Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology

Arun Thangamani, Tiago Trevisan, Vincent Loechner, Stephane Genaud, Bérenger Bramas

Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology

Arun Thangamani*
Strasbourg Univ. and Inria, France
arun.thangamani@inria.fr

Tiago Trevisan Jost*
Strasbourg Univ. and Inria, France
tiago.trevisan-jost@inria.fr

Vincent Loechner
Strasbourg Univ. and Inria, France
loechner@unistra.fr

Stéphane Genaud
Strasbourg Univ. and Inria, France
genaud@unistra.fr

Bérenger Bramas
Strasbourg Univ. and Inria, France
berenger.bramas@inria.fr

Abstract

The study of numerical models for the human body has become a major focus of the research community in biology and medicine. For instance, numerical ionic models of a complex organ, such as the heart, must be able to represent individual cells and their interconnections through ionic channels, forming a system with billions of cells, and requiring efficient code to handle such a large system. The modeling of the electrical system of the heart combines a compute-intensive kernel that calculates the intensity of current flowing through cell membranes, and feeds a linear solver for computing the electrical potential of each cell.

Considering this context, we propose *limpetMLIR*, a code generator and compiler transformer to accelerate the kernel phase of ionic models and bridge the gap between compiler technology and electrophysiology simulation. *LimpetMLIR* makes use of the MLIR infrastructure, its dialects, and transformations to drive forward the study of ionic models, and accelerate the execution of multi-cell systems. Experiments conducted in 43 ionic models show that our *limpetMLIR* based code generation greatly outperforms current state-of-the-art simulation systems by an average of 2.9×, reaching peak speedups of more than 15× in some cases. To our knowledge, this is the first work that deeply connects an optimizing compiler infrastructure to electrophysiology models of the human body, showing the potential benefits of using compiler technology in the simulation of human cell interactions.

CCS Concepts: • **General and reference** → *Performance*; • **Computer systems organization** → **Parallel architectures**; • **Software and its engineering** → **Compilers**.

*Both authors contributed equally to the paper

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02…$15.00
https://doi.org/10.1145/3579990.3580008

Keywords: Code generation and optimization, code transformation, domain-specific languages, vectorization.

ACM Reference Format:
Arun Thangamani, Tiago Trevisan Jost, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. 2023. Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579990.3580008

1 Introduction

The fields of biology, chemistry, and physics do not only rely on experimental, laboratory-related researchers, but also on analyzing and experimenting with computer models that require significant computational performance. Many computer-experimented works are based on mathematical concepts to design and model the physical behavior of the matter of interest. This modeling often involves relying on a language abstraction, typically a domain-specific language (DSL), to provide the necessary instrument to describe the computation. As an example, this work is addressing the simulation of cardiac tissues in the domain of electrophysiology, which is fundamentally modeled by ordinary differential equations (ODEs). Specifically, the openCARP [26] framework aims to foster the needs of large parts of this community for reproducible research with such simulations. The challenge for representing ODEs boils down to how should one translate this level of language abstraction into efficient code, so that compilers can optimize it for fast execution. However, openCARP provides a simplified code generation process that hinders compilers for optimization opportunities.

Compiler technology has become paramount to harness performance of applications. The rise of Multi-Level Intermediate Representation (MLIR) [15] as a new compiler infrastructure has enabled the expansion of code transformations to higher abstraction levels. Though MLIR has already been applied in a number of fields, from linear algebra [4, 32] to quantum technology [20], it has not yet been leveraged as a viable replacement to traditional code generators in

frameworks that model electrophysiology. In electrophysiology, ionic models describe the interactions between cells composing tissues, as for example the human heart.

The purpose of this work is precisely to bring the MLIR compiler technology into the field of electrophysiology to improve the code generation from a DSL. After the construction of Abstract Syntax Trees (ASTs), *limpetMLIR* code generator makes use of conventional MLIR dialects to produce a binary code that is far more optimized than the one produced by a traditional compiler from a C/C++ translation. In particular, our optimizations fully take advantage of the Single Instruction Multiple Data (SIMD) execution capability (vectorization) supported by modern CPUs.

The main contributions of this paper are:

1. a compiler frontend for a seamless integration between ionic models and MLIR, capable of harnessing the power of compiler technology into electrophysiology,
2. a set of code transformations and optimizations to further improve the execution of ionic models,
3. a full experiment platform for ODE models that takes advantage of MLIR and demonstrates the benefit of integrating novel compiler technology and differential equations.

The paper is organized as follows. Section 2 first presents the motivation and the software pieces making up openCARP, and details the current compilation flow, from the model description using a DSL to the code generation. Section 3 details our code generation using MLIR and optimizing transformations. A thorough evaluation of our proposal is done in section 4 on 43 models included in openCARP. A discussion is provided in section 5, and related work is covered in section 6. Finally, section 7 concludes this paper.

2 Compilation Flow in openCARP

2.1 Motivation

Computational modeling and simulation of cardiac electrophysiology have gained importance in recent years, playing a major role in cardiac research. Myokit [7] and openCARP [26], although having different objectives, are the two most commonly used cardiac electrophysiology simulators based on ionic model descriptions in academia and research. Ionic models are written using domain-specific (markup) languages such as EasyML [1], CellML [16], and SBML [11], and provide biomedical researchers a high-level abstraction to describe electrical currents (as ions) flowing through cell membranes. These ionic models compute the right-hand side of the ODEs solved in the simulation, that corresponds to the right-hand side of a matrix that is fed to a high-performance solver (outside the scope of this paper).

One should notice that such simulations on multiple cells (a human heart contains about 2 billion muscle cells) could take very long to complete on computationally expensive cardiac models, raising concerns in terms of the viability of their execution. This motivates new research to tackle how code generation is handled by the aforementioned frameworks. And ultimately, harnessing compiler technology for optimizing the code generation of these models will make it possible to simulate much larger or more precise systems. This will contribute to cardiac research by providing a better understanding of the heart electrical functioning, and potential malfunction causing arrhythmia.

2.2 Representation of Ionic Models

Ionic models essentially represent the chemical interactions that occur between the muscular cells of cardiac tissue. They compute the current flows crossing cell membranes from a given state of the cells (the cells *state variables*). Being a mathematical discretized representation of a dynamic system, an ionic model is a combination of arithmetic, mathematical and even control-flow operations.

The choice of a language for describing an ionic model is challenging. A high-level abstraction accelerates the modeling process, but may not provide the flexibility one wants for expressing models. As a counterpart, a low-level abstraction provides the needed flexibility and expressiveness, while providing a more cumbersome modeling strategy. On the other hand, a variety of scientific fields rely on a discretized mathematical model for simulating a complex system solved by ODEs, each imposing different restrictions to code generation and to specific data structures and computations.

The description of an ionic model should ideally combine expressivity and flexibility, along with allowing fast description of a model. In the previously mentioned openCARP framework, models are described using EasyML, a convenient and robust markup language that is widely used by specialists as a DSL. From a compiler perspective, EasyML expressions follow the ideas behind the static single assignments (SSA) [8], a form of representation that is common within compiler representations. SSA values can only be assigned once, a property that eases the use of many known optimizations by the compiler. For all these reasons, we borrow the syntax and semantics of EasyML in order to enable compatibility between our proposed solution and a state-of-the-art format for describing ionic models.

The adoption of EasyML has not only been considered as a result of its use by modeling specialists. Indeed, the realization that EasyML serves as a common point to other language references is one of the main advantages and motivation for its use. The language delivers characteristics of both a frontend and intermediate representation: one can not only design its own model directly using EasyML, but the language can serve as intermediate representation for other description languages. Figure 1 illustrates how EasyML serves as an intermediate representation for different formats: CellML [16], SBML [11], and MMT [7] formats can be converted to EasyML through semi-automatic scripts available in the openCARP and Myokit.

