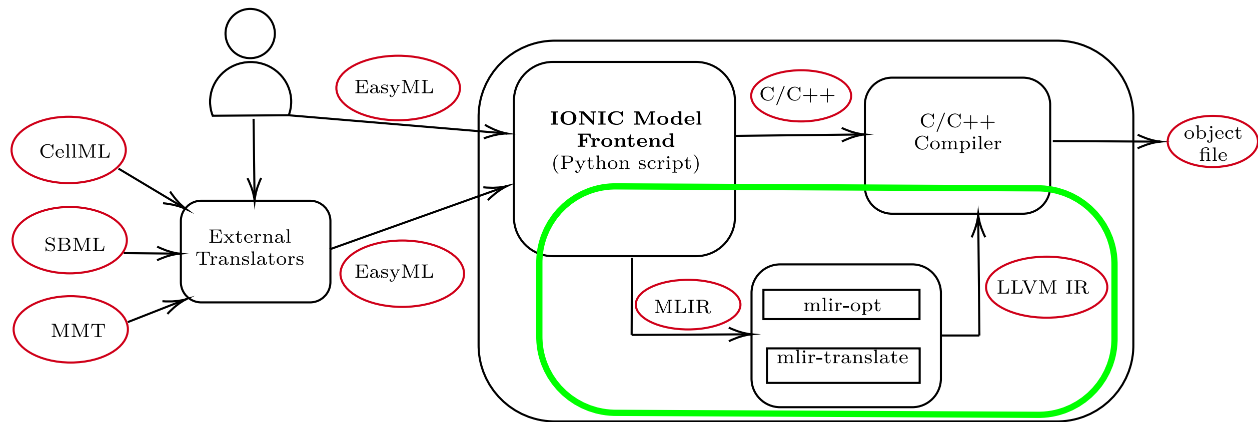


Figure 4: Compilation flow of Limpet with MLIR code generation (from [11]

In Figure 4 we can see the compilation flow of Limpet using MLIR code generation, the main difference with how Limpet works in Figure 2 is that Limpet also produces an MLIR-generated version of the code (either CPU+vectorization or GPU), this is in addition to the normal C/C++ code (in other words, both are generated).

For the hybrid (CPU+GPU) version we will produce in this internship, we will use both the GPU and vectorized versions provided by the Camus team.

3.1.2 Handling heterogeneity

The first objective of this internship is to adapt the code of Limpet for heterogeneous architectures. The decision of which tools to use to handle heterogeneity was already fixed at the start of the internship. We will therefore use StarPU (See section3.1.2.2) which is a task-based runtime system for heterogeneous architectures.

3.1.2.1 Different method of handling heterogeneity

To efficiently handle heterogeneity in Limpet with task-based programming, we envisaged two different methods of work distribution. Both of them come with their advantages and disadvantages so it's important to look into which one is more adapted for Limpet.

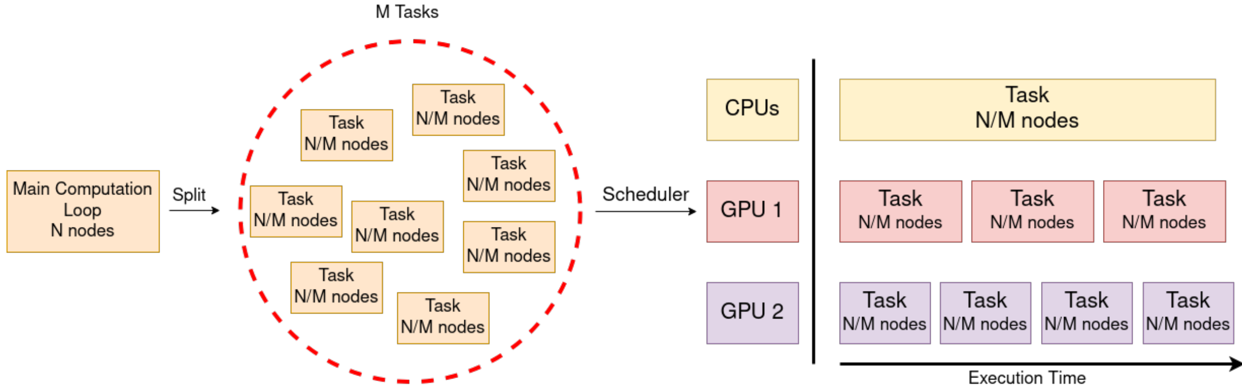


Figure 5: Simple example of handling heterogeneity with task scheduling

The first method would be to distribute work into a large amount of really small, equal-sized tasks, and by equal-sized, we mean that they contain the same number of nodes. The number of tasks should be large enough that we have way more tasks than devices. This method allows us to let a scheduler handle all the load balancing by distributing tasks to available devices. Since there already exist a lot of different scheduling heuristics for heterogeneous architecture that takes into account data locality, communication cost, and the performance of the different workers, for example, the Heterogeneous Earliest Finish Time scheduling heuristic or HEFT (See [18]), this solution is quite simple to implement.

However, the main drawback of this method is granularity. Granularity is a measure of the amount of work performed by a task, with coarse-grained parallelism happening when we have a small number of large tasks and fine-grained when we have a large number of small tasks. If tasks are too coarse-grained there might not be enough of them for the scheduler to be efficient and distribute enough work to every device. But if the tasks are too fine-grained there won't be enough computation to compensate for task overhead, leading to bad performance.

We can see an example of this method in Figure 5, here the scheduler used is *list scheduling*. It works by distributing tasks to each device as soon as they are free (in other words, aren't currently computing any task), leading to well-balanced work distribution. Of course, more advanced schedulers like HEFT can be used.

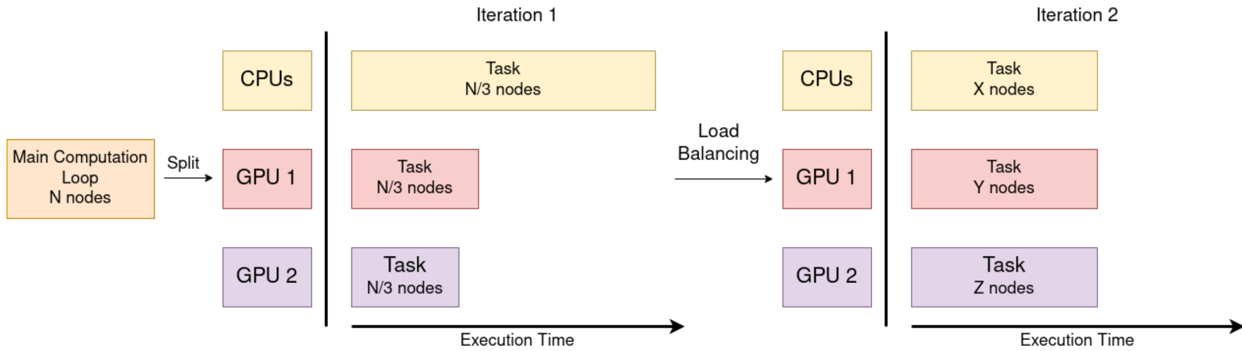


Figure 6: Simple example of handling heterogeneity with dynamic load balancing

The second method would be to fix the number of tasks in function of the amount of devices used. Then adjust the size of each task to fit the performance of the device they will run on. This can be done either by doing calibration runs to collect data on the performances of each device for a given task. Or by dynamically adjusting task size at execution. This allows for better control of task granularity but at the cost of having

to directly handle the load balancing (while scheduling would be handled transparently by StarPU).

Both calibration runs and dynamic load balancing also come with their overhead, which has to be taken into account. In Figure 6 we can see an example of dynamic load balancing, here we compute the first iteration (or timestep) with a naive distribution (equal amount of nodes on each device) to evaluate the performances of each device. Then in the next iteration, we distribute the new numbers of nodes for each task in function of their performances during the previous iteration.

In the case of OpenCARP, we selected the second solution, mostly because of granularity issues, but also because the communications between ionic models and the membrane potential solver force us to frequently use barriers to wait for all tasks to finish to collect the data needed by the solver. This dramatically limits the freedom of the scheduler, reducing the efficiency of the first solution.

3.1.2.2 StarPU

StarPU[3] is a task-based runtime system for heterogeneous architectures. It aims to help developers to better exploit the performance of such architectures transparently. StarPU internally deals with the low-level issues to enable the programmers to focus on algorithmic aspects. StarPU handles issues such as:

- Task dependencies
- Optimized heterogeneous scheduling
- Data movement and coherency

The primary data structures in StarPU are what we call a codelet. They are computational kernels with multiple implementations, for example one implementation for CPU and another for GPU. Tasks in StarPU are composed of a codelet and a data set. Running a StarPU task consists of running the codelet on a data set on one of the devices supported by the codelet. A task, therefore, describes the data it accesses, but it also needs to describe how it accesses them. In other words for each data structure accessed by a task the user needs to specify an access mode (either read, write, or both). StarPU then infers task dependencies from data dependencies.

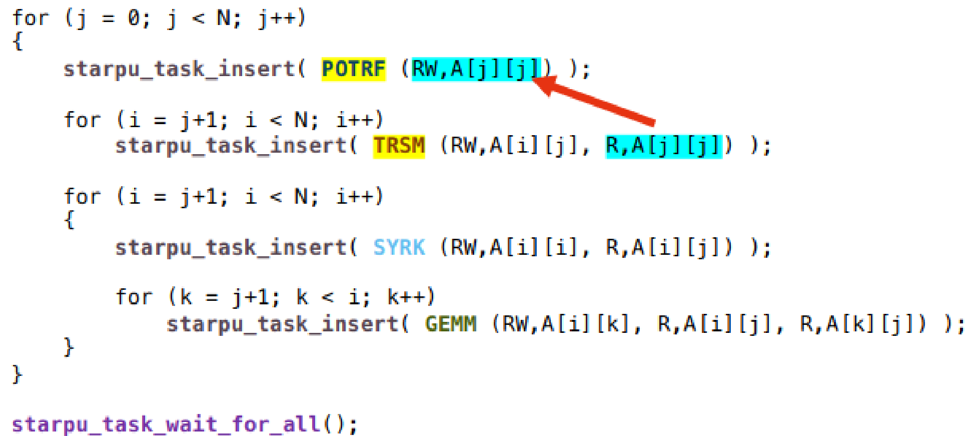


Figure 7: Example of StarPU task submission for Cholesky decomposition

In Figure 7 we can see an example of how StarPU tasks are submitted. On each `starpu_task_insert` we notice two parameters: the first one is the codelet (For example `POTRF`), and the second one is a list of data structures paired with their access modes, this is the data set of the Task. These data structures are represented in StarPU by what we call data filter. Here we can see the `TRSM` tasks access in read mode the same case of the matrix A as the `POTRF` tasks. This means that any `TRSM` tasks depend on all the previously submitted `POTRF` tasks and can only be executed once the tasks it depends on are completed.
This example results in the task graph in Figure 8. This task graph is automatically inferred by StarPU from the data dependencies between tasks. Here we observe that `TRSM` tasks indeed depend on previously submitted `POTRF` tasks.
The advantage of this system is that if the programmers want to support another architecture, for example, a codelet as an OpenCL and a CPU implementation and we want to add a CUDA implementation, since StarPU handles all data transfer between devices, the programmers do not need to write code to handle the transfer for the new implementation, he just needs to add a CUDA implementation to the codelet and the rest will be handled by StarPU.

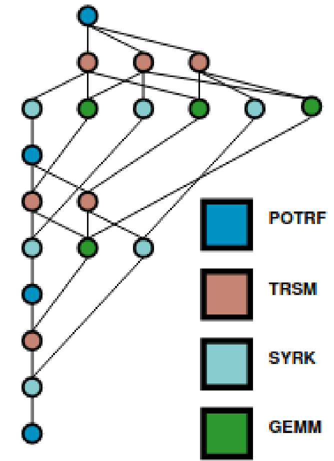


Figure 8: Resulting task graph of the code on figure 7

StarPU is well adapted for both solutions presented in section 3.1.2. In the case of the first method, StarPU comes with a built-in scheduler and a large amount of predefined scheduling strategies that are adapted to heterogeneous problems. For the second method, StarPU also comes with task profiling and tools to help with calibration runs. For those reasons we chose to integrate StarPU into Limpet to handle load balancing in OpenCARP

3.1.3 Handling precision Analysis

Our second goal in this internship is to run precision analysis to test how we can implement mixed precision efficiently in Limpet's code generation.

3.1.3.1 Introduction to floating point errors

Floating-point errors all come down to the same issue, there is no way to represent every real number with a limited amount of bits. There is already an infinity of real numbers between 1 and 1.1, and we need to represent the range of numbers much larger than that. And many numbers such as infinite decimal numbers (for example, 1/3 or 1/6) cannot even be represented.

This is why we have to use floating-point numbers, which are approximate representations of real numbers. Floating-point numbers are similar to normalized numbers, they represent numbers in the form: $1.decimal \times 2^n$ in the IEEE 754 standard and are composed of three parts :

- The sign S : represent if the number is either positive or negative
- The mantissa M : where $M \in [0, 1[$. If we number the mantissa bits from left to right $m_1, m_2, m_3, ...m_n$ then $M = m_1 \times 2^1 + m_2 \times 2^2 ... + m_n \times 2^n$. The mantissa represents the digits after the decimal point of the number.
- The exponent E : represented by a biased number, it's real value is $E - bias$ (the value of bias depends on the precision, $bias = 127$ in single-precision). The exponent represents the scale of the number.