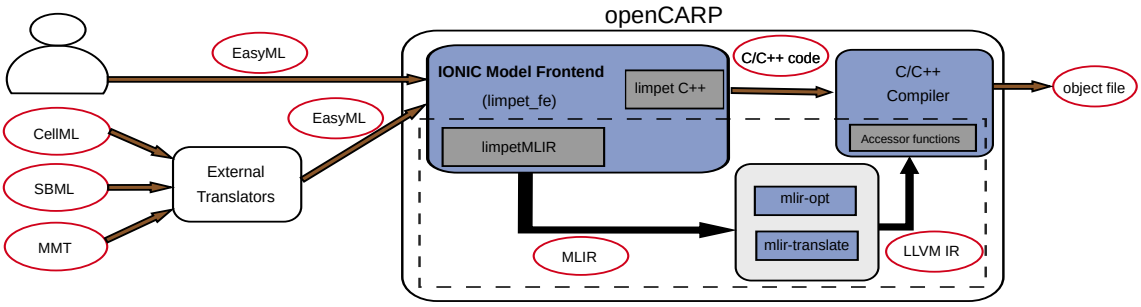

Figure 1. An overview of the lifting strategy used to enable ionic models in MLIR. The left-hand side show EasyML working as a frontend language and as an intermediate representation for different formats. The right-hand side describes the proposed compilation flow that takes EasyML as input language.

The language was primarily designed to allow for fast implementation of differential equations. Variable assignments, *if* statements and the precedence of arithmetic operations follow those of C/C++ languages. While focusing on the expressivity of computational models in electrophysiology, EasyML is not seen as comparable to traditional programming languages. Its main characteristics are:

1. The language is not Turing complete in that it cannot express loops.
2. Variables `diff_` and `_init` are recognized as part of the language and represent the differential equation for a variable and its initial value, respectively.
3. *Markup* statements can be used to change the variables handled by the code generation. Examples of markup keywords are `external`, `lookup`, `method(integration_name)`, etc. These keywords allow the user to control different aspects of the generated code, such as, variables that are external to the cell state variables, creation of lookup tables for fast linear interpolation, or which discrete integration method (among a collection) should be used for a specific equation.

These language structures and features make EasyML compatible with numerous models for cardiac cell simulation. In the following section, we describe the code generation we have developed for the betterment of cardiac-cell simulation.

3 Optimized Code Generation

3.1 Overview

Our code generation for cardiac models, called *limpetMLIR* in the following, is conceptualized to leverage the inherent parallelism found within cells and improve the efficiency of the *compute* stage. The simulation flow in openCARP is mainly divided into two stages: (1) In the *compute* stage, cells compute ionic currents written in the EasyML format. Each cell can have access to a shared read-only state that store common information among them, along with a private read/write state which is updated at the end of the stage. (2) The *solver* stage starts once all cells have synchronized after the *compute* stage completion. In this stage, computed values are passed to a linear solver for the actual ODEs solving.

While the simulation flow is a two-stage execution, we take advantage of the lack of synchronization among cells in the compute stage to parallelize their execution through SIMD operations. In other words, we make use of MLIR dialects and operations to fully vectorize the computation of cells state variables. We have adapted many features from the openCARP framework and C/C++ languages to seamlessly work in MLIR. Function pointers from C, integration methods, and lookup table (LUT) acceleration are automatically ported to MLIR for vector support. While our overall contribution is to connect a simulating infrastructure for the study of the human heart to compiler technology, our work shows a proof-of-concept MLIR code generation that does not require an additional dialect proposal. Though this could be perceived as lacking since many previous work on MLIR [9, 28, 29] have proposed new dialects, we have found no justification for the addition of a new language. Indeed, MLIR includes all necessary dialects and operations required to optimize the execution of ionic models. An MLIR dialect extension would conflict with the proposal of an easy-to-use and upstream-compatible code generator.

Figure 1 shows the complete compilation flow:

1. EasyML works as a frontend language (or high-level intermediate representation) for different formats (CellML, SBML, MMT). Semi-automatic tools can be used for language conversion.
2. Inside openCARP, a frontend called *limpet*, written in Python, translates EasyML into an abstract syntax tree (AST) representation for syntax and semantic analysis in the implemented ionic model description. This process is a common component between our code generator *limpetMLIR* and the original openCARP *limpetC++* that generates C/C++ code.

```
1  Vm; .external(); .nodal(); .lookup(-100,100,0.05);
2  Iion; .external(); .nodal();
3  group{ u1; u2; u3; }.nodal();
4  group{ Cm = 200; beta = 1; xi = 3; }.param();
5
6  u1_init = 0; u2_init = 0; u3_init = 0; Vm_init = 0;
7  diff_u3 = 0;
8  diff_u2 = -(u1+u3-Vm)*cube(u2);
9  diff_u1 = square(u1+u3-Vm)*square(u2)+0.5*(u1+u3-Vm);
10 u1;.method(rk2);
11
12 Iion = (-(Cm/2.)*(u1+u3-Vm)*square(u2)*(Vm-u3)+beta);
```

Listing 1. Modified version of Pathmanathan [25] ionic model written in EasyML

3. From the AST, *limpetMLIR* makes use of `vector`, `controlflow`, `arith`, `math`, `memref`, and `openmp` dialects to implement the majority of EasyML features.
4. Finally, *limpetMLIR* includes two additional code transformations for further performance improvements: a vectorization-based LUT computation, and a data layout optimization to avoid the effects of non-consecutive data storage of variables within ionic models.

In the following paragraphs, we walk the reader through the proposed compilation flow and detail aspects of the code generation and proposed optimizations.

3.2 Preprocessor

The description of an ionic model generates AST nodes with distinct properties: some can only be computed at runtime, while others generate a set of values with constant-qualified behavior. Therefore, it becomes necessary to provide a preprocessor capable of analyzing nodes in order to determine which values can be calculated at compile time. We have included in *limpetMLIR* a preprocessor akin to C/C++ preprocessors, capable of handling a myriad of compile-time operations, such as arithmetic, mathematical, and condition. We keep track of constant-qualified values and their utilization in order to propagate compile time constant values. The preprocessor stage works as part of the code generator phase that is described below. Once operations with constant values are found, the preprocessor kicks in and compute their propagation values and computations.

3.3 MLIR Code Generation

3.3.1 Example. In order to give an overview of the code generation process, we use a slightly modified version of the Pathmanathan [25] model written in EasyML, shown in Listing 1. The model was artificially changed to include two features: (1) LUT for the `Vm` variable where values of `Vm` are known to vary between -100 and 100, and we chose a step of 0.05; and (2) the use of `rk2` integration method for `u1`. We have also simplified the equations for `u1` to `u3` in order to limit the example size. One may notice that this modified model does not correspond to a real case scenario, and is only an illustrative example for the presented section.

```
1  #pragma omp parallel for schedule(static)
2  for (int __i=start; __i<end; __i++) {
3    Pathmanathan_state *sv = sv_base+__i;
4    //Initialize the ext vars to current values
5    Iion = Iion_ext[__i], Vm = Vm_ext[__i];
6    //Compute lookup tables
7    LUT_data_t Vm_row[NROWS_Vm];
8    LUT_interpRow(&IF->tables[Vm_TAB], Vm, __i, Vm_row);
9    //Compute storevars and external modvars
10   Iion = (((((-(p->Cm/2.))*((sv->u1+sv->u3)-(Vm)))
11     *(square(sv->u2)))*(Vm-sv->u3))+p->beta);
12   //Complete Forward Euler Update
13   diff_u2 = ((-((sv->u1+sv->u3)-Vm))*cube(sv->u2));
14   u2_new = sv->u2+diff_u2*dt;
15   u3_new = sv->u3+diff_u3*dt;
16   //Complete RK2 Update
17   diff_u1 = (((square(((sv->u1+sv->u3)-(Vm))))*
18     (square(sv->u2))) +(0.5*(sv->u1+sv->u3-Vm)));
19   double u1_new;
20   {
21     t = t + dt/2;
22     sv_intermed_u1 = sv->u1+dt/2*diff_u1;
23     diff_u1 = (square(sv_intermed_u1+sv->u3-Vm)*
24      square(sv->u2))+(0.5*(sv_intermed_u1+sv->u3-Vm));
25     u1_new = sv->u1+dt*diff_u1;
26   }
27   //Finish the update
28   Iion = Iion, sv->u1 = u1_new;
29   sv->u2 = u2_new, sv->u3 = u3_new;
30   //Save all external vars
31   Iion_ext[__i] = Iion, Vm_ext[__i] = Vm;
32 }
```

Listing 2. Baseline version generated code snippet of the modified Pathmanathan [25] model from Listing 1

Lines 1 and 2 declare variables `Vm` and `Iion` as external, which means they represent voltage and current that flows in and out of a cell. Lines 3 and 4 define groups of variables, that are specific to each cell (line 3), and common values among cells (line 4). Value initialization is done in line 6 through the `_init` suffix, and equations from `u1` to `u3` are defined in lines 7-9. Line 10 tells the code generator to use the 2-point Runge-Kutta (*rk2*) integration method for the derivative of `u1`, while the default *Forward Euler (fe)* method is used for the remaining variables. Finally, the current flow (`Iion`) follows the equation from line 12, while `Vm` is not updated.

3.3.2 Overview of the code generation. The original openCARP framework uses a simple code generator written in python to translate EasyML into different C/C++ functions that initialize parameters, state variables, lookup tables, and that compute the current flows across the cell membranes. Considering Amdahl's law [2] and the impact of each function in execution time, we focus on optimizing the `compute` function of models, where a considerable amount of execution time is spent.

Listing 2 depicts a snippet of the compute function generated by openCARP. It shows a straightforward translation from the EasyML code of Listing 1 to C/C++.

One may notice that there is no loop-carried dependency between iterations of the loop. Indeed, each iteration corresponds to the execution of the model from a different cell. In line 3, we retrieve the state of each cell that will be used to compute its new state. The absence of synchronization and

```
1  %3 = call load_single_value_to_vec(%arg2, %c8_i32)
2     : (memref<1xi8>, i32) -> vector<8xf64>
3  scf.for %arg13 = %0 to %1 step %c8 {
4   %4 = arith.index_cast %arg13 : index to i32
5   %5 = arith.muli %arg6, %4 : i32
6   %6 = arith.index_cast %5 : i32 to index
7   %7 = memref.cast %arg5 : memref<8xi8> to memref<?xi8>
8   %8 = memref.view %7[%6][] : memref<?xi8> to memref<1
        xi8>
9   %9 = arith.muli %arg8, %4 : i32
10  %10 = arith.index_cast %9 : i32 to index
11  %11 = memref.cast %arg7 :memref<8xi8> to memref<?xi8>
12  %12 = memref.view %11[%10][] : memref<?xi8> to memref
        <1xi8>
13  %13 = arith.muli %arg10, %4 : i32
14  %14 = arith.index_cast %13 : i32 to index
15  %15 = memref.cast %arg9 :memref<8xi8> to memref<?xi8>
16  %16 = memref.view %15[%14][] : memref<?xi8> to memref
        <1xi8>
17  %c3_i32 = arith.constant 3 : i32
18  %17 = call load_struct_to_vec(%8, %c8_i32, %c3_i32)
19     : (memref<1xi8>, i32, i32) -> vector<8xf64>
20  %c0_i32_1 = arith.constant 0 : i32
21  call LUT_interpRow_n_elements_vec(%arg11, %17,
22     %4, %arg12, %c0_i32_1) : (memref<1xi8>,
23     vector<8xf64>, i32, memref<1xi8>, i32) -> ()
24  %cst = arith.constant dense<2.0e+0> : vector<8xf64>
25  %18 = arith.divf %2, %cst : vector<8xf64>
26  %19 = arith.negf %18 : vector<8xf64>
```

Listing 3. MLIR code snippet generated by *limpetMLIR* for the modified Pathmanathan [25] model from Listing 1.

communication among cells inside the loop, shown by the simplicity of the `omp` directive in line 1, allows us to leverage SIMD execution: that is, each cell can be thought of as representing one element of a vector operand.

Therefore, we provide a full *SIMDfication* of the `for` loop through MLIR in order to improve performance over the original generated code. While the latter tries to rely on the compiler for vector-based optimization opportunities, we make use of MLIR to explore vectorization not as an optimization feature, but as an intrinsic feature. More specifically, `vector` and `openmp` dialects are used to vectorize and parallelize the code, along with `control-flow`, `arith`, `math`, and `memref` dialects. Through these dialects, we can express vectors of ionic models that: access external and state variables of cells, support a variety of integration methods for differential equations, LUTs to accelerate computation, and communicate data among cells through function pointers. Listing 3 shows part of the main `scf.for` loop that iterates over cells numbered between `%0` and `%1`, and that increments the counter by the vector length, user-defined as eight (`%c8`). Every generated operation uses vector types of size *eight*, leading each loop iteration to execute eight cells in parallel.

Integration methods. An important aspect of an ordinary differential equation lies in the selection of the appropriate method for the temporal discretization of an approximate solution. Depending on the desired accuracy, users can use different methods to compute a value of the next time step. We have implemented directly in MLIR the following integration methods.

Forward euler (fe) [5] is a fast and explicit first-order method for solving ODEs. It is the default method used by openCARP when none is specified by the user. In Listing 2 (Lines 14 and 15), variables `u2` and `u3` use *fe* as their integration method.

Runge-Kutta with 2 steps (rk2) [10] is an explicit second-order method. It provides better accuracy than *fe*, with twice as much computations (two calls to the `f` function). Variable `u1` in Listing 1 (line 10), and Listing 2 (lines 17-26) show an example use of *rk2*.

Runge-Kutta with 4 steps (rk4) [10] is an explicit fourth-order method that provides more accuracy than *rk2* with more than twice computations, i.e. 4 times slower than *fe*.

Rush-larsen [27] is one of the most popular first-order methods for discretizing ODEs in dynamic models of cardiac electrophysiology [19]. Easy to implement and more stable than *fe* and its variants, it is the preferred method for simulating *gates*, which represent the movement of proteins forming the ion channel in response to the membrane potential.

Sundnes method [30] is an extension of the Rush-larsen in a second-order scheme, which is proven to be more efficient than its predecessor over stiff problems.

Markov_be is a backward method inspired by Euler. It uses an implicit first-order Runge-Kutta method, where models require values to be in between 0 and 1. A refinement process is used to keep values as precise as possible, so this method is used for models where accuracy is paramount.

These integration methods were all directly implemented through the multiple dialects found within MLIR, from `arith` and `math` to `scf`. This has also corroborated to showing that no extra IR language is needed to express the operability of the ionic models.

Data access. *Accessor* functions are implemented to retrieve values of external variables of ionic models, and state variables of a cell. Stride accesses are enabled by `gather` and `scatter` operations from the `vector` dialect, allowing to fetch state variables stored in non-contiguous memory addresses. We also generate *accessor* functions for single-valued broadcasts, and contiguous memory accesses, the latter being necessary for our code transformation discussed in section 3.4.1. Lines 1 and 18 from Listing 3 show examples of *accessors* for contiguous and non-contiguous memory accesses that fetch data from eight cells in parallel.

Multimodel support. Electrophysiology simulations also allow multiple models to interact, accessing the same data. This leads to a hierarchy of cells relying on a parent-offspring relation. Offspring cells are allowed to access and modify the content (or state) of their parent. In the openCARP framework, this feature is supported through a combination of conditional statements that check the existence of the parent and its values, and function pointers that connect the appropriate parent data with its offspring. We support this feature by conditionally accessing data from the parent through MLIR `gather` and `scatter` operations that also handle such conditions. If the parent information cannot be found, it falls through the common local variable storage.