Therefore, given S, E and M an IEEE 754 floating-point number as the value $-1^S \times (1 + M) \times 2^{E-bias}$, this representation allows emulating a large amount of real numbers through a broad range of exponents allowing computation on very large or small scales.

However, most real numbers cannot be directly represented and are rounded toward the nearest representable real number. This leads to computations where we are doing operations between multiple inexact numbers, such operations can lead to even larger errors. This is what we call a rounding error.

Doing a floating point operation between two numbers with a very different exponent can lead to what we call absorption error. For example, if we try to add $3*10^{-15}$ to $1.2*10^{20}$ we only get $1.2*10^{20}$, completely losing any information from the first operand because we don't have enough significant digits to represent $3*10^{-15}$ on such a large number.

Another common problem with floating point numbers is when we try to subtract two numbers subject to rounding errors (or by performing addition on two numbers of opposite signs). Since a subtraction on two real numbers with similar exponents can result in a number with a smaller exponent, this can lead to the error carried by one or both operands being bigger than the result and canceling it. This is what we call cancellation error.

Let us look at the following example[8]: If we look at the equation $b^2 - 4ac$, b^2 and $4ac$ are both results of floating point multiplication and thus are subject to rounding errors. if we say for example than $b = 3.34$, $a = 1.22$ and $c = 2.28$, then the exact value of $b^2 - 4ac$ is 0.0292. However, in half-precision b^2 rounds to 11.16 and $4ac$ rounds to 11.12, so we have $11.16 - 11.12 = 0.04$, which is a completely accurate calculation by the way. We can see that even though the subtraction of the two values introduces no error, this computation results in a value that shares no digit with the exact value. This is because the relative errors carried by b^2 and $4ac$ are of exponents greater or equal to the one of the exact value and therefore completely cancel out all significant digits of the result.

3.1.3.2 Planned experiments

To reflect on how and where we could implement mixed precision in the code generated from the ionic models, we chooses to run precision analysis to emulate the following methods of using mixed precision:

- **Switching an entire model to a lower precision**: Before testing the more sophisticated method of using mixed precision, it is important to test whether some models are robust enough to handle lowering the precision of the entire computation without damaging the accuracy of our results too much. Those tests will also allow us to check if some models are already badly affected by floating point error even with the whole computation running in double precision. In other words, this will allow us to have a general idea of the quality of the models.

- **Switching some parts of a model to a lower precision**: Not all floating-point operations produce floating-point errors, and typically some parts of the computation will produce or even worsen those errors more than other parts. Following this idea, we can run precision analysis while lowering precision on the part that does not tend to produce or amplify errors while keeping the problematic parts in double precision. This will also allow us to have a better understanding of how each part of the code affects floating-point errors.

- **Switching precision during different periods of the simulation**: A lot of simulations involving ionic models use periodic stimuli in their computations. As we can see in Figure 9 this results in computation where the values of variables fluctuate a lot after those stimuli, then return to a more stable state. In other words, maybe only lowering precision during those stable periods will help reduce the effect of lowering precision on accuracy. Another intuition is that maybe if we lower the precision in an early period of the simulation, the errors will have more time to be amplified than if we reduced the precision at a later stage.

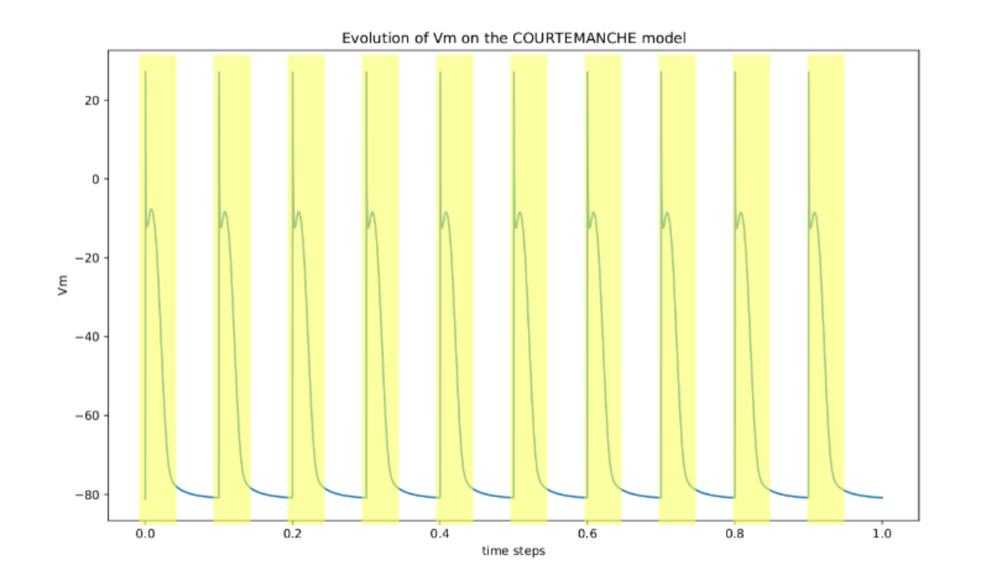


Figure 9: Periodic stimuli in the *COURTEMANCHE* ionic models

3.1.3.3 FPTaylor

At the early stage of the internship, we first considered FPTaylor[17] as a tool to perform precision analysis on our models. FPTaylor works by using the floating-point operations of a computation and the range of value in which they operate to create a Taylor approximation formula able to over-approximate the resulting floating point error of the computation, this result in an upper limit of the error we can expect.

The main advantage of this approach is that it does not depend on the input, the resulting over-approximation depends solely on the formula. This means that we can guarantee that we will not get an error worse than the upper limit no matter which set values are used during the computation. FPTaylor also produces an approximation that is more accurate than the one resulting from methods using interval arithmetic.

This method is perfectly appropriate to fix an upper limit of the error on different models depending on precision. It would also be possible to approximate only some parts of the model to detect how each part affects the accuracy.

However, the method used by FPTaylor comes with some drawbacks. First, this method is ill-fitted if we want to estimate how the different periods of a model affect floating point error, as we would use the same formula resulting in the same over-approximation. FPTaylor also has some problems handling conditionals like if-else. For these reasons, we later decided to use the tools described in the next section to handle precision analysis.

3.1.3.4 Verificarlo

To run precision analysis on the ionic models, we choose to use Verificarlo[6]. Verificarlo is a tool for debugging and assessing floating point precision. It can be used to estimate the number of significant digits in a computation, but also estimate the impact of precision change and explore the compromise between performance and precision.

To evaluate the number of significant digits, Verificarlo relies on Monte Carlo Arithmetic (MCA). To be more precise, it uses LLVM to replace all floating point operations with Monte Carlo Arithmetic equivalents, applying some randomization to the operations. This allows us to use each execution as a trial of a Monte Carlo simulation. With enough samples, these results can be used to statistically estimate the floating-point error in a computation. Those results can be used, for example, to estimate the number of significant digits

of the result of a computation covered by Verificarlo. We will define how Verificarlo uses Monte Carlo Arithmetic to simulate floating point error in section 3.1.3.5

Verificarlo comes with the following functionalities:

- Can simulate either round-up error or cancelation error (see section 3.1.3.1) or both. This can help identify which type of floating point error is causing problems.
- Can be configured to only cover some functions of the code. This can be used to detect which parts of the code cause a superlinear propagation of the error by observing how the error grows from the inputted values to the outputted values. In other words, if the errors in the output as grown worse than the input we can deduce this part of the code is amplifying errors.
- Verificarlo can be used to add noises to floating-point operations to simulate lower precision, such as single or half-precision. This allows evaluating the impact of switching to a different precision without having to modify the code.

However, Verificarlo comes with the inherent drawbacks of random methods like Monte Carlo Arithmetic. The results are only empirical estimations, and not guaranteed upper limits of the potential floating point error, unlike FPTaylor. The results obtained from Verificarlo are also very dependent on the set of values used for the trial runs. Running the same experiments with a different set of values (especially when using ones of different magnitude) can lead to very different results.

We choose to use Verificarlo for precision analysis since it comes with most of the tools we needed to explore the compromise between precision and performance. It also conveniently uses LLVM which is already present in the OpenCARP version we are using in this internship (described in section 3.1.1). Since ionic models are deterministic, OpenCARP mostly does the same computation with the same values for a given model and is, therefore, less affected by the drawbacks of Verificarlo.

3.1.3.5 Monte Carlo Arithmetic (MCA) in Verificarlo

In this section, we will explain how Verificarlo adds noise to floating-point operations to simulate floating-point errors. And how do we measure those errors.

Verifircarlo uses Monte Carlo arithmetic to model both rounding and cancelation errors. It can model such errors for any possible virtual precision t, where t is the size of the mantissa of the desired virtual precision, to run experiments with other floating-point precisions, such as single or half precision. To model errors on a given floating point value x at virtual precision t, Verificarlo uses the following function[14]:

$$inexact(x) = x + 2^{e_x - t}\xi$$

Where $e_x = \lfloor \log_2 |x| \rfloor + 1$ represents the order of magnitude of x and ξ is a uniform random variable in $]-\frac{1}{2}, \frac{1}{2}[$ (which serves to randomize our results). Here, t is used to represent the magnitude of the errors, as we can see in Figure 10.

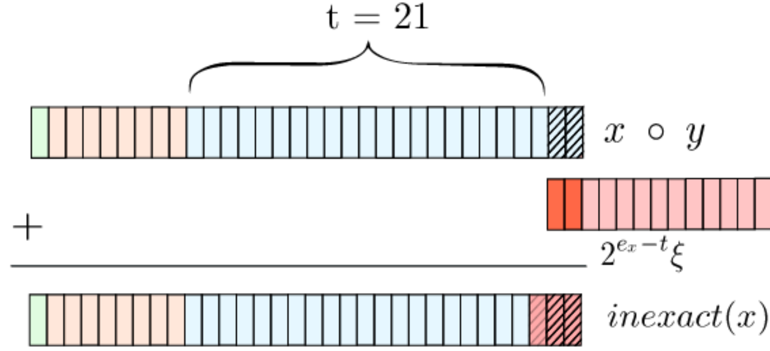


Figure 10: MCA random rounding for $t = 21$ on a single precision value (23 bits of mantissa) (from [13])

Each floating-point operation $x \circ y$ is then replaced by an MCA operation depending on which errors we want to represent (with $round(x)$ a function that takes a real number x and returns the nearest representable floating-point value):

- For rounding error and absorption only: $x \circ y \rightarrow round(inexact(x \circ y))$
- For cancellation error only: $x \circ y \rightarrow round(inexact(y) \circ inexact(y))$
- For both : $x \circ y \rightarrow round(inexact(inexact(x) \circ inexact(y)))$

Now we know how Verificarlo uses MCA to randomize the result and estimate errors. Now we need a way to measure the accuracy of our results from a set of trial runs. For this, we can compute the number of significant digits of our results. The number of significant digits of $round(x)$ is defined by the number of digits, starting from the digit with the highest exponent (for example, starting from 1 in 0.0001234), which are accurate representations of x (i.e., if $round(x) = 1.111$ for $x = 1.110$ then the number of significant digits of $round(x)$ is 3).

To statically estimate the number of significant digits s in base β of a given computation using MCA, Parker proposed the following formula[14]:

$$s = -\log_\beta \frac{\sigma}{|\mu|}$$

Here, σ is the standard derivation and μ is the mean distribution of the results. Since we do not know the exact distribution of our results, we need to estimate it empirically by using a large number of Monte Carlo trials. We will use 10 trials for our experiments, which should be sufficient to perform simple tests.

3.2 Planning

We will describe in this section the progress of this internship. In particular, since some of our objectives depend on the work of the INRIA CAMUS team, we will describe how the progress of their work affected our decisions. Similarly, we will also describe the events that led us to change our decisions or priority.

In Figure 11 we can see our progress for the first half of the internship (07 February to 29 April 2022). The internship started with reading papers about different tools and methods of precision analysis. The objective was to understand which tool we needed to run the precision analysis on OpenCARP and how we could implement it. We initially choose to use FPTaylor (see Section 3.1.3.3) as a tool to estimate floating-point errors instead of Verificarlo. To implement FPTaylor we decided to use MLIR to create a new Intermediate representation to recover the upper limits of floating point errors of our computation during compilation. This led us to spend some time learning MLIR for this purpose.

However, on 14 April, we had a visit from Pablo Oliveira, who works on Verificarlo, to discuss precision analysis. We discussed together on the drawback of using FPTaylor, but also about Verificarlo. After gaining a better understanding of Verificarlo, we chose to use it instead of FPTaylor to run our experiments since it was both easier to use and more adapted for our objectives.

During this time we also added StarPU to Limpet and started developing the multi-tasks CPU version (see section 3.4.1). Most of the time spent on this part was due to learning the code of OpenCARP, there was no overhead to learning StarPU as we already used it on a previous internship (See [2]) on a similar subject. We also planned a one-week visit to Strasbourg. The purpose of the visit was to meet people in the INRIA CAMUS team who are working on the MLIR code generation (see Section 3.1.1) to discuss how to better implement StarPU in their version of Limpet. We also discussed the future GPU version and what they had planned at the time. This visit required us to produce the first version of our multi-task implementation to show them what we had planned.

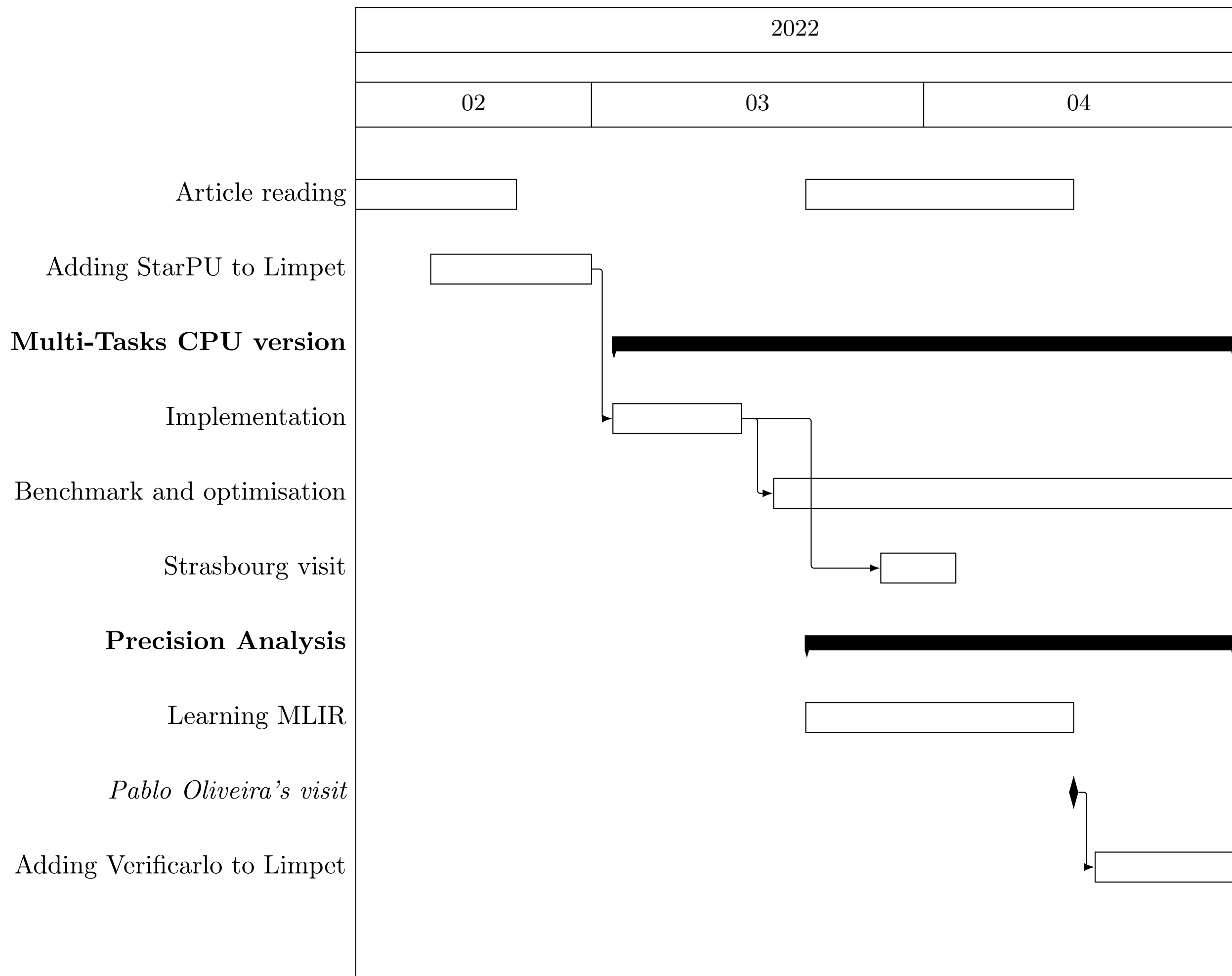


Figure 11: Time chart of our work during the internship (February-April)

Let us now look at our progress for the second half (5 May to 29 July 2022) of the internship in Figure 12. We started May by finishing what was left for the multi-tasks CPU version, but also the implementation of Verificarlo into Limpet. We then started to do some experiments on precision analysis but stopped once the GPU version was released (late May). Our focus switched to implementing the heterogeneous versions of limpet (see multi-GPU and hybrid versions at the start of section 3.4) as we wanted to show results at the

Microcard workshop (06-07 July 2022).

However, the GPU version is still in development and at the time of the internship had several performance issues, making it slower than the base OpenCARP version. We also ran into multiple difficulties implementing our heterogeneous versions. We later concluded that to produce efficient heterogeneous versions of Limpet, some features needed to be added to the GPU versions. For those reasons, we decided to instead fully focus on precision analysis for the Microcard workshop and the rest of the internship.

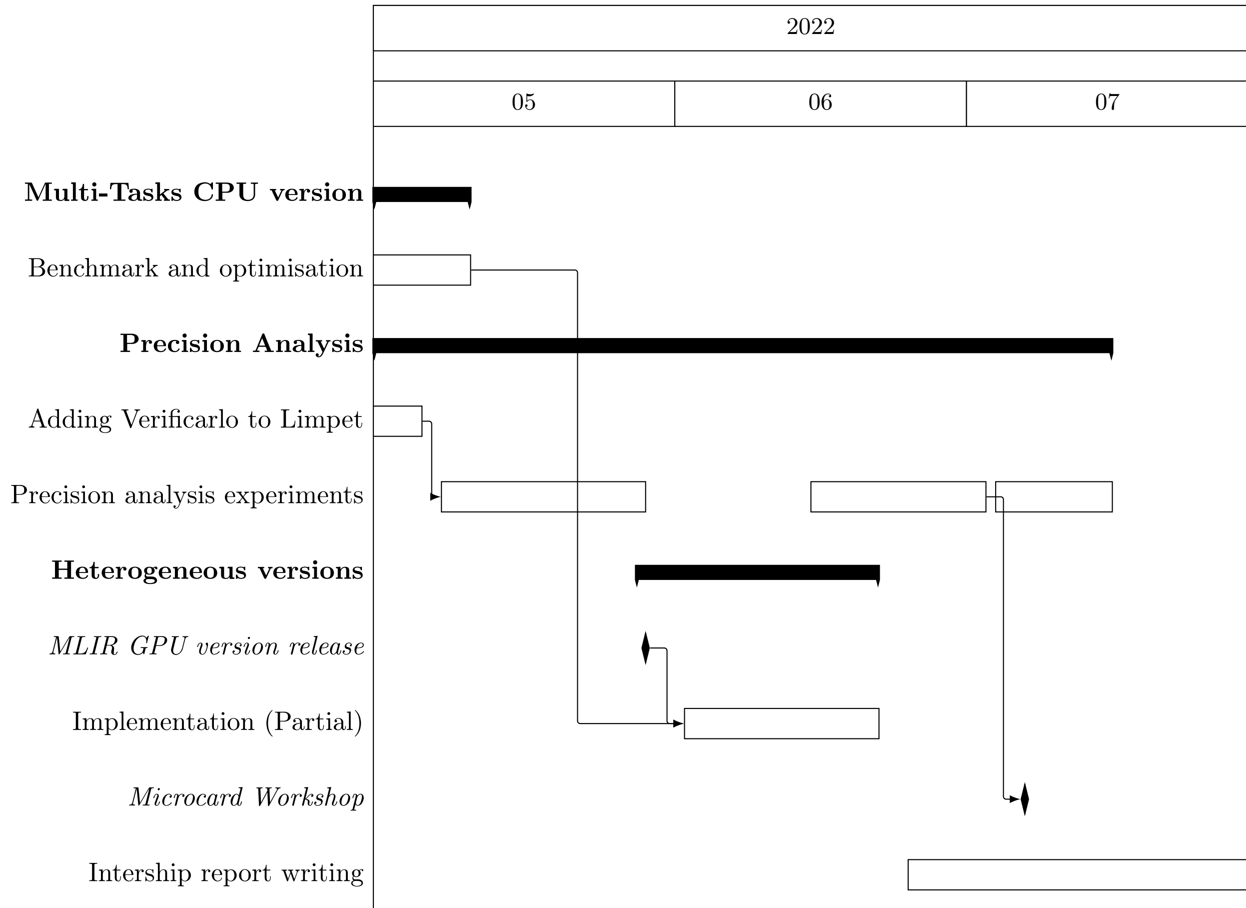


Figure 12: Time chart of our work during the internship (May - July)

3.3 Integration of the new tools into the OpenCARP framework

Before starting to work on the problematic, we need to integrate StarPU and Verificarlo into the OpenCARP framework and make them work together with the MLIR code generation version. This includes dealing with any compatibility issues between those tools. Therefore, in this section, we will describe the work conducted to implement these tools together into the framework.

We will be starting from the version produced by the INRIA CAMUS team using MLIR (See section 3.1.1). Therefore, any work needed to add this version to the base version of OpenCARP was not done by us. However, since the MLIR code generation version was still in development there was some upkeep throughout the internship to stay updated with the development of the vectorized and GPU version.

3.3.1 Adding StarPU

The next step was to include StarPU. This required us to change the code generation for both the C/C++ and MLIR versions for the generated code to use tasks. We also had to change the precompiled used in Limpet for them to be able to use a task-based version of OpenCARP code. This includes adding all the necessary data structures for StarPU to handle tasks, changing the signature of multiple functions to make task size parameterable (this will be needed later for the heterogeneous version), and replacing the computation loop of the ionic models with a task-based version.

There were no compatibility issues between StarPU and the MLIR code generation. However, compared to the C/C++ version of the code, We had to handle StarPU differently for the MLIR version of the models to run them efficiently. Those differences will be described later in section 3.4

3.3.2 Adding Verificarlo

Since Verificarlo works using an LLVM compilation pass to convert floating point operations, to add Verificarlo to the OpenCARP framework we had to modify the compilation chain to apply this pass on the ionic model code and every source code used in the computation.

While adding Verificarlo did not add any compatibility issue for StarPU, the LLVM version used for the MLIR code generation was not compatible with Verificarlo. The MLIR version of limpet requires LLVM 15 while Verificarlo was only compatible with LLVM versions up to 11. We had to modify Verificarlo to be compatible with LLVM 15 while keeping it retro-compatible with LLVM 11 as it was a requirement of the project.

The compatibility issue was due to a change in how vectors are handled starting from LLVM 13, and the use of a new pass manager which was incompatible with Verificarlo's code.

We also had to modify Verificarlo to do some of our experiments. We needed the functionality to change the floating-point precision in the middle of a simulation, which was not possible in the current Verificarlo version. This led us to add new user calls to Verificarlo to change the precision during the execution of the program.

3.3.3 Final version

In Figure 13 we can see the compilation flow of Limpet after our contribution. Nodes in yellow represent everything related to the MLIR code generation, green for every part where StarPU is concerned, and blue for Verificarlo. The nodes with the red border are parts of Limpet modified or added by our contributions.

The overall workflow is mostly the same as the base version, the ionic model compiler converts the EasyML model into two versions: the C/C++ and MLIR. The difference is that both versions of the generated code and the precompiled are modified to use task-based parallelism. The MLIR version is run through multiple optimizations passes by the MLIR optimizer, then both versions go through Verificarlo to replace all instrumented floating-point operations. The Verificarlo version of the code is then handled by LLVM. Once all models are compiled, all the ionic models and related pre-compiled are added to the ionic model library to be used by the OpenCARP simulator.

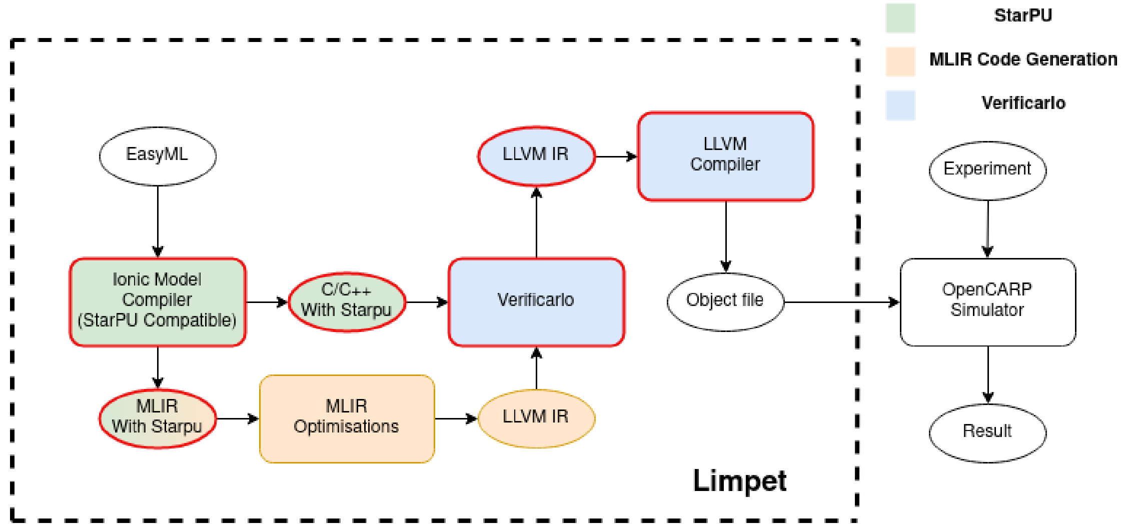


Figure 13: Workflow of Limpet after our modification

3.4 Implementation of Limpet's heterogeneous version

In this section, we will describe our contribution to developing a heterogeneous version of the code generated by Limpet. However, since the GPU version wasn't fully ready by the end of this internship, we couldn't finish producing the heterogenous version. For this reason, in this section, we will also describe what remains to be done to achieve the heterogeneous version.