The code generation process described previously supports most of the features found in the original openCARP code generator. More precisely, 43 out of 47 ionic models for cardiac cell simulation are supported, and illustrate the flexibility of our code design. In the next sections, we describe two optimization opportunities for improving the performance of the generated code: a data layout transformation, and an optimization on the LUT interpolation.

3.4 Code Transformations

Two code transformations were implemented for improving the performance of the generated code: (1) a data layout transformation to avoid the effects of non-consecutive data storage of variables within ionic models in memory; and (2) the vectorization of LUT interpolation.

3.4.1 Data layout transformation. From the data perspective, ionic models are described as a combination of shared and private information among cells. While the former is defined as a read-only region that delivers no optimization opportunities (SIMD memory loads of a single data are usually efficiently implemented by the hardware), the latter has been originally modeled as to regroup values of a single cell in a contiguous manner (an array-of-structures, AoS). This design becomes non-optimal when multiple cells are processed in parallel, as is the case of vector memory accesses in our solution.

We implemented a data layout transformation to avoid the effects of non-consecutive data storage of cell variables within ionic models. This classical approach consists of rearranging the same state variable from successive ionic cells consecutively: data is stored in an array-of-structures-of-blocks (or array-of-structures-of-arrays, AoSoA) form [14, 34], a combination of the classical array-of-structures (AoS, non-consecutive) and structure-of-arrays (SoA, completely consecutive but large) forms. Using the AoSoA data storage format, we:

1. avoid memory operations on addresses that are far from one another - and thus avoid TLB misses,
2. improve data locality - and thus improve cache accesses,
3. enable efficient vector load/store hardware operations.

This transformation is implemented as part of the code generation process, and can be enabled through a compiler flag.

3.4.2 Linear interpolation optimization. By analyzing the generated assembly code and checking the hardware performance counters in our benchmarks, we noticed that in many models, one particular function is called very often and needs to be manually vectorized: the LUT (lookup table) interpolation function, a powerful means within ionic models to avoid recomputation of mathematical formulas. LUT variables are computed using a linear interpolation from a set of precomputed values, in a user-defined interval and step. Although LUT utilization significantly improves the performance of models (reaching more than 6× from the non-LUT version), we have seen considerable speedup degradation when relying on the original scalar LUT implementation that the compiler could not automatically vectorize. To overcome this issue, we implemented a fully vectorized MLIR version of LUT interpolation, leading to a considerable gain in performance. A call to its function implementation (`LUT_interpRow_n_elements_vector_8xf64`) is shown in line 21 of Listing 3.

To conclude, along with the aforementioned optimizations, having our code generation fully compatible with MLIR further motivates us to use the myriad of available transformations. Loop invariant code motion and common subexpression elimination are two examples of in-tree optimizations that are also beneficial for speeding up performance.

4 Experimental Results

In this section, we evaluate our proposed technique on a 2× 18-core Cascade Lake Intel Xeon Gold 6240 @2.6GHz CPU, with Turbo Boost and Hyperthreading disabled, and 192GB (5.3 GB/core) of RAM @2933MT/s. The Intel Xeon Cascade Lake processor architecture supports all the three `SSE`, `AVX2`, and `AVX-512` vector instruction sets that we tested.

We implemented our proposed compiler scheme on top of the openCARP source from the git source repository[1] (Jul. 2022). We compiled the codes using the LLVM infrastructure from trunk (July. 2022), which includes all necessary compilation tools, from the Clang compiler to MLIR.

We evaluated our proposed technique on 43 different ionic models available in openCARP. The benchmark program we used is provided by the openCARP package as the `bench` binary, which runs by default a $100,000$ steps simulation (one-second duration with a $0.01ms$ time step). It calls the ionic model at each time step to compute the state of each mesh element as described in the model. Each model was run using a total of 8,192 cells, in order for the largest models not to take more than two hours to execute, and thus, limiting the duration of the experiments to a few hours. Execution times were measured by running the models five times, eliminating the two extrema, and averaging the remaining three.

Experiments were conducted using three vector architectures: `SSE` - with a vector size of two doubles; `AVX2` (four doubles); `AVX-512` (eight doubles). On each architecture, we evaluated the codes on a number of threads (and cores) ranging from 1 to 32, and used the geometric mean (*geomean*) to average speedup results in all cases.

4.1 Single Thread Execution

Figure 2 shows speedups comparing the execution time of the baseline openCARP version to our *limpetMLIR* version, both on one thread. The horizontal axes are composed of

[1] https://git.opencarp.org/openCARP/openCARP

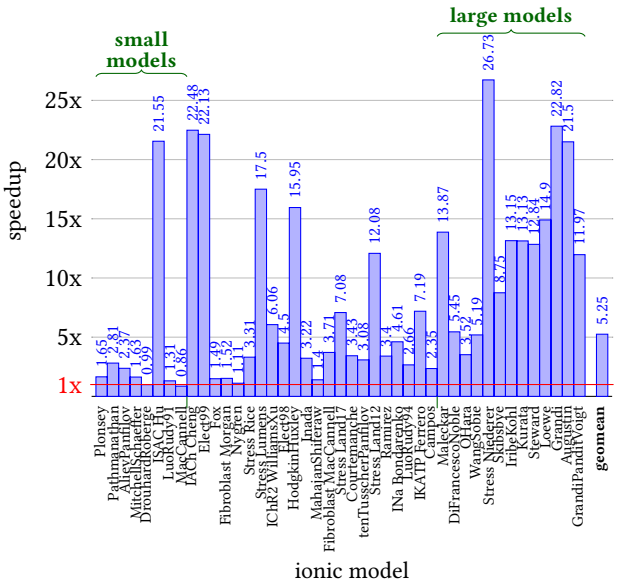

Figure 2. Speedup of the *limpetMLIR* version of the code compared to the baseline openCARP version, using one single thread (sequential) on an AVX-512 architecture.

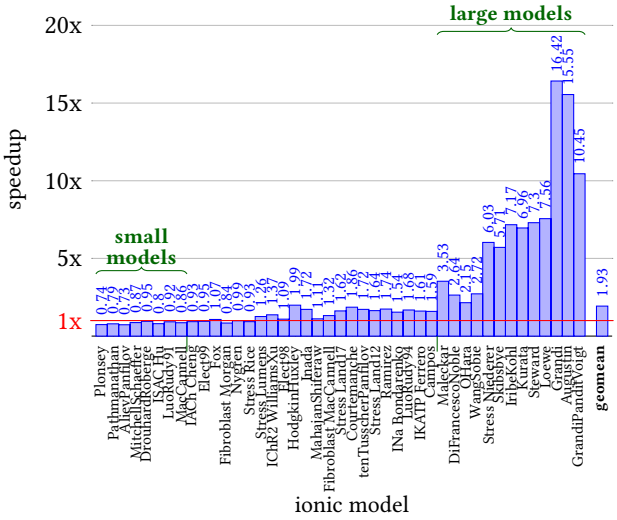

Figure 3. Speedup of the *limpetMLIR* version of the code compared to the baseline openCARP version, using 32 OpenMP threads on a 32 cores AVX-512 architecture.

43 ionic models, ordered from the shortest to the longest execution time from the baseline openCARP version. We arbitrarily split those models into three sets of *small*, *medium*, and *large* ionic models. The small set is composed of the eight models running in less than a minute on our experimental platform, the medium one of 22 models running in 1-5 minutes, and the large one of 13 models taking more than 5 minutes. Large models are usually the most precise and close to the physiology, and as such, they are the most relevant ones for many practical applications, *e.g.*, virtual drug testing in cardiac research.