To better understand all the versions of Limpet that were produced before and during this internship or will be produced in the future, and how they relate to each other, we will describe the different versions here:

- **Base version (OpenCARP):** This is the original version of Limpet's code generation, this version works on CPU and handles parallelization for multiple CPUs with OpenMP. However, this version lacks vectorization.
- **Vectorized version (University of Strasbourg):** This version is based on the base version but produces a vectorized version of the code. It uses MLIR for code generation.
- **GPU version (University de Strasbourg / still in development):** This is the GPU version of Limpet's code. It uses MLIR for code generation, and CUDA is used for the computation on GPU.
- **Multi-Tasks CPU version (our contribution):** This version is based on the vectorized version and therefore uses MLIR for code generation. This version splits the workload into multiple tasks and distributes them between the CPUs. This differs from the vectorized version where all CPUs work on the same loop.
- **Multi-GPUs version (our contribution / still in development):** This version is based on the GPU version and therefore uses MLIR for code generation. It splits the workload into multiple tasks to run on multiple GPUs. This version is heterogeneous if different GPUs are used and need load balancing.
- **Hybrid version (our contribution / still in development):** This version combines the vectorized version and multi-GPU version and therefore uses MLIR for code generation. It splits the workload into multiple tasks to run on CPUs and GPUs. This version is heterogeneous because of the differences in performance between CPU and GPU and needs load balancing.

3.4.1 Multi-Tasks CPU version

The first step of integrating StarPU into Limpet is to add StarPU to the CPU version of the code. The base CPU code is already parallelized using an *OpenMP parallel for* loop. Since all the iterations of the loop are the same independent computation, and since the CPU version is a homogenous problem, the current implementation of OpenCARP is already good enough. Therefore the goal of adding StarPU to the CPU version isn't to gain better performance but to lay the groundwork for the heterogeneous version, especially since the GPU version was not ready at the start of this internship.

We mostly just need to replace this loop with a task version of the code. There are two ways we can convert the loop, both versions have their different advantages and disadvantages:

- The first method is to simply encapsulate the loop into a task and let OpenMP do the parallelization, StarPU uses a type of task called parallel task to achieve this. This is the simplest method, it allows for very low task overhead and good granularity in most cases. But it will also perform badly in cases where we have multiple very small regions, as we can only handle each region sequentially with this solution, resulting in bad granularity.
- The second method is to split the loop into different tasks, proportional to the amount of CPU for equitable work distribution. This method enables better control of task granularity allowing us to adapt task size to the size of the caches. For example as we can see in the figure 14, the loop is split into K tasks, where K is multiple of the number of CPU selected in such a way that task size is small enough to fit into an L2 cache. However, this solution also causes more task overhead.

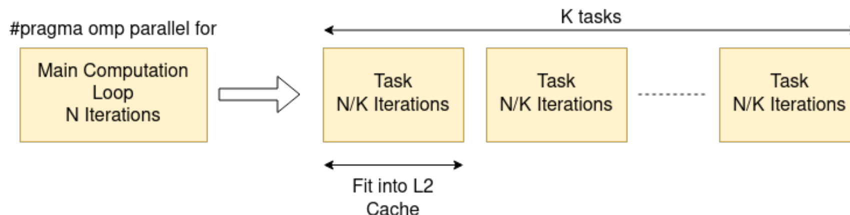


Figure 14: Conversion of a loop into multiple tasks

In terms of performance, the first solution should give better results, as parallel tasks handle granularity better and lead to less overhead (See [4]). This is more efficient than splitting the same code into multiple small tasks (in which case we suffer a lot more from task overhead). However since we're going to need to handle multiple tasks for the heterogeneous versions, we choose to implement the second method to start working on adapting the code for handling multiple tasks. To implement the multi-tasks CPU version and the future heterogeneous version, we need:

- Parameterable task size as described in Section 3.3.1
- Some way to distribute work among the different tasks (while the ideal method of distribution differ between the CPU versions and the heterogeneous versions, the overall code structure is the same)
- Adding data filter to the main data structure of the computation for StarPU to handle data transfer between devices.

Working on a multi-tasks CPU version will allow us to work on all those requirements before the GPU version is released.

To run those tasks efficiently, we need to pay close attention to how vectorization is handled in the MLIR version of Limpet. We will use AVX512 for vectorization, so our vectors can hold 512 bits of data, as we manipulate the data structure filled with double precision values (64 bits), the vectorization can handle

$512/64 = 8$ values at the same time. However, if we handle computation where the number of nodes isn't a multiple of 8 then we cannot vectorize some of the nodes, so some of the nodes need to be computed without vectorization.

On one task this means that at most 7 nodes will be handled sequentially, however, this can result in a lot of nodes being handled sequentially if we use a lot of tasks. If we look at the example figure 15, we can see that with 100 nodes 4 of them will not be vectorized. However, for 10 tasks, if we naively split the work into 10 equal parts, then each task iterates on 10 nodes with 2 of them handled without vectorization, for a total of 20 non-vectorized nodes. In other words, it is 5 times the amount compared to the version with one task.

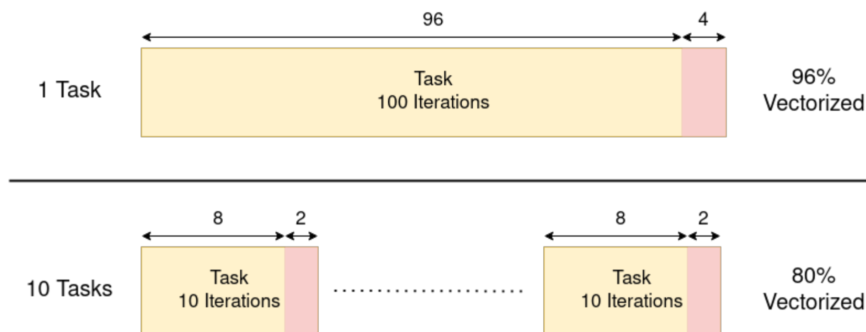


Figure 15: Proportion of vectorized code depending on the number of tasks using AVX512

Needless to say, this can lower performance quite significantly in some cases. To avoid this problem, we have to adjust the task size so that the number of nodes in all tasks (except the last task if the total number of nodes is not a multiple of 8) is a multiple of 8. This can be done using Algorithm 1. This algorithm works by splitting the nodes equally among each task, then if any task has a number of nodes that is not dividable by 8, it removes those nodes from the tasks and then redistributes the remaining nodes among the tasks by groups of 8. Any rest is distributed to the last task.

Algorithm 1 Compute vector-conscious task sizes for n tasks, x nodes, and a vector size v ($v = 8$ in AVX512)

$rest \leftarrow x\%n$
$size \leftarrow x/n$
$rest \leftarrow rest + (size\%v) * n$
$z \leftarrow rest/v$
$rest \leftarrow rest\%v$
for $i = 0$; $i < n$; $i++$ **do**
  $taskSize[i] \leftarrow size$
  if $z > 0$ **then**
    $taskSize[i] \leftarrow taskSize[i] + v$
    $z \leftarrow z - 1$
  end if
end for
$taskSize[n-1] \leftarrow taskSize[n-1] + rest$
return $taskSize$

To compute the number of tasks n we use the following formula : $n = cell_y((x \times s)/L2_{size})$, where s is the size of a node in memory, $L2_{size}$ the size of the L2 cache and $cell_y(X)$ a function that returns the nearest multiple of y that is superior or equal to X. We use $cell_y$ to guarantee that for y CPUs, every CPU will have the same number of tasks.

It is important to note that to guarantee that tasks fit into the L2 cache, we have to take into account that the task sizes output by Algorithm 1 are not equal, some tasks will be at most 8 nodes larger than others. This issue may cause some tasks to not fit in the cache even with our method. We also want tasks to not completely fill the cache to fit the lookup table in the cache. To take these issues into account we modified the formula to $n = cell_y((x \times s)/(L2_{size}/2))$, this will produce a greater amount of smaller tasks.

3.4.2 CPU Version experiments

The experiments in this section were run on plafrim's Bora clusters using the following architectures: 2×18 *core Intel CascadeLake,* 192 *GB Memory (*1024 *for L2 caches), and AVX512 for vectorization. All experiments in this section use only one region.*

We remind once again that we do not expect speedup from the multi-tasks CPU version as the problem is homogeneous and does not profit much from task-based parallelism while still suffering from task overhead. However, observation of task overhead on the CPU version can allow us to predict the effect of task overhead on the GPU and hybrid versions. This allows us to work on minimizing such issues without having access to the GPU version of OpenCARP that was not released in the early stage of the internship.

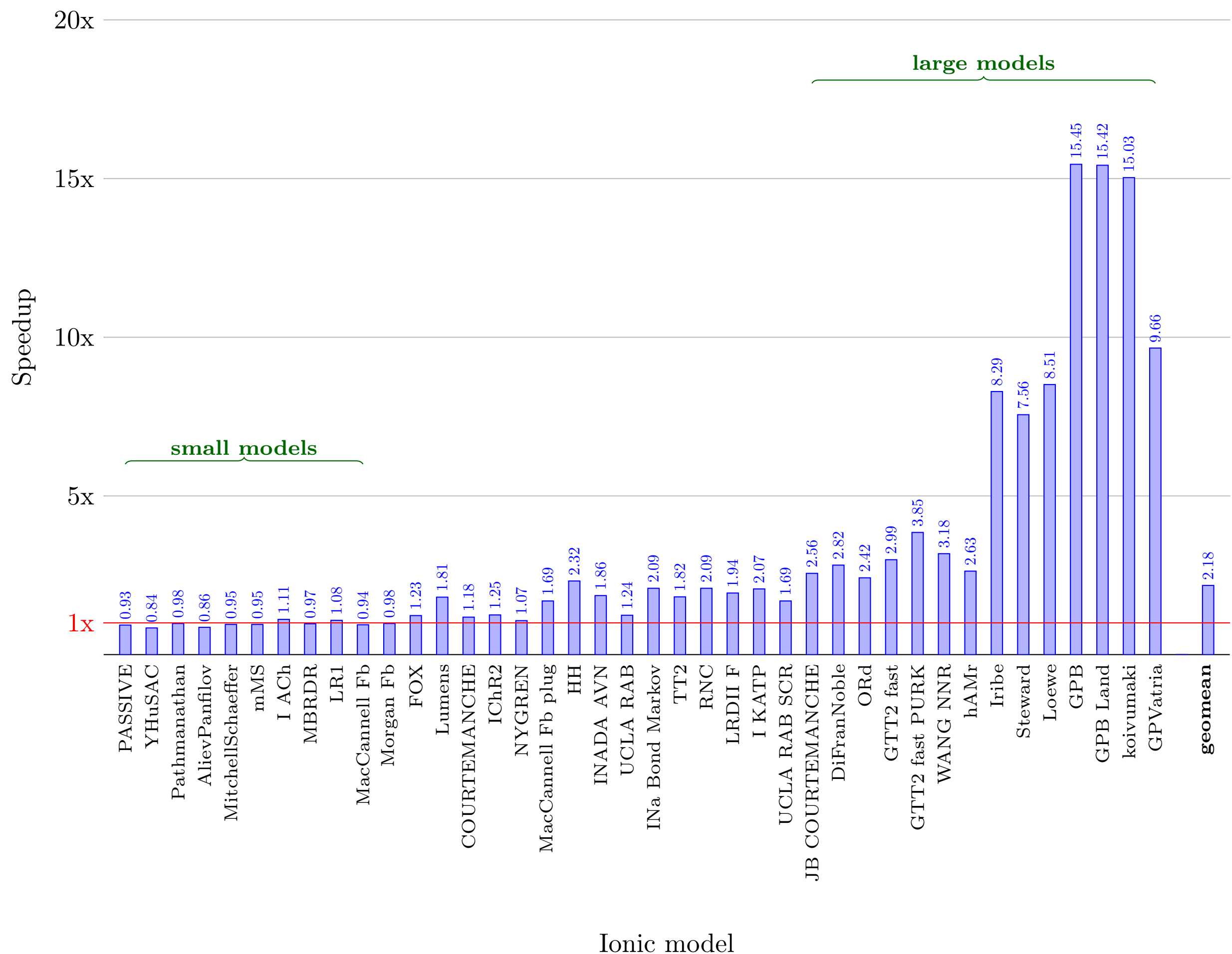


Figure 16: Speedup of the MLIR version of the code compared to the original code, using 32 OpenMP threads on a 32 cores AVX512 architecture, for 8192 nodes and 10000 steps. (From [11])

Firstly, to fully understand the results obtained in our experiment, it's important to take a look at the

performance of the MLIR version which we are based on for our StarPU version and the effect of model size on performance. In the figure 16, we can see the performance obtained by the MLIR version (without StarPU) compared to the OpenCARP's base version. Only the models that are handled by the MLIR compilation pass are represented in this plot. The models are sorted by size, with the smaller models on the left and the larger models on the right. Models are roughly grouped into three categories depending on how long they took to complete with the base version of Limpet: below 1 minute for small models, between 1 and 5 minutes for medium models, and above 5 minutes for large models.

We can observe larger models tend to get better speedup with vectorization, while the small models are barely affected. This is because it is much easier to get speedup on large computations compared to the smaller models where barely any time is spent on computation. However, the speedup obtained isn't strictly linear compared to the size of the models. This is because some models use lookup tables a lot, while some models barely or don't use them. This matters for speedup because models with no lookup tables contain a lot of computation relative to their size.

Now we can look at the performance of our StarPU version. As we can see in figure 17, the MLIR + StarPU version obtains slower performance on all models, as expected. However, the performance gap between the 2 versions is larger on smaller models. This is mostly because task overhead becomes more noticeable for models with fewer computations. On those models, the performance of the StarPU version can be as much as 2 times slower. However, in large models, the gap is only around 5% to 15%, which is much more acceptable.

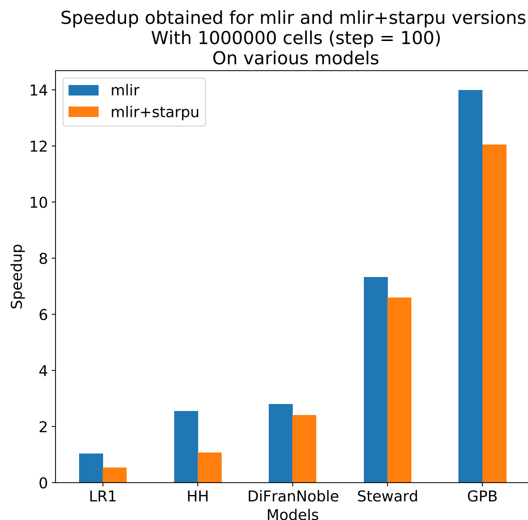


Figure 17: Comparison of speedup obtained on different models, between the MLIR and MLIR + StarPU versions compared to the original code, on 1M nodes and 100 steps

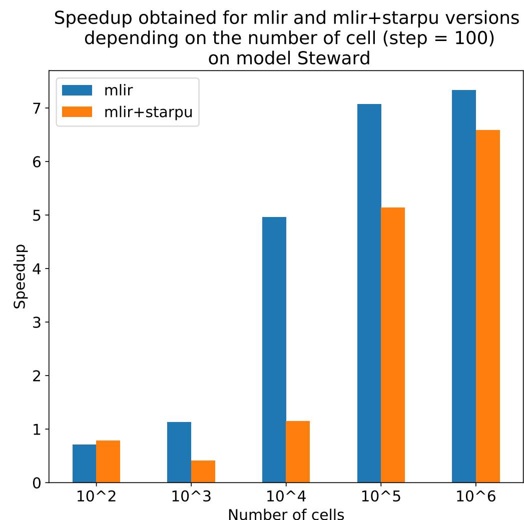


Figure 18: Comparison of the speedup obtained in the Steward models, between the MLIR and MLIR + StarPU versions compared to the original code, depending on the number of nodes (100 steps)

Now, in figure 18, we can look at the comparison of performance obtained depending on the number of nodes. This comparison is made on Steward, which is considered one of the large models. Here, we can see that the difference in performance becomes smaller as more nodes are covered in our computation. The only exception to these rules is in the case where we have only 100 nodes. This is due to the way we choose the number of tasks in function to the number of nodes. When the number is too low to create tasks with enough nodes for each CPU (this means at last 8 because of vectorization), then we default to using one task shared by all CPUs exactly like we described in the first method of section 3.4.1. As for the cases where we

have enough nodes to use multiple tasks, we can observe that it is vital for the problem to be large enough to not experience a huge loss in performance (StarPU version 5 times slower for 10000 nodes).

We can conclude from those experiments that the effect of task overhead on performance, while noticeable on small problems, becomes less severe when we have a lot of computation. This cost should get smaller on the GPU version as the number of tasks will get smaller as we do not need to feed all the CPUs with multiple tasks each. However, it is clear that without some form of task dependency optimization, the multi-tasks approach will not work well for the hybrid version. It would be preferable to use one task shared by all CPUs for the part not handled by the GPU.

3.5 Precision analysis experiments

In this section, we will discuss the result of our experiments using Verificarlo to run precision analysis on the code generated by limpet.

These experiments are run to observe how the precision of floating-point computation affects the accuracy of the simulation; for this, we use Verificarlo to simulate a lower precision by adding noise. In other words, the floating point operations are not really replaced by a lower precision one, and so, our computation does not benefit from speedup we would normally expect by lowering precision. On the contrary, the computations run with Verificarlo are much slower because of the operations needed to add noise to floating-point operations. Therefore, those experiments are in no way a means to observe how precision affects execution time.

These experiments are also very long. For experiments done on 1000 nodes and 5000 timesteps with 10 runs,the time it takes to do precision analysis with Verificarlo on one model for one configuration can take between $3h$ and $32h$. Since we are still exploring different possibilities for precision optimization and doing precision analysis with Verificarlo takes a lot of time, we can't afford to exhaustively test every model with every possible configuration. Therefore, we only ran precision analysis on a sample of model. We will present in this section some of our results.

All the experiments in this section will be run on Plafrim's Bora clusters on a simulation with 1000 nodes and 5000 steps.

3.5.1 Precision modification on an entire model

To understand how floating point precision affects the accuracy of a model, we first want to look at what happens if we switch the precision of the whole model from double precision to single precision or even half precision. This will allow us to observe if some models can be completely switched to a lower floating-point precision. These experiments will also enable us to know if some models already have accuracy issues in double precision. In this and all the following sections, we will evaluate accuracy by observing the number of significant digits of our results (in base 2).

We will first run our experiment on the *COURTEMANCHE* ionic model; this is a medium-sized model that uses a lookup table in its computation. First, we want to observe what happens when we degrade precision in simulations where we use only one stimulus at the start of the computation (which is the default setting of an ionic model simulation in Limpet).

We can see in Figure 19 the potential of the membrane V_m of the model in a simulation with a stimulus at the time step 0. We can see that the membrane potential increases dramatically immediately after the stimulus and then gradually decreases until it stabilizes approximately 400 steps later. Once V_m starts to stabilize, its value stays roughly the same until the end of the computation. Similar patterns can be observed for the value of V_m in most ionic models. We will call the time steps after a stimulus where the value of V_m is unstable, the *active periods* of our computation, and we will call the steps when the value of V_m is stable, the *stable periods*. For example, in this graph, the *active period* is from 0 to 400, and the *stable period* from 400 to 5000

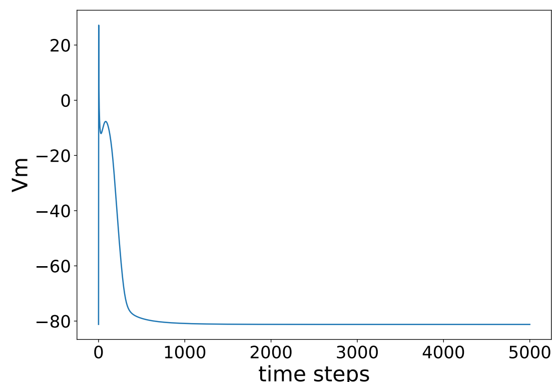


Figure 19: Membrane potential (V_m) depending on timesteps in the *COURTEMANCHE* model (1 stimulus)

In Figure 20 we can see the number of significant digits of the action potential V_m depending on the virtual precision (52,23 and 10). Interestingly, we can notice that all three levels of precision experience a large dip in accuracy during the active period of the simulation. Accuracy also stabilizes for all precisions during the stable period. After that, the accuracy stays roughly the same for the entire stable period.
The accuracy of the membrane potential stabilizes around 24 bits during the stable period for double precision ($t = 52$) and only drops to 21 bits during the active period. For single precision ($t = 23$) the accuracy reached 10 bits of mantissa during the active period but stabilized at 21 bits for the stable period. Finally, for half-precision ($t = 10$) the accuracy reached a very low point of 1 bit during the active period to later reach around 8 bits during the stable period. We notice that double-precision handles the drop during the active periods much better by only losing 3 bits during the active period compared to the stable period (versus 11 for single precision and 7 for half).

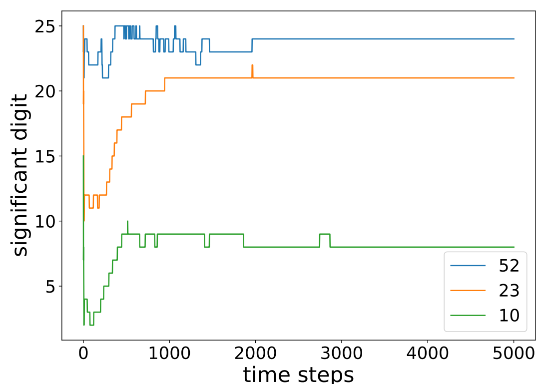


Figure 20: Number of significant digits for V_m in the *COURTEMANCHE* model depending on the number of iterations (one stimulus)

At first glance, our result seems to confirm our hypothesis in Section 3.1.3.2 that accuracy is mainly affected when the computation is very active (or the active periods as we call them), as the drop in accuracy seems to superpose with the active period of our simulation. This can be explained by the fact that, when values are stable, we are less at risk of encountering errors, such as absorption errors that mostly occur when the exponents of our values are very different. We also notice that the accuracy does not seem to worsen over time, as it stays the same throughout the stable period.

Let us now see whether these observations can be generalized to simulation with multiple stimuli; we will add a stimulus every 1000 steps. This experiment can be observed in figure 21, we can notice that an active period occurs for around 400 steps after each stimulus. In Figure 22, we can see the accuracy of

the membrane potential for different levels of precision in the same simulation (stimulus every 1000 steps). As expected, the first 1000 steps of our result are similar to those in Figure 20. However, after the second stimulus, a similar pattern starts to repeat at every stimulus: the accuracy drops in the active period and rises back to a stable level during the stable period.

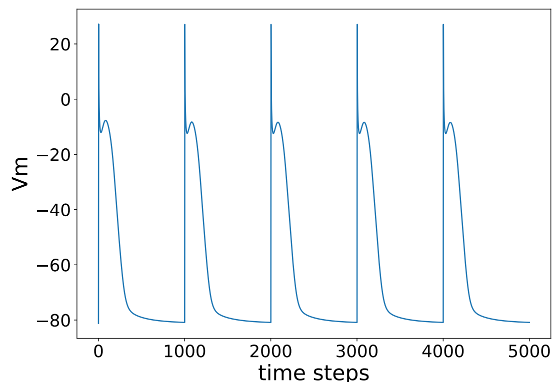


Figure 21: Membrane potential (V_m) depending on timestep in the *COURTEMANCHE* model (stimulus every 1000 steps).

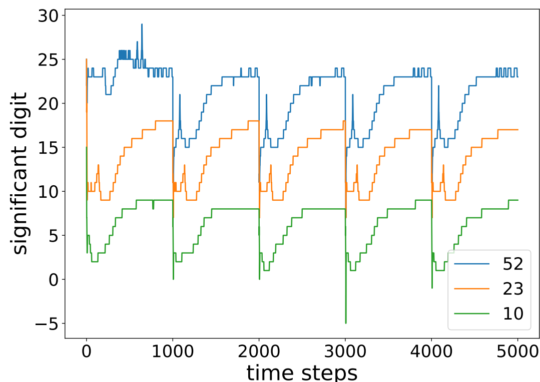


Figure 22: Number of significant digits for V_m in the *COURTEMANCHE* model, depending on the number of iterations (stimulus every 1000 steps).

We notice that in double precision, the drop in accuracy during the second active period is much worse than during the first (the accuracy drops to 20 after the first stimuli and 10 after the second). The accuracy drop then stops worsening for subsequent active periods. In contrast to the active periods, the stable periods result in a similar accuracy during the entire simulation for all precision. We can observe in those results that the accuracy does not seem to be linked to time steps (in other words, the overall accuracy does not degrade over time, except double precision having a more accurate first active period). Instead, in those experiments, the accuracy seems to be mostly related to the precision used and whether we are in an active or stable period.

Currently, these observations are only true for the membrane potential. Therefore, we need to look at different variables. Another variable commonly observed for cardiac stimulation is the transmembrane current (I_{ion}). Let us look at I_{ion} using the same experiment: one stimulus every 1000 steps, starting from step 0.

As we can see in figure 23, the behavior is very similar to V_m : an active period after each stimulus for about 400 steps followed by a stable period. Changes in values at the start of active periods are much sharper for I_{ion}, for the first 20 or so steps, the value of I_{ion} fluctuates between -80 and 10. The rest of the active periods are much calmer, with I_{ion} staying very close to 0.