The *limpetMLIR* version achieves a geomean speedup of 5.25× on AVX-512 architecture. These results show one important aspect of our code generation: the acceleration can be much higher than the size of the vectors, up to more than 26×. Although it may sound surprising, the effects of our optimizations go far beyond the raw vectorization and CPU computation power, reaching also how memory is accessed (simultaneous load/store assembly instructions) and taking advantage of our data layout optimization (efficient use of the memory caches). A different version of the code might also trigger other compiler optimizations that affect, *e.g.*, register allocation, pipelined execution, and out-of-order execution.

The observed speedups are low and irregular in small models, and more significant and consistent for larger models. This is expected: on short codes executing short optimized loops in less than a fraction of a millisecond, it is more difficult to achieve good performance than on longer loops containing more computationally-expensive operations. Some notable exceptions (*e.g.*, ISAC Hu) are more operationally intensive than appeared to be: they share the characteristics of (1) calling costly mathematical functions that were efficiently vectorized[2] by our optimizer *and* (2) not using lookup-tables (LUT).

4.2 Thirty-two Threads Execution

Figure 3 presents the speedup results on a 32 OpenMP threads execution (with using 32 physical cores). The measured speedups compare the baseline and *limpetMLIR* versions in the same conditions, *both running in parallel on 32 threads*: the 1× line represent the execution time of the baseline openCARP parallel code on 32 cores. The *limpetMLIR* version achieves a geomean speedup of 1.93×, but only 0.83× on small models, 1.34× on medium models, and a very good 6.03× on large models. Smaller models with very short execution times suffer a slowdown, mainly because of the synchronization and optimization overheads, or because they are by nature memory-bound and not compute-bound.

4.3 Execution Time and Speedup over SSE, AVX2, and AVX-512

We confirmed our analysis of those differences between *small*, *medium*, and *large* models in fig. 4. We compare the average execution times (y-axis) of the three classes of models running on 1 to 32 cores (x-axis). The dashed lines represent linear speedup. All models running in less than some 50 seconds suffer a slowdown compared to the dashed line. In small models, we observe that the scalability is very poor due to the execution of very short parallel loops: the overhead of synchronizations between threads is very high compared

[2]We rely on Intel's Short Vector Math Library (SVML) library for the vectorization of mathematical functions.

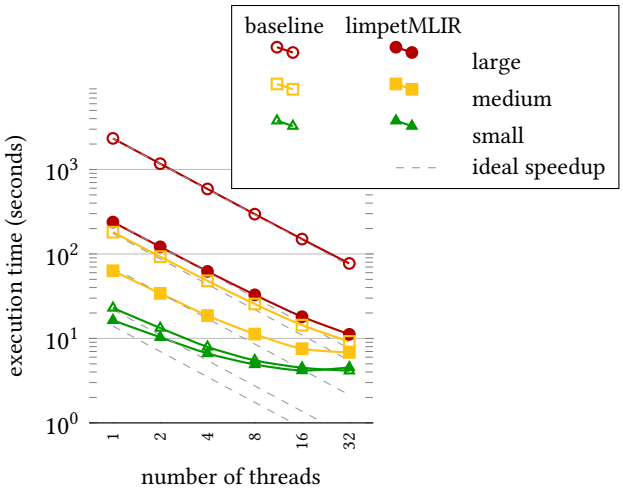

Figure 4. Average execution times of three classes of ionic models: small, medium, large (on AVX-512)

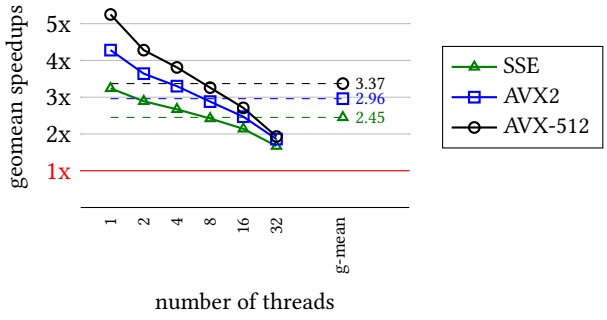

Figure 5. Geomean speedups for SSE, AVX2, and AVX-512 across varying threads (in the power of two)

to the computation time itself, and the curve flattens as the number of cores increases. Although our optimizations (filled symbols) have some positive effect using a small number of threads, they induce a slowdown when reaching 32 cores, as the gain due to parallelism also completely disappears. This is less perceptible in medium models, but still present starting at 8 cores: the *limpetMLIR* execution times get closer to the baseline version at this point. In large models, the *limpetMLIR* version consistently executes $8-10\times$ faster than the baseline, along with an almost ideal parallel speedup both on the baseline and the optimized version.

Figure 5 shows the geometric mean speedups, with respect to the baseline openCARP version, achieved by the *limpetMLIR* version on all the three (SSE, AVX2, and AVX-512) vector architectures across varying threads from 1 to 32. In all cases the AVX-512 architecture outperforms AVX2 and AVX2 outperforms SSE. This behavior is expected as AVX-512 calculates eight values for one hardware operation, whereas AVX2 calculates four, and SSE calculates two. Notice that this is true even if instructions cost and frequency might differ between these three vector architectures.

The difference flattens as the number of cores increases, mainly due to the slowdowns of small models. When restricting those results to the set of large models, we get consistent speedups of $3.80\times$ on SSE, $5.13\times$ on AVX2, and $6.03\times$ on AVX-512 on 32 cores. The overall geomean speedup over all models and all architectures is $2.90\times$.

4.4 Impact of Data Layout Optimization

We found that the data layout optimization was essential in increasing the speedups of medium and large ionic models. This happens because they access more memory (state value) than smaller models. For instance, the Stress_Niederer model has its speedup increased from $4.98\times$ to $6.03\times$ in a 32-thread AVX-512 configuration. The geomean speedup of all models, in a 1 to 32 thread AVX-512 configuration, goes from $3.12\times$ to $3.37\times$ thanks to the data layout optimization.

4.5 Roofline Model

Figure 6 shows the roofline model for our various ionic models. The operational intensity on the x-axis is the number of floating point arithmetic operations divided by the number of memory operations (in Flops/Byte). The number of memory loads and stores were extracted by instrumenting the generated MLIR code of the ionic models. The number of arithmetic operations were measured for each ionic model using the processor performance counters.

The y-axis of fig. 6 represents the GFlops/s performance, as the number of arithmetic operations divided by the execution time, on our 32 cores AVX-512 platform. The peak performance using 32 cores was measured experimentally with the *Empirical Roofline Tool (ERT)* [35] as 760GFlops/s, DRAM bandwidth as 199GB/s, and L1 cache bandwidth as 1052GB/s. Notice that the maximum DRAM bandwidth according to the architecture specification is 140.8GB/s (shown as a gray dashed line in the figure).

One can observe in fig. 6 that many of these codes have an operational intensity lower than the DRAM bandwidth *versus* performance limit (around 4 Flops/Byte); the majority of them are memory-bound. Codes of the large class perform quite well: those on the right of the figure are compute-bound and of same order as the peak 760 GFlops/s limit (GrandiPanditVoigt for example), and those on the left are also close to the memory maximum bandwidth limit (OHara and Wang-Sobie for example). OHara and some medium models (*e.g.*, Courtemanche) exceed the DRAM bandwidth thanks to their efficient cache usage.

There are models with less than 20 GFlops/s performance from in the *small* and *medium* models and they are mostly memory-bound. The DrouhardRoberge model in particular does 19 GFlops/s, but its operational intensity is less than 1/4 Flops/Byte. We observed the slowdown for the small models (the *limpetMLIR* version is below the baseline version), as explained earlier (fig. 4) by their very short execution time.

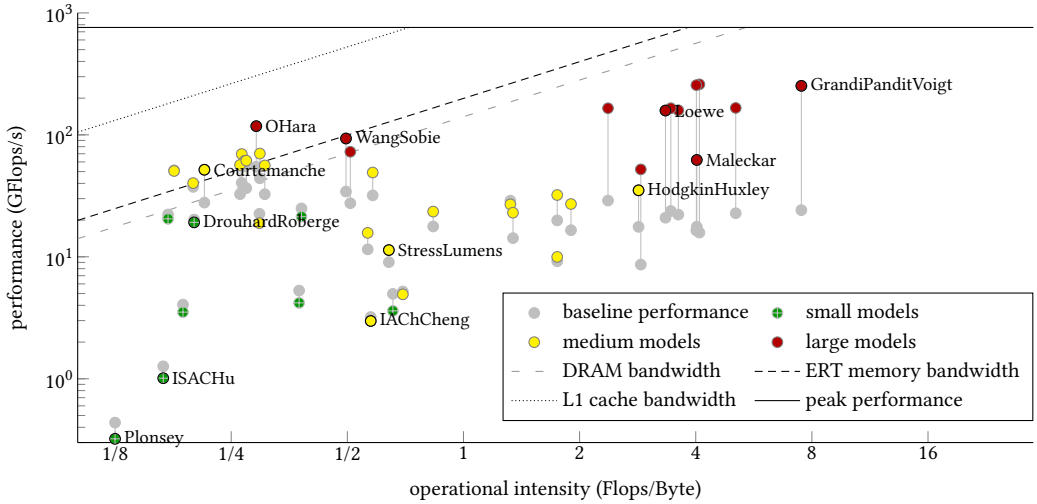

Figure 6. Roofline model for the different ionic models with `AVX-512` vectors on 32 cores when compared to baseline openCARP (peak performance of our experimental platform: 760 GFlops/s, DRAM bandwidth: 199 GB/s, L1 cache bandwidth: 1052 GB/s)

Overall, those roofline results are as expected. The roofline model just confirms our previous analyses and shows that many of our optimized codes are reasonably close to the maximal performance of this architecture. Some improvements still seem possible in some of them (*e.g.*, HodgkinHuxley, Maleckar), and this will be investigated in the future.

5 Discussion

We discuss the broad applicability and generalization of the optimized code generation proposed in *limpetMLIR*. Our technique essentially applies to a parallel loop in which we can arrange data access uniformly. A particular instance of this pattern is *stencil computations*. The simulation of cardiac tissues we have dealt with can be seen as a stencil computation expressed in a DSL. The heavy computations lie in a parallel for loop, in which is computed the updated states of the stencil elements, for each time step of the simulation.

Many DSL relies on code generators to produce high-level language code while the stencil computation part exhibits properties that enable a more specific compilation process. In that case, the burden of optimizing the code falls on a general compiler. In our case, `clang` and `gcc` failed to vectorize the loop even when specifying aggressive optimization options. Intel `icc` 19.1.3 could vectorize the loop when annotated with the OpenMP `simd` directive, but only reached an overall AVX-512 geomean speedup of 2.19×, much lower than *limpetMLIR* (3.37× as seen in fig. 5). Indeed, the loop body contains many challenging constructs for the compiler's auto-vectorizer, among which are complex control-flow operations, function calls, or pointer arithmetic with hard aliases used in complex stride-based memory accesses for which compilers are unable to derive access patterns.

More generally, our proposal is applicable and beneficial to a parallel loop whose body has the following properties: (i) the code (or DSL) can be expressed using MLIR dialects; (ii) loop iterations should perform regular access to data stored in arrays to enable our `accessor` functions to `gather to` and `scatter from` vector types; and (iii) if the code contains control flow operations, it has to be SIMD-friendly for the vectorization to be efficient. For example, the vectorization of an if/else condition requires both blocks to be executed and element-wise selected according to a mask, which may lead to performance degradation in large portions of conditional code. If the code only contains compute intensive arithmetic operations and math library function calls, it can be efficiently vectorized.

The main difficulty to generalize our technique is to translate the body of the loop to MLIR vectorized code. However, using a DSL harnesses the expressivity of the loop body so that the translation of the DSL operations to MLIR code has about the same complexity as translating them to C code.

6 Related Work

Our solution intersects with previous research in two subjects: (i) from an application-focused side, frameworks that simulate electrophysiological systems through differential equations, such as FEniCS [17], Myokit [7], Chaste [21] and even openCARP [26], have their own code generators; and (ii) proposals of compiler extensions, languages and transformations to accelerate domain-specific applications.

Myokit [7] is a Python-based tool for modeling and simulating cardiac cellular electrophysiology that share similar simulation characteristics with openCARP. Its major advantage lies in the ability to export models into multiple formats and programming models, such as, openCL, CUDA,

and Matlab. FEniCS [17] focuses primarily on solving partial differential equations to simulate systems of finite elements, and thus, it has been conceived for modeling and simulating physics problems. Chaste [21] is another example of an open-source tool for the simulation of mathematical models in physiology and biology. In spite of these frameworks advance simulation of real systems, their code generators do not intend to provide a tight integration with the optimizing compiler, and rather means for simulation. While Myokit can produce code for heterogeneous architectures with OpenCL and CUDA support, it still relies on the compiler to find opportunities for parallelism. On the other hand, FEniCS handles parallelism through an OpenMP SIMD directive and, therefore, it also delegates the optimization task directly to the compiler. Contrary to the code generation process from these frameworks and the original openCARP code generation, our tight integration with novel compiler technology enables to provide helpful optimization hints to the compiler.

The emergence of MLIR has brought new possibilities for the use of multiple simultaneous IRs within different abstraction levels. It has also driven the idea of using MLIR as a building block for new IRs in order to permit interleaving levels of abstractions and further optimization opportunities.

Gysi et al. [9] propose a hierarchy of dialects (IRs) for GPU-based stencil computations that is effective in weather and climate applications. A multi-level rewriting flow is proposed to progressively lower abstractions level-by-level, applying optimizations at the most appropriate abstraction. The approach has shown significant speedup in comparison to state-of-the-art solutions for climate and weather simulation, proving that extra levels of abstractions can help to devise new optimizations. Sommer et al. [29] propose a dialect, and a lowering process to optimize sum-product network inference in both CPUs and GPUs, while DistIR [28] is an IR for distributed computation that employs MLIR to optimize neural networks. Recently, many works have proposed to extend MLIR with new dialects to analyze, optimize and accelerate heterogeneous applications in a variety of domains [12, 18, 24]. Our work, however, requires no extra abstraction layer and no additional dialect, and can be seen as complementary to previous solutions. We make use of MLIR as an enabling tool for our code generator and compiler transformations, similar to some previous works [6, 13, 33]: Bondhugula [6], and Katel et al. [13] present evaluations of the modularity of MLIR with sequences of transformations and customizable passes to optimize matrix multiplications. Vasilache et al. [33] builds composable abstractions for leveraging tensor algebra computation.

Polygeist [23] is possibly the project that most resembles our work. It was primarily proposed as an entry point for generating MLIR affine code from C/C++ polyhedral-compatible code. The project has since evolved to support a wider set of C/C++ languages, and could now be seen as a frontend for these languages into MLIR. One could potentially benefit from this new support and use it to transform the original C/C++ generated code from openCARP to MLIR. However, this approach would lead to many complications: openCARP does not have a compiler-friendly implementation. It heavily relies on function pointers, external linear solvers, and pointer-based data structures for representing the state of each cell, along with their potentials and currents. A translation from the original C/C++ implementation of openCARP to MLIR would be cumbersome, in order to adapt each unfriendly feature to a data structure that can be understood by MLIR and its dialects. It also does not cope well with any change that may be done within the ionic model framework, as any important modification will reflect into a new implementation change into MLIR transformations. Our solution is directly embedded into EasyML, the markup language used to describe ionic models, and eases MLIR code generation by adapting to its data structures.