In figure 24 we can look at the number of significant digits obtained for the same experiment on I_{ion}. We observe an inverse behavior compared to the membrane potential. The accuracy rises during active periods and decreases during stable periods. At first glance, it seems like our hypothesis that errors are worse during the active periods is false for I_{ion}

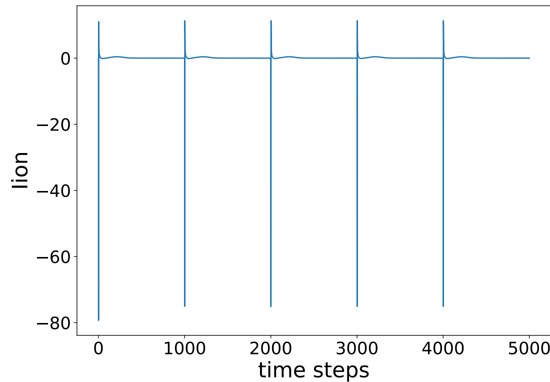


Figure 23: Transmembrane current (I_{ion}) depending on timestep in the *COURTEMANCHE* model (stimulus every 1000 steps)

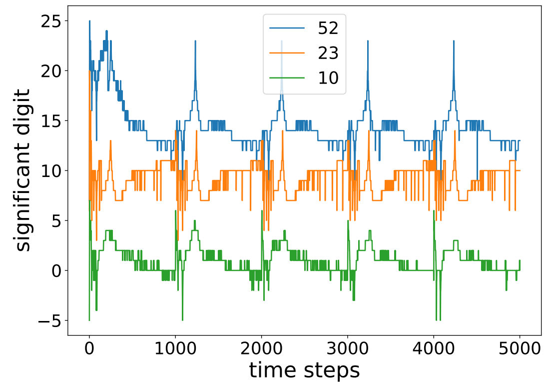


Figure 24: Number of significant digits for I_{ion} on the *COURTEMANCHE* model depending on the number of iterations (stimulus every 1000 steps)

However, if we look in Figure 25 at the standard deviation of I_{ion} for double precision, we notice that the standard deviation is rising during active periods and drops during stable periods. While this behavior is more in line with the hypothesis of errors being larger during active periods, we would expect the numbers of significant digits of I_{ion} to drop when standard deviation increases.

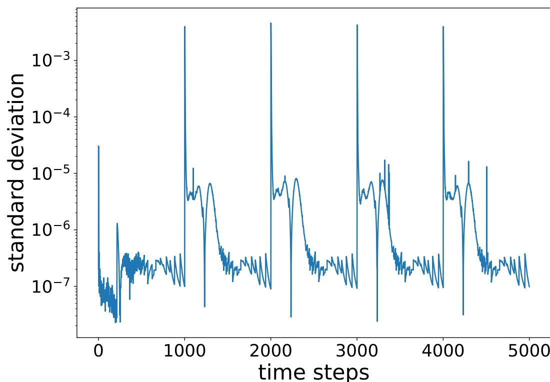


Figure 25: Standard deviation of I_{ion} depending on timestep in the *COURTEMANCHE* model (stimulus every 1000 steps)

To understand how it is possible for standard deviation to rise at the same time as the number of significant digits, we must remember that the number of significant digits is a measurement of the *relative* errors of a floating point value. To give a simple example, if the errors of $round(x)$ compared to the value x (we remind that $round()$ return the nearest floating point representation of a real number) is 0.1 the impact on the number of significant digits would be worse if $x = 1$ than if $x = 10000$. The standard deviation however is based on the absolute error, so as long as the errors from $round(x)$ are the same the exponent of x does not matter. This means that when both the exponent of our values and the standard deviation are rising simultaneously, if the exponent is growing faster then the relative error will decrease instead of increase (or in the case of relative accuracy like the number of significant digits, it will increase instead of decrease). The opposite is true when both are decreasing but the exponent is decreasing faster.

This is exactly what is happening here, during the stable period the standard deviation actually stabilizes but the value of I_{ion} keeps converging toward 0 resulting in a fast drop in exponents, and therefore in lower relative accuracy. This is an inherent problem of using relative measurement when the observed value is close to 0, but this doesn't necessarily mean that the accuracy is getting worse, especially since the absolute error is not increasing. For these reasons, it is necessary to also observe the absolute errors (standard deviation in this case) to observe such cases.

In general, the effect of the precision on the accuracy I_{ion} seems to be similar to the effect on the accuracy of V_m. For example, the gap between double precision and single precision is roughly the same for both V_m and I_{ion}, and we observe the same for half precision. Since V_m and I_{ion} both depend on each other it isn't too surprising that their accuracy seems to be related.

Let us now look at a different model to see if these behaviors can be generalized to the other ionic models. In figure 26, we can see the result of the same experiment on the *FOX* model, which is a smaller model than *COURTEMANCHE*.

First, let us look at the results for the membrane potential V_m. Again, we notice drops in accuracy for all precisions during the active periods, then the accuracy goes back to a stable level during the stable period. However, compared to the same experiment with *COURTEMANCHE*, we also notice that we lose very little accuracy between double and single precision: for a difference of 29 bits of mantissa between double and single precision, we roughly lose 5-6 bits of accuracy at most. However, the accuracy starts to drop fast at half precision compared to single precision: for a difference of 13 bits of mantissa, we lose 12-13 bits of accuracy.
To better understand what is happening here, we added some intermediate levels of virtual precisions with 32, 16, and 12 bits of mantissa. We notice that the results are very similar for a virtual precision of 52 and 32 bits. Both precisions result in the accuracy stabilizing at around 24 during the stable period. However, if we look at all the virtual precisions with a value t of 23 or less, we see that during the stable periods the accuracy of the results is as close to t as possible: for example, single precision stabilizes at 23 bits of accuracy and half-precision at 10 bits. This explains why the accuracy gap is larger between single and half-precision than between double and single-precision. Accuracy appears to reach a peak at 24 bits for double precision and virtual precision 32. But if we use virtual precisions lower than 24, the errors are low enough so that all the representable digits are accurate. Therefore, we can conclude that for V_m, during stable periods, the errors obtained are very small, and we need a virtual precision t high enough (at last $t > 24$) to notice them.

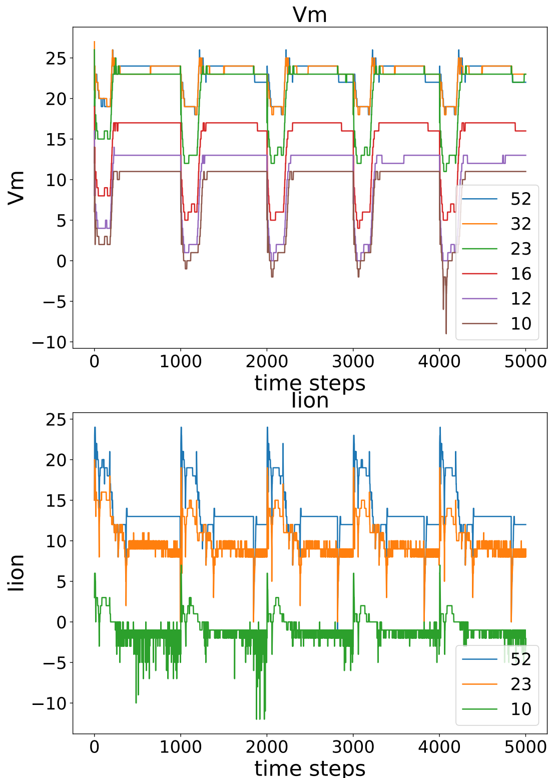


Figure 26: Number of significant digits for V_m and I_{ion} on the *FOX* model depending on the number of iterations (stimuli every 1000 steps)

If we look at the result for I_{ion} in Figure 26, we notice the same behavior as what we observed in Figure 25: the number of significant digits increasing during active periods and decreasing during stable periods. The reason here is identical, absolute error stays the same during the stable period while I_{ion} converges toward zero, making our relative measurement of accuracy worse.

We can conclude that, while the accuracy of some model suffer more when we decrease the precision. The relation between precision and accuracy is very similar between each model.

3.5.2 Precision modification on part of a model

We will now perform precision analysis while only focusing on some parts of the model. To be more precise we will do precision analysis on the lookup tables.

This time what we want to know isn't if we can lower the precision of lookup tables, but how lookup table resolution can affect accuracy. The idea is that if we decrease the resolution of our lookup tables, interpolation happens more often and results are less accurate. The opposite is also true if we increase the resolution of lookup tables, then results are more accurate. Therefore, our goal with the experiments in this section is to better understand the relation between lookup table resolution and accuracy, and to find a way to help the user to choose which resolution to use for his lookup tables.

Verificarlo does not allow us to manipulate the resolution of a lookup table. But we can get a similar result by lowering the precision of the output of the lookup tables to a certain number of bits of mantissa t. For example, if we lower the precision of the output of a lookup table to $t = 20$ then it's equivalent to saying that every output value from the lookup table has only 20 significant digits (in base 2). Therefore if we look at how it affects the accuracy of our results we have an idea of how the accuracy of the lookup table affects the accuracy of V_m and I_{ion}.

Of course, in reality, the values obtained from lookup table interpolation in doesn't always contain the same errors, but this allows us to get a lower bound of the accuracy when the worse approximation interpolated by a lookup table has t significant digits. This helps the user choose the correct resolution by giving him an idea of what is the worse error he can allow.

For this experiment to work, we raised the resolution of our lookup table to avoid interpolation. Since we want to replace the effects of the real interpolations by lowering the precision of the lookup table's output, we want to limit the effect of the reals interpolations as much as possible. However, we cannot completely isolate our experiment from interpolation since this would require the lookup tables to precompute every output for every possible double precision floating point value. This prevents our results from being 100% accurate. This is not too much of a problem, however, as Monte Carlo arithmetic only produces an estimate in the first place.

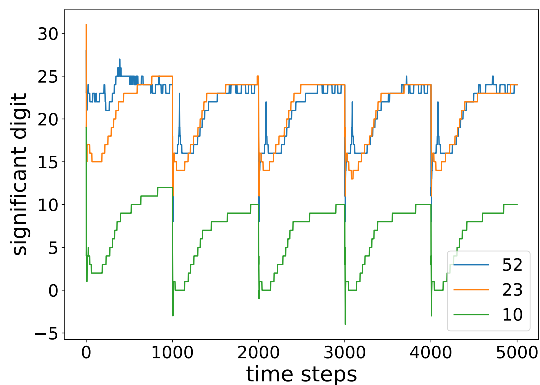


Figure 27: Number of significant digits for V_m in the *COURTEMANCHE* model when instrumenting only the lookup table output, depending on the number of iterations (stimuli every 1000 steps)

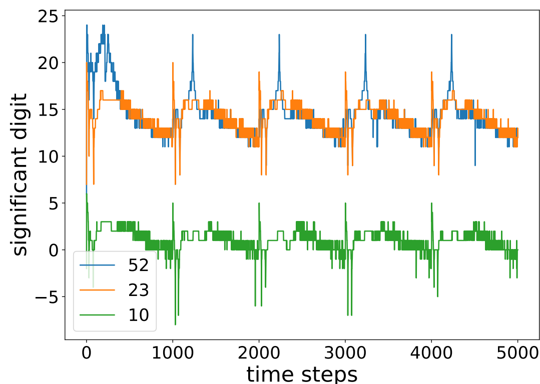


Figure 28: Number of significant digits for I_{ion} on the *COURTEMANCHE* model when Verificarlo only instruments the output of the lookup table, depending on the number of iterations (stimuli every 1000 steps).

In figure 27 we can see the number of significant digits obtained on a simulation with regular stimuli every 1000 steps with the *COURTEMANCHE* model when instrumenting only the lookup table's outputs.

This means that only the output of the lookup tables is randomized by Verificarlo. The results have a similar pattern to the one observed in Figure 22, however, the overall accuracy is higher when Verificarlo only instruments the lookup table. This behavior is expected since a smaller part of the model is randomized, which usually leads to a more accurate result.

Depending on the model, the accuracy observed when instrumenting the output of the lookup table can be pretty different from the ones observed in the fully instrumented version for both V_m and I_{ion}. In figure 28 we can see the result for I_{ion} in the same experiment. Like for V_m, the accuracy is overall better since we only instrument part of the model. We also observe that the result between double and single precision is much closer than in figure 24. This difference means that some operations not covered by the lookup table play an important role in how precision affects accuracy.

Now, we want a way to interpret those data to see how the accuracy of V_m and I_{ion} are related to the resolution of a lookup table.

First, since ionic models use double precision for their floating point operation, the maximum number of significant digits (in base 2) that a variable can have is 52. Since the virtual precision t we set with Verificarlo represents the number of significant digits in the outputs of the lookup tables, this means that when we set it to 52 the output is completely accurate, in other words, it represents the case where there is no interpolation. Therefore, the number of significant digits obtained for double precision in Figures 27 and 28 can only come from the usual rounding and cancelation errors that come with double precision and not from interpolation.

Therefore, we can use the double precision results as a basis to single out the effect of floating-point errors on accuracy. Now we just have to compare the result obtained with lower precision with those obtained with double precision to observe the effect of lookup table interpolation on accuracy. For example, let x_i be the accuracy obtained at time step i when the lookup tables output are in single precision (23 bits of mantissa). Here, x_i represents the accuracy obtained when every interpolation of the lookup table results in output with only 23 significant digits. Now, let y_i be the accuracy obtained for 52 significant digits, we can obtain $z_i = y_i - x_i$ the number of significant digits lost due to interpolation in iteration i.

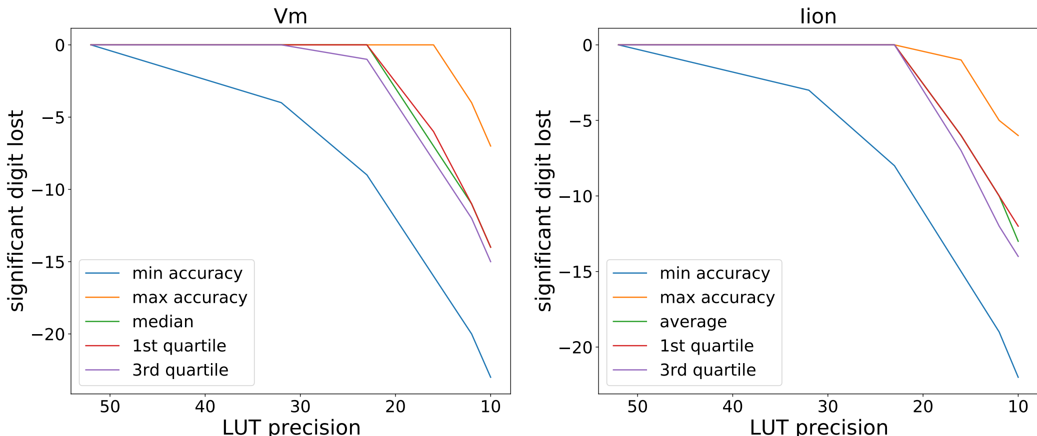


Figure 29: Number of significant digits of accuracy lost depending on the number of significant digits of the lookup tables (LUT) outputs

Using this method on several levels of virtual precisions (LUT precision in Figure 29) to represent different levels of interpolation error, we can approximate the relation between interpolation errors and accuracy. In figure 29 we can see how the accuracy of the output of the lookup tables affects the accuracy of the result. We observe that for the *COURTEMANCHE* model, on average most iterations do not lose too much accuracy for both V_m and I_{ion} when the accuracy of the lookup table results is around the 23 significant digits. In

this graph, when we speak of the median, a median m at LUT precision t means that 50% of the timesteps have a loss of at most m significant digits in accuracy for virtual precision t compared to virtual precision 52. We can observe for V_m that at virtual precision 23, 75% of the timesteps lose at most 2 bits of accuracy compared to the base result (virtual precision 52). This drop becomes much larger for half-precision where the minimum loss is around 7 bits of accuracy and 75% of the timesteps lose more than 14 bits. Max accuracy loss is always far ahead of the 3rd quartile, but this is mostly because we have some timesteps with no errors (52 significant digits) on the base results so the loss on those steps appears pretty severe even if the resulting accuracy is the same as the other steps. Similar results can be observed for I_{ion}.

In this example, we can conclude that limiting the maximum errors from the interpolation of our lookup tables to 23 significant digits minimum lead to good results.

3.5.3 Precision modification depending on time steps

When experimenting with changing the floating point precision in an entire ionic model computation, we noticed that accuracy during active periods (time steps after a stimulus where the values of V_m and I_{ion} are unstable) was not affected in the same way as during stable periods (time steps where the values of V_m and I_{ion} are stable). From our results, we observed that the standard deviation was higher during active periods, as we could see in Figure 25. On the basis of those observations, we asked ourselves if we could get similar results by lowering the precision only during stable periods rather than during the whole computation.

What we hope to observe here is that we can preserve the accuracy during the active periods by keeping them in double precision. The method used for these experiments was to set the precision to double precision for a set number (400) of time steps after each stimulus (the number of time steps is not selected automatically, but chosen by hand based on our previous observation on the duration of the active periods), then switch to a lower precision (single or half) until the next stimulus. For brevity's sake, we will call the experiments where we lower precision on just the stable period: partial lowering.

We can observe this experiment for the *COURTEMANCHE* model in figure 30, here we choose to do a partial lowering to single precision. We compare the result obtained with partial lowering to the results of the base version (everything in double precision), but also to the version when all time steps are in single precision to see how our new results fit between the two.

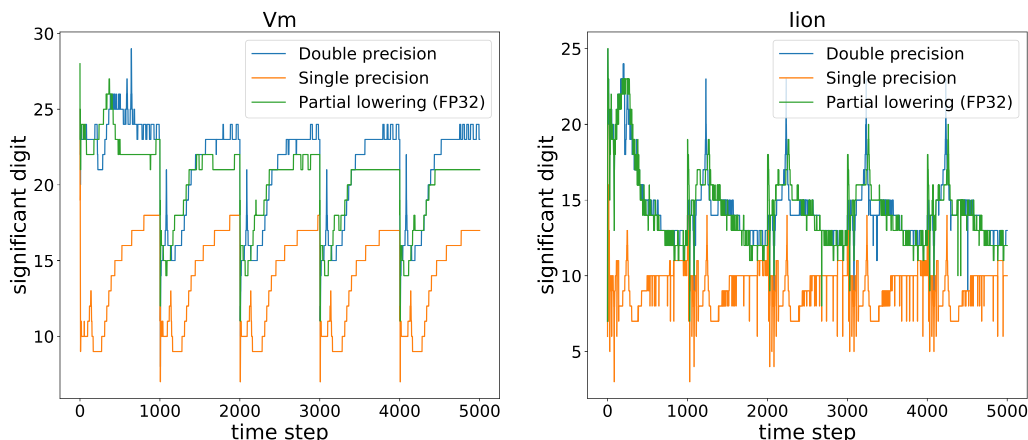


Figure 30: Comparison of significant digit obtained when changing accuracy either the whole model or only during active periods on the *COURTEMANCHE* model. (Stimuli every 1000 steps/Single precision during stable periods)

What we observe here is that the results for partial lowering to single precision are very similar to the

one obtained when the whole computation is in double precision. The accuracy is indeed lower during the stable periods, but the results are very close to the double-precision versions during active periods.

What is interesting is that we obtain better accuracy during the stable periods by using partial lowering than when all time steps are in single precision. This is most likely because we would avoid some errors that would carry over to the stable periods if we switched the active periods to single precision.

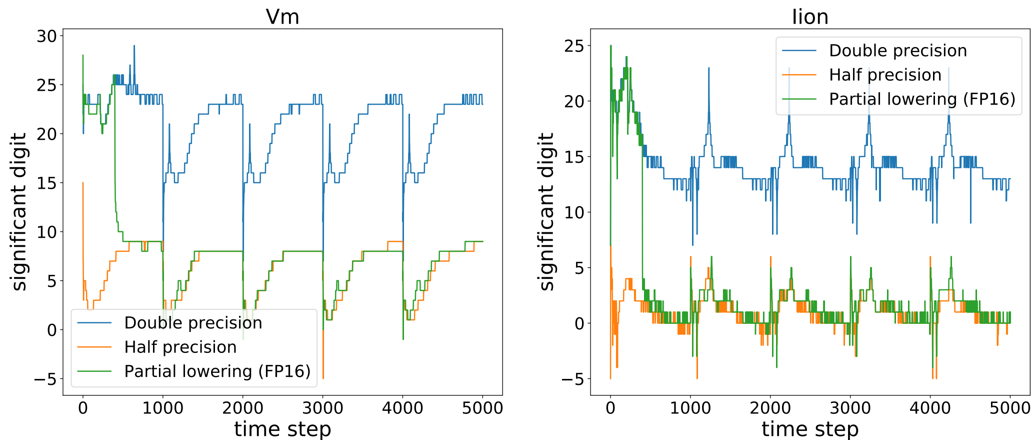


Figure 31: Comparison of the significant digit obtained when changing the accuracy of the entire model or only during active periods on the *COURTEMANCHE* model. (Stimuli every 1000 step/Half precision during stable periods)

In the figure 31 we did partial lowering on the same model, but this time to half precision. What we observe here is that once the precision is lowered to half-precision, accuracy becomes similar for both the partial lowering and the full half-precision version. However this time, in contrast to when we used partial lowering with single precision (in Figure 30), the accuracy does not recover once we switch back to double precision, we are stuck with poor accuracy until the end of the simulation. Therefore, it seems that there is some threshold after which the error is too high for the accuracy to recover.

Sadly, if we look at figure 32, this threshold differs from model to model in such a way that some models do not benefit from partial lowering even in single precision. Here, the accuracy isn't able to recover from the errors even if we just switch to single precision. This means that we cannot generalize partial lowering to all models, since, for the case of the *FOX* model, it would be better to switch the whole computation to single precision as we would be using less memory for a larger amount of time steps. Therefore, if we wanted to know which model would benefit from partial lowering, we would have to evaluate them on a case-by-case basis.

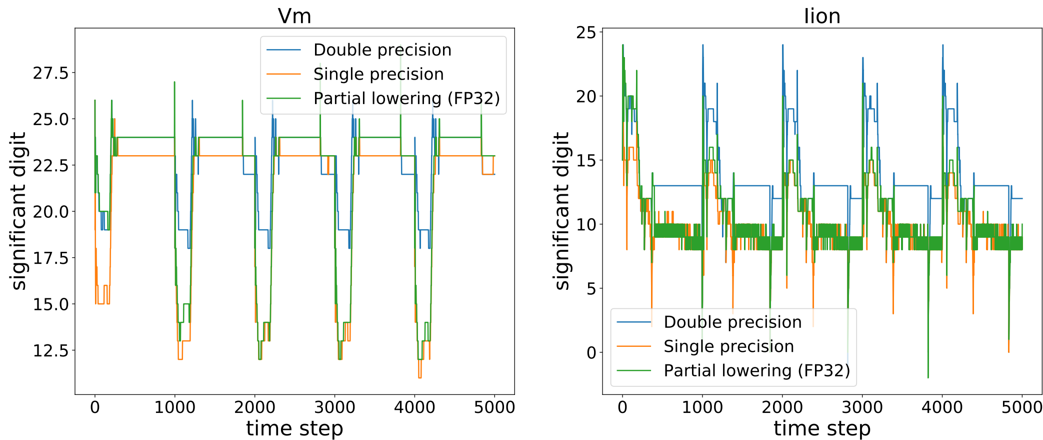


Figure 32: Comparison of the significant digit obtained when changing the accuracy of the entire model or only during active periods on the *FOX* model. (Stimuli every 1000 steps/Single precision during stable periods)

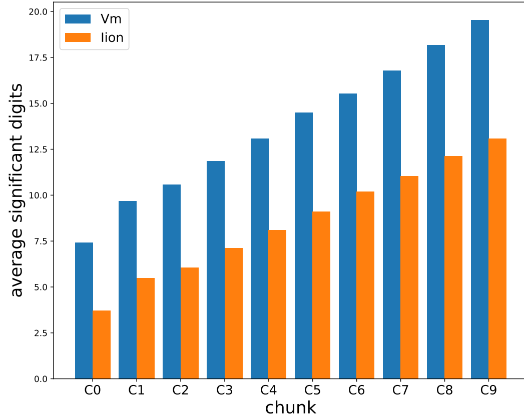
Another question we can ask ourselves is whether switching the precision earlier has a greater impact on the accuracy than doing it later. In other words, if we had to change, for example, 500 of our 5000 time steps to single precision, would it affect accuracy differently if we did it earlier in the simulation?

To answer this question, we will use the following experiments: Let us say that we have a simulation over 5000 timesteps, to see if changing precision sooner or later impacts accuracy differently, we will separate the simulation into 10 chunks of 500 steps. We will then compare 10 versions of the same experiment, in which different chunks were switched to lower precision each time, and look at how the accuracy was affected for the different versions.

One of our intuitions was that maybe lowering precision earlier meant that errors had more time to propagate and worsen than if we lowered precision later. To verify this, we will look at the accuracy of the final timestep to see if it is affected by whether we lowered precision in an early or late chunk. We will also look at the average accuracy among the timesteps (in other words, the mean of accuracy obtained over the 5000 timesteps), to see if some periods of the simulation have a greater impact on accuracy than others. Since we add a stimulus every 1000 timesteps in our experiments, we point out that by cutting the simulation into 10 chunks c_i, $i \in [0, 9]$ where each c_i represents 500 time steps $ts \in [i \times 500, (i + 1) \times 500[$ (i.e., c_3 represents time steps 1500 to 1999) then when i is even, c_i includes an active period.

As we have seen in previous experiments, if we lower precisions too much then accuracy won't be able to recover even if we raise precision back to double-precision, for example when we lower to half-precision on the *COURTEMANCHE* model (See figure 31). In such cases, the results are predictable for the average accuracy, if we lower the precision of an earlier chunk then the impact on average accuracy will be bigger since more timesteps will be stuck with low accuracy. We can observe this behavior in Figure 33 where we can see that lowering the accuracy in the early chunks has a bigger impact on the average accuracy for both V_m and I_{ion}.

However for the accuracy of the last timestep we can see it doesn't matter if we lower precision earlier or later. In figure 34 we can see some fluctuation on the final accuracy due to the randomization of Verificarlo, but no matter on which c_i we lower precision, the final accuracy is around 10 for V_m and 5 for I_{ion}. The only exception is the last two chunks c_8 and c_9 where I_{ion} is more affected (even going down to 0 significant digit). This is most likely due to the accuracy being more affected in the chunk (or the chunk directly after) where we lowered precision.



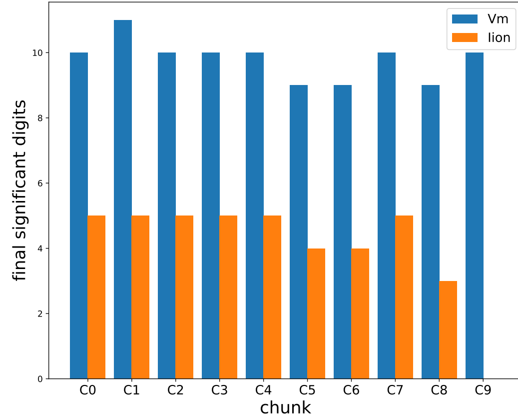
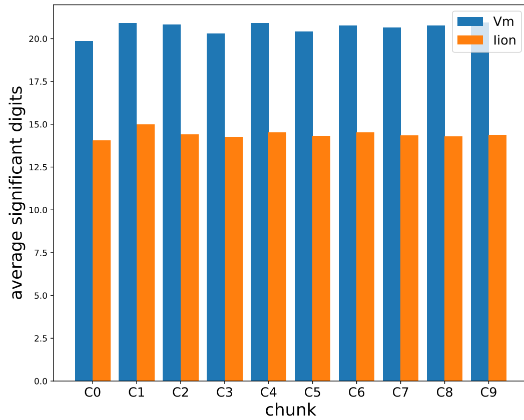


Figure 33: Average accuracy (in significant digits) in function of which c_i we lowered to half-precision in the *COURTEMANCHE* model.