7 Conclusion

We proposed several compilation techniques based on MLIR to automatically transform a *parallel for* loop embedding an ionic model computation (described using a DSL) into an efficient *vectorized parallel for* loop. It illustrates how a novel compiler technology such as MLIR can be used to produce efficient code in situations where traditional compilers fail in vectorization.

Our ongoing work aims to generalize our approach to enable ionic models not only to execute efficiently on CPUs, but also on other heterogeneous hardware supported by MLIR. Having *e.g.*, both CPU and GPU codes can further benefit from task-based programming libraries for heterogeneous architectures, such as StarPU [3]. We also consider exploring some ionic models description or optimization variations, such as an efficient spline interpolation method to replace or complement in some cases the currently used linear interpolation. Future investigations will also focus on power consumption *versus* compute time performance evaluation, along with enabling compiler-aware techniques for the use of approximate computing [22] in ionic models.

Acknowledgments

The authors would like to thank the anonymous reviewers for their thorough and valuable comments.

This work was supported by the European High-Performance Computing Joint Undertaking EuroHPC under grant agreement No 955495 (**MICROCARD**), co-funded by the Horizon 2020 programme of the European Union (EU), and France, Italy, Germany, Austria, Norway, and Switzerland (https://microcard.eu).

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

A Artifact Appendix

A.1 Abstract

This artifact provides all required tools and dependencies needed to compile and execute applications, and generate figures 2, 3, 4 (optional), and 5 of our paper. Among the tools you will find are: the openCARP source code that implements *limpetMLIR* (the main contribution of this paper), LLVM-15.0.2 infrastructure, with all necessary compilation tools, from the Clang compiler to MLIR, and evaluation scripts.

More specifically, our artifact (comprised file) consists of:

- A docker image containing all dependencies needed for the experiments,
- A Dockerfile showing all dependencies needed by the experiments,
- Patchs to LLVM and openCARP, and scripts to run the ionic models (benchmarks) and save the results in .txt files.
- And inside the docker image, `libintlc.so.5` and `libsvml.so` libraries to support vectorization.

For each of the ionic model under consideration, the artifact evaluation can be used to compute (i) the execution time resulting from using *limpetMLIR* openCARP and baseline openCARP, (ii) the speedup compared to baseline openCARP. Note that the exact speedups may differ from the ones reported in the paper, because of the differences in the actual hardware.

A.2 Artifact Check-list (meta-information)

- **Algorithm:** *LimpetMLIR* to generate optimized code for ODEs.
- **Program:** *LimpetMLIR* and baseline openCARP code with ionic models (benchmarks).
- **Compilation:** Clang, Clang++, and Python.
- **Data set:** 43 ionic models available in openCARP cardiac electrophysiology simulator.
- **Run-time environment:** Our artifact has been developed and tested on Linux environment.
- **Hardware:** We recommend a 2x 18-core Cascade Lake Intel Xeon Gold 6240 @2.6GHz CPU, with Turbo Boost and Hyperthreading disabled, and 192GB (5.3 GB/core) of RAM @2933MT/s with different vector instruction sets (`SSE`, `AVX2`, and `AVX-512`) for establishing the exact results presented in our paper.
- **Execution:** We provide scripts to (i) build openCARP (both versions), (ii) run the experiments and (iii) calculate speedups as displayed in the paper.
- **Output:** Executing the scripts will output simulation data and execution time for each ionic model (both *limpetMLIR* and baseline version).
- **How much time is needed to complete experiments:** To reproduce (i) Figure 2 takes around 10 hours; (ii) Figure 3 takes around 30 minutes; (iii) Figure 5, which includes the results for Figures 2 to 4, takes 30 hours approximately. One should notice that experiments highly depend upon the processor in use. Execution times mentioned above were collected in a *2x 18-core Cascade Lake Intel Xeon Gold 6240 @2.6GHz CPU*. Since Figure 5 contemplates all results needed for previous figures, running its experiments will guarantee that all figures can be generated.
- **Publicly available:** Yes
- **Code licenses:** ACADEMIC PUBLIC LICENSE

A.3 Description

A.3.1 How delivered. As a comprised file, allowing users to choose from two approaches: (1) docker image with all dependencies and scripts needed for the experiments, and (2) scripts, patchs to LLVM and openCARP, and a Dockerfile that users may use to build their own image to run experiments. Although both approaches can be used, we advise users to rely on (1) since all that is needed is already included in the image.

A.3.2 Hardware dependencies. An x86-64 machine with support for `SSE`, `AVX2`, and `AVX-512` vector instruction sets. The results in this paper were obtained using a 2x 18-core Cascade Lake Intel Xeon Gold 6240 @2.6GHz CPU machine.

A.3.3 Software dependencies. All software dependencies have been pre-installed in the provided docker image. However, if the user chooses to use approach (2) from section A.3.1, Intel libraries `libintlc.so` and `libsvml.so` should be provided and are not delivered separately in this artifact (but only inside the Docker image). Docker software is required for both approaches.

A.3.4 Data sets. 43 ionic models available in openCARP cardiac electrophysiology simulator.

A.4 Installation

Download the compressed file available in ACM DL [31] at https://doi.org/10.1145/3554349, extract it, and enter in the `cgo-paper9/` directory.

- For approach (1):
(the first command can take several minutes to complete)

```
$ docker load < docker-paper9.img
$ docker run ––volume $(pwd):/results -it
cgo-paper9-artifact bash
```

- For approach (2):

```
$ docker build -f Docker-cgo-paper9 . -t
cgo-paper9-artifact
$ docker run --volume $(pwd):/results -it
cgo-paper9-artifact bash
```

More information can be found in the README.md file inside this artifact. The next sections will explain how to run and evaluate applications using the two approach.

A.5 Experiment Workflow

Once the docker image is set up, you can run the `evaluation.sh` script to build and run ionic models on *limpetMLIR* and baseline openCARP version. We provide various command line options to the script in order to run different experiments. The options are:

```
$ ./evaluation.sh -fig2 true # run experiments for Fig. 2,
$ ./evaluation.sh -fig3 true # run experiments for Fig. 3,
$ ./evaluation.sh -fig5 true # run experiments for Fig. 2-5.
```

By default `-fig3` option is set to true. Once the script completes its evaluation, you can see all the output result files in the folder named `output`.

A.6 Evaluation and Expected Result

Once all the output files available are in the `output` folder, you can run the `res.sh` script to see the speedups obtained thanks to *limpetMLIR* compared to the baseline version. Similarly, `res.sh` provides various options:

```
$ ./res.sh -fig2 true # generates fig2.pdf of Figure 2,
$ ./res.sh -fig3 true # generates fig3.pdf of Figure 3,
$ ./res.sh -fig4 true # generates fig4.pdf of Figure 4,
$ ./res.sh -fig5 true # generates fig5.pdf of Figure 5.
```

You can find the PDF plots in the `output` folder.

Expected results can be analyzed for each figure produced by the evaluation process:

- **Figure 2** depicts the speedup of *limpetMLIR* against the baseline openCARP version, using one thread on an `AVX-512` architecture. Speedups are expected in most of the applications, with those marked as **large models** showing a more significant speedup compared to the **medium** or **small** ones;

- **Figure 3** shows similar speedup comparison, but using 32 threads on a 32 cores `AVX-512` architecture. The multi-thread environment has a deeper impact on small applications. Equivalent performance or even slowdowns are expected for these applications due to synchronization overheads, or because they are by nature memory-bound. Therefore, speedups should be observed for **medium** and **large** models, with the latter showing better overall speedups due to being more compute-bound;

- **Figure 4** illustrates the average execution time of the three classes (**small**, **medium**, and **large**) on an `AVX-512` architecture. The figure is expected to show that **large** models scale better than the other two classes, and that our technique shows better performance over the baseline since its lines are higher in the figure;

- Lastly, **Figure 5** summarizes the effects of different architectures (`SSE`, `AVX2`, and `AVX-512`) across multi-thread environments (in the power of two). We expect to see overall speedups for all combinations of number of threads and architectures. However, the trend will show better speedups (1) for fewer threads due to effects of synchronization overhead and higher cache usage for large threads counts, and (2) for higher vector length instruction sets (speedup of `AVX-512` $\geq$ speedup of `AVX2` $\geq$ speedup of `SSE`).

One should notice speedups are highly dependent upon the processor frequency, number of threads and cache size. Thus, we do not expect for users of this artifact to obtain the exact same results found in section 4. Instead, results should show that *limpetMLIR* outperforms the baseline on the **large** and **medium** application set, while **small** applications can potentially observe slowdowns.

A.7 Experiment Customization

Users can customize experiments in the following ways:

- We have written the scripts in a simple way for easy understanding and customization. You can use `./bin/bench` executable available in build folders (`build_base, build_sse, build_avx2 and build_avx512`) to run/test different experiments.

- Our comprised file consists not only of a docker image prebuilt with all dependencies needed, but also a Dockerfile, scripts for running and obtain speedups, and an LLVM patch (link) that fixes SVML code generation. Users may use the Dockerfile to set up their own testing environment, build and install all dependencies needed for the experiments. Keep in mind that access to Intel's SVML library (not included inside this artifact, but only inside the Docker image) is needed to reproduce our results.

- You may also venture yourself on creating ionic models. Add a new file in `openCARP/physics/limpet/models` with your own equations, following the syntax of EasyML. Update `openCARP/physics/limpet/models/mlir_imp_list.txt` to tell *limpetMLIR* to use the MLIR code generation.

A.8 Notes

- We recommend disabling Intel Turbo Boost and Hyper Threading technologies in the host machine to avoid the effects of frequency scaling and resource sharing on the measurements.
- We request the user to make sure that the `libsvml` library is linked to the `bench` executable because `math` operations are vectorized using `libsvml`.

References

[1] 2021. EasyML. https://opencarp.org/documentation/examples/01_ep_single_cell/05_easyml.

[2] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (Feb. 2011), 187–198. Issue 2. https://doi.org/10.1002/cpe.1631

[4] Aart JC Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *arXiv preprint arXiv:2202.04305* (2022).

[5] BN Biswas, Somnath Chatterjee, SP Mukherjee, and Subhradeep Pal. 2013. A discussion on Euler method: A review. *Electronic Journal of Mathematical Analysis and Applications* 1, 2 (2013), 2090–2792.

[6] Uday Bondhugula. 2020. High performance code generation in mlir: An early case study with gemm. *arXiv preprint arXiv:2003.00532* (2020).

[7] Michael Clerx, Pieter Collins, Enno de Lange, and Paul G.A. Volders. 2016. Myokit: A simple interface to cardiac cellular electrophysiology. *Progress in Biophysics and Molecular Biology* 120, 1 (2016), 100–114. https://doi.org/10.1016/j.pbiomolbio.2015.12.008 Recent Developments in Biophysics & Molecular Biology of Heart Rhythm.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[9] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-Accelerated Climate Simulation. *ACM Trans. Archit. Code Optim.* 18, 4, Article 51 (sep 2021), 23 pages. https://doi.org/10.1145/3469030

[10] Ernst Hairer, Syvert P. Norsett, and Gerhard Wanner. 1993. *Solving Ordinary Differential Equations I. Nonstiff Problems* (2nd rev. ed. 1993. corr. 3rd printing ed.). Springer, Berlin. https://archive-ouverte.unige.ch/unige:12346 ID: unige:12346.

[11] Michael Hucka, Andrew Finney, Herbert M Sauro, Hamid Bolouri, John C Doyle, Hiroaki Kitano, Adam P Arkin, Benjamin J Bornstein, Dennis Bray, Athel Cornish-Bowden, et al. 2003. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19, 4 (2003), 524–531.

[12] Ryan Kabrick, Diego A Roa Perdomo, Siddhisanket Raskar, Jose M Monsalve Diaz, Dawson Fox, and Guang R Gao. 2020. CODIR: towards an MLIR codelet model dialect. In *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE, 33–40.

[13] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2022. MLIR-Based Code Generation for GPU Tensor Cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/3497776.3517770

[14] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. 2015. Automatic data layout optimizations for gpus. In *European Conference on Parallel Processing*. Springer, 263–274.

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

[16] Catherine M Lloyd, Matt DB Halstead, and Poul F Nielsen. 2004. CellML: its future, present and past. *Progress in biophysics and molecular biology* 85, 2-3 (2004), 433–450.

[17] Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. 2012. FFC: the FEniCS form compiler. In *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 227–238.

[18] Martin Lücke, Michel Steuwer, and Aaron Smith. 2021. Integrating a functional pattern-based IR into MLIR. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 12–22.

[19] Megan E Marsh, Saeed Torabi Ziaratgahi, and Raymond J Spiteri. 2012. The secrets to the success of the Rush–Larsen method and its generalizations. *IEEE transactions on biomedical engineering* 59, 9 (2012), 2506–2515.

[20] Alexander McCaskey and Thien Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 255–264.

[21] Gary R Mirams, Christopher J Arthurs, Miguel O Bernabeu, Rafel Bordas, Jonathan Cooper, Alberto Corrias, Yohan Davit, Sara-Jane Dunn, Alexander G Fletcher, Daniel G Harvey, et al. 2013. Chaste: an open source C++ library for computational physiology and biology. *PLoS computational biology* 9, 3 (2013), e1002970.

[22] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–33.

[23] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59.

[24] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. Comet: A domain-specific compilation of high-performance computational chemistry. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 87–103.

[25] Pras Pathmanathan and Richard A Gray. 2013. Verification of computational models of cardiac electro-physiology. *Int J Numer Method Biomed Eng* 30, 5 (Nov. 2013), 525–544.

[26] 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. 2021. The openCARP simulation environment for cardiac electrophysiology. *Computer Methods and Programs in Biomedicine* 208 (2021), 106223. https://doi.org/10.1016/j.cmpb.2021.106223

[27] S Rush and H Larsen. 1978. A practical algorithm for solving dynamic membrane equations. *IEEE Trans Biomed Eng* 25, 4 (July 1978), 389–392.

[28] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Andrew Fitzgibbon, and Tim Harris. 2021. DistIR: An Intermediate Representation for Optimizing Distributed Neural Networks. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 15–23. https://doi.org/10.1145/3437984.3458829

[29] Lukas Sommer, Cristian Axenie, and Andreas Koch. 2022. SPNC: An Open-Source MLIR-Based Compiler for Fast Sum-Product Network Inference on CPUs and GPUs. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11. https://doi.org/10.1109/CGO53902.2022.9741277

[30] Joakim Sundnes, Robert Artebrant, Ola Skavhaug, and Aslak Tveito. 2009. A second-order algorithm for solving dynamic cell membrane equations. *IEEE Transactions on Biomedical Engineering* 56, 10 (2009), 2546–2548.

[31] Arun Thangamani, Tiago Trevisan Jost, Vincent Loechner, Stéphane Genaud, and Bérenger Bramas. 2023. Artifact for Lifting Code Generation of Cardiac PhysiologySimulation to Novel Compiler Technology. https://doi.org/10.1145/3554349.

[32] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 27–38. https://doi.org/10.1109/LLVMHPC54804.2021.00009

[33] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. *CoRR* abs/2202.03293 (2022). arXiv:2202.03293 https://arxiv.org/abs/2202.03293

[34] Nicolas Weber and Michael Goesele. 2014. Auto-Tuning Complex Array Layouts for GPUs.. In *EGPGV@ EuroVis*. 57–64.

[35] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

Received 2022-09-02; accepted 2022-11-07