Figure 34: Accuracy of the last time steps (in significant digits) in function of which c_i we lowered to half-precision in the *COURTEMANCHE* model.



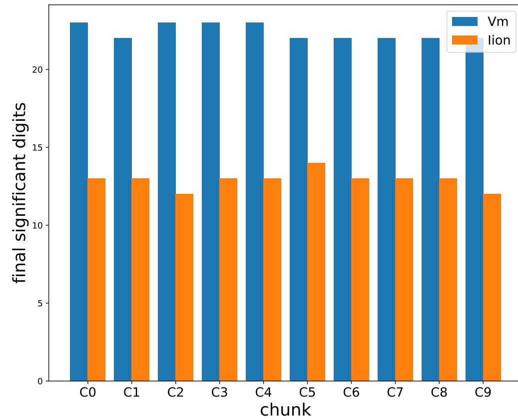


Figure 35: Average accuracy (in significant digits) in function of which c_i we lowered to single precision in the *COURTEMANCHE* model.

Figure 36: Accuracy of the last time steps (in significant digits) in function of which c_i we lowered to single precision in the *COURTEMANCHE* model.

Let us now repeat the same experiment by lowering the precision to a threshold from which we know the accuracy is able to recover. For example, we lower to single precision on *COURTEMANCHE* (see Figure 30. Then as we can see in figure 35 and 36, neither the average nor final accuracy seems to be affected differently by on which c_i we lower precision. We can conclude that in such cases lowering precision earlier doesn't seem to propagate more errors, and that it doesn't matter much for average accuracy if we lower precision on a chunk with or without an active period.

3.5.4 Perspectives for optimizations

Our primary goal for running those precision analyses was to see if it was possible to do some precision-based optimization on Limpet. Therefore, in this section, we will quickly discuss some of the optimizations that we think are possible on the basis of our results.

Our initial belief was that probably no models could be switched to single-precision without a significant loss of accuracy. However, if we look at, for example, the *FOX* model (see Figure 26) we only have a difference of around 4 bits of accuracy between double and single precision. Therefore, it is perfectly possible that some models can, in fact, be switched to single precision entirely.

Following the same reasoning, partial lowering of precision to target only the active periods works well on some models and could be used as a compromise between double and single precision on models where lowering to single precision impacts accuracy too much.

Of course, whether some models would benefit or not from those optimizations depends a lot on the accuracy needed by the users. If we look at the result in Figure 37, we notice that no matter the precision used we get very similar results. In other words, even if the results are accurate up to a very limited number of significant digits, they still follow the same pattern no matter the precision used.
Therefore, if the users do not need a lot of significant digits, but just something accurate enough to correctly represent cardiac activity it could be possible for some models to lower precision to half-precision.

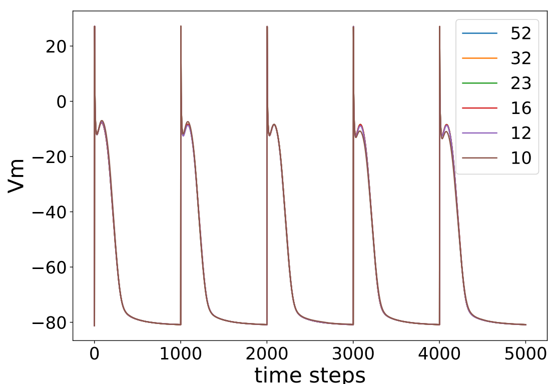


Figure 37: Membrane potential (V_m) depending on timestep in the *COURTEMANCHE* model (stimulus every 1000 steps) for different precisions.

Finally, many models barely (or even did not) reach above 23 bits of significant digits even with double precision. Therefore, it is possible that even on models where we have to keep the computation to double precision to preserve accuracy, we would not lose a lot of accuracy by storing our values in single precision.

However, while this would reduce memory consumption and potentially lead to better performance, this could also backfire and reduces performance, since casting between floating-point precision is pretty costly.

We still need to perform more precision analysis experiments to find more potential optimizations. In particular, we need to find how we could do pre-calibration runs to evaluate models quickly (from an execution time point of view) and automatically to find which precision optimizations are possible.

4 Remaining works

We could not finish producing any of the heterogeneous versions we initially planned since the GPU version they depended on was still in development by the end of this internship; some of the functionalities we needed to run our heterogeneous versions efficiently are not available in the current GPU version.

However, we still developed part of the implementation. And since we already did a similar implementation during a previous internship (see [2]), we already know how to implement those versions in Limpet. Therefore, in this section, we address the requirements that have been met and the requirements that still need to be done to implement those versions.

To achieve both versions, we need to satisfy the following requirements :

1. Make a multi-tasks version of the GPU version to run it on multiple GPU
2. Enable a way to run both the vectorized and GPU version without re-compiling.
3. Add a way to select on which GPU the computation will run, allowing StarPU to transparently handle the transfer from the host to the desired GPU.
4. Add asynchronous data transfer for the GPU version to overlap data transfer with computation.
5. Implement dynamic load balancing between the different devices.

The item 1 has already been completed. We needed to adapt the code to split the work among multiple tasks and handle data transfer between them. These modifications were already completed during the development of the multi-tasks CPU version.

The item 2 is partially completed. The vectorized version and the GPU version are only generated if we pass a specific option during compilation. However, the options to generate the vectorized version and the GPU version were exclusive, so we had to modify the code generation to be able to choose between them during execution. However, since we did not finish item 3 the way it currently works is not optimal as we have to use a CPU to handle data transfer to the GPU.

The items 3 and 4 have not been completed. In the current GPU version, all the code related to GPU is hidden between functions generated automatically by MLIR. Therefore, we have no way to transfer the parameters needed for the related CUDA functions to handle asynchronous data transfer or to handle which GPU runs which part of the computation without modifying how the MLIR code is generated. To do this, we need to work with the INRIA CAMUS team to add the necessary modifications.

The item 5 is not yet implemented; however, we already implemented dynamic load balancing on a similar project during a previous internship. The implementation is similar to what we described earlier in figure 6, we measure the execution of each task at the end of each iteration to decide the work distribution of the next iteration. We will now describe the algorithm[9] we used to find the number of nodes that each device needs in the next iteration to have a similar execution time.

Let n be the number of devices and $i \in [0, n[$ be the index assigned to each device. Let $x_i \in [0, 1]$ be the ratio of nodes assigned to the device i for the current iteration. With $\sum_{i=0}^{n} x_i = 1$. Let t_i be the time it takes the device i to run its task during the current iteration. Let c be the total number of nodes of the simulation. We can then define m_i : the average execution time per node for the device i as $m_i = t_i/(x_i \times c)$.

Let us introduce two new variables y_i, the ratio of nodes assigned to device i at the next iteration (with $y_i \in [0, 1]$) and T the time such that for each device we have $\forall i, m_i \times y_i \times c = T$ (or $\sum_{i=0}^{n} m_i \times y_i \times c = T$). We want to find each y_i.

We have $y_i = \frac{T}{m_i \times c}$ with $\sum_{i=0}^{c} y_i = 0$ therefore we have $\sum_{i=0}^{n} \frac{T}{m_i \times c} = 1$ which gives us $T = \frac{c}{\sum_{i=0}^{n} 1/m_i}$. By replacing T, we can obtain :

$$y_i = \frac{\frac{c}{\sum_{i=0}^{n} 1/m_i}}{m_i \times c} = \frac{c}{m_i \times c \times \sum_{i=0}^{n} \frac{1}{m_i}}$$

which we can convert to :

$$y_i = \frac{1}{m_i \sum_{i=0}^{n} \frac{1}{m_i}}.$$

Now we have a formula that allows us to find each y_i. Therefore, by using the task profiling functionality of StarPU, we can recover the execution time of our tasks for a given iteration to find the node distribution for the next iteration. This allows us to find a load balance for each timestep based on the performances obtained during the previous timestep.

However, while we can do this every iteration, adjusting the load balance requires us to modify all the StarPU data filters used for our computation and transfer a lot of data between our devices. This is very expensive, so we cannot afford to do it for each time step. To be efficient, we need to choose a threshold (say,

for example, 1.1) s. If there is not at last one value t_i for which $t_i > t_min * s$ (where t_min is the smallest t_i), then we do not adjust the load balance for this iteration.

5 Conclusion

In conclusion, I completed my end-of-study internship at the INRIA Bordeaux Research Institute. During this 6-month research internship, I was tasked with adapting a cardiac electrophysiology simulation code to work on heterogeneous architectures. I was also tasked with performing precision analysis on the same code to explore different precision-related optimizations. To accomplish those tasks, I could exploit the skills I acquired during my master studies in high-performance computing at the University of Bordeaux.

I could finish adding all the tools we needed to the cardiac simulation software and making them work together. The main challenge was that some of those tools were still in development and not always compatible; I had to keep up with updates for the MLIR-generated versions produced by the INRIA CAMUS team. I learned a lot about working on an active project through these experiences. I learned about some new tools like the LLVM compilation platform and Verificarlo, but also learned more about tools I was already familiar with, like StarPU and CMake.

For the task of handling heterogeneity, I produced a task-based CPU version using StarPU. This was to lay the ground for future heterogeneous versions, as they will need a way to handle tasks to distribute work to different devices. However, I was unable to fully produce any heterogeneous version during the internship, since the GPU version we needed was still too early in development by the end of the internship. For this reason, we chose to focus more on the precision analysis part of the internship.

For precision analysis, various experiments were conducted to understand how floating-point precision affected the accuracy of cardiac ionic models. I also conducted similar experiments to see how accuracy is affected by the errors introduced by lookup table interpolation. We believe that some memory optimizations are already possible following these results. We could present our first results at the Microcard workshop in July.

Since I will continue working on a subject similar to this internship during my thesis for the same project, this will give me a head start for my future works. We plan to finish the heterogeneous versions we started and extend them for multi-region simulation. For precision analysis, we plan to continue the experiments and test some of the possible optimizations to measure their effects on energy consumption.

I gained many skills through this internship, mainly because I had the opportunity to work with multiple people from different fields. But also because of the diverse scientific presentations that I was brought to listen to. I learned a lot about code generation, compilation tools such as MLIR, how to analyze and handle errors produced by floating point operations, load balancing, and also about cardiac electrophysiology, which is a subject that I hope to have a decent understanding of before the end of my thesis.

References

[1] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California, Dec 2003.

[2] Vincent Alba. Génération de codes parallèles pour simulation cardiaque. `https://fr.overleaf.com/read/ccjmnwrhgsbr`, 2021. M1 internship report, Accessed: 2022–07-19.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] Terry Cojean. *Programmation des architectures hétérogènes à l'aide de tâches divisibles ou modulables.* Theses, Université de Bordeaux, March 2018.

[5] Autumn A. Cuellar, Catherine M. Lloyd, Poul F. Nielsen, David P. Bullivant, David P. Nickerson, and Peter J. Hunter. An overview of cellml 1.1, a biological model description language. *SIMULATION*, 79(12):740–747, 2003.

[6] Christophe Denis, Pablo De Oliveira Castro, and Eric Petit. Verificarlo: Checking floating point accuracy through monte carlo arithmetic. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 55–62, 2016.

[7] F. H Fenton and E. M. Cherry. Models of cardiac cell. *Scholarpedia*, 3(8):1868, 2008. revision #91508.

[8] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991.

[9] Pierre Huchant. *Static Analysis and Dynamic Adaptation of Parallelism.* Theses, Université de Bordeaux, March 2019.

[10] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[11] Vincent Loechner. Microcard deliverable 6.1. 2022.

[12] OpenMP. Openmp-api-specification-5-2, 2021.

[13] Yohan Chatelain Pablo Oliveira, Eric Petit. Verificarlo - 29th vi-hps – reim. `https://www.vi-hps.org/cms/upload/material/tw29/Verificarlo.pdf`, 2018. Accessed: 2022–07-19.

[14] Douglas Stott Parker and David Langley. Monte carlo arithmetic: exploiting randomness in floating-point arithmetic. 1997.

[15] Gernot Plank, Axel Loewe, Aurel Neic, Christoph Augustin, Yung-Lin Huang, Matthias A.F. Gsell, Elias Karabelas, Mark Nothstein, Anton J. Prassl, Jorge Sánchez, Gunnar Seemann, and Edward J. Vigmond. The opencarp simulation environment for cardiac electrophysiology. *Computer Methods and Programs in Biomedicine*, 208:106223, 2021.

[16] Mark Potse. Microcard: Numerical modeling of cardiac electrophysiology at the cellular scale, March 2022.

[17] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Trans. Program. Lang. Syst.*, 41(1), dec 2018.

[18] Haluk Rahmi Topcuoglu, Salim Hariri, and Minyou Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Syst.*, 13:260–274, 2002.