GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR

Tiago Trevisan Jost, Arun Thangamani, Raphaël Colin, Vincent Loechner, Stéphane Genaud, Bérenger Bramas

GPU Code Generation of Cardiac Electrophysiology Simulation with MLIR

Tiago Trevisan Jost*, Arun Thangamani*, Raphaël Colin, Vincent Loechner(✉), Stéphane Genaud, and Bérenger Bramas

Inria Nancy Grand Est and ICube Lab., University of Strasbourg, France.
{arun.thangamani,berenger.bramas}@inria.fr
{raphaelcolin,loechner,genaud}@unistra.fr

Abstract. We show the benefits of the novel MLIR compiler technology to the generation of code from a DSL, namely EasyML used in openCARP, a widely used simulator in the cardiac electrophysiology community. Building on an existing work that deeply modified openCARP's native DSL code generator to enable efficient vectorized CPU code, we extend the code generation for GPUs (Nvidia CUDA and AMD ROCm). Generating optimized code for different accelerators requires specific optimizations and we review how MLIR has been used to enable multi-target code generation from an integrated generator. Experiments conducted on the 48 ionic models provided by openCARP show that the GPU code executes 3.17× faster and delivers more than 7× FLOPS per watt than the vectorized CPU code, on an Nvidia A100 GPU versus a 36-cores AVX-512 Intel CPU.

Keywords: automatic GPU code generation · code transformation · MLIR · domain-specific languages · heterogeneous architectures

1 Introduction

Cardiac electrophysiology is a medical specialty in which the research community has long been using computational simulation. Understanding the heart's behavior (and in particular cardiac diseases) requires to model the ionic flows between the muscular cells of cardiac tissue. Such models, called *ionic models*, describe the way an electric current flows through the cell membranes. The widespread practice in this field is for experts to describe their ionic model in a domain-specific language (DSL), which essentially enables to model the current flow by ordinary differential equations. The openCARP[1] [15] simulation framework has been created to promote the sharing of the cardiac simulation efforts from the electrophysiology community. To describe ionic models, this framework offers a DSL named EasyML [20], from which a code generator can derive `C/C++` code.

The next major advances in cardiac research will require to increase by several orders of magnitude the number of cardiac cells that are simulated. The ultimate

* Both authors contributed equally to the paper
[1] https://opencarp.org

goal is to simulate the whole human heart at the cell level [16], that will require to run several thousands of time steps on a mesh of several billions of elements.

In order to achieve such simulations involving exascale supercomputers, the generation of efficient code is a key challenge. This is the general purpose of our work. We propose to extend the original openCARP code generator using the state-of-the-art compiler technology MLIR (Multi-Level Intermediate Representation) [11] from LLVM [10].

MLIR offers a means to express code operations and types through an extensible set of *intermediate representations* (IR), called dialects, each dedicated to a specific concern, at different levels of abstractions. The code representation can use a mix of operations and types from different IRs. Representing the code at an appropriate level of abstraction enables transformation and optimizations that would be difficult to achieve with a single general purpose IR. An example of a high-level abstraction dialect is the `linalg` dialect which defines operations on matrices, and comes with a set of optimizations that can take advantage of some mathematical properties.The `linalg` operations can then be transformed into operations expressed in a less abstract IR (this is called *lowering*). An intermediate level of abstraction is the `scf` dialect to represent control flow operations like loops and conditional statements. Eventually the code is lowered in a dialect, such as `llvm`, that has the ability to generate machine code.

In a previous paper [18], we introduced architectural modifications to openCARP to enable the generation of CPU vectorized code using MLIR. We have shown that the MLIR generated vectorized code outperforms the original `C/C++` code compiled (and optimized) by standard compilers (`clang`, `gcc`, and `icc`).

In this paper, we present our work to take further advantage of the capabilities of MLIR, to extend the code generator to GPU code generation. This represents a stepping stone for the final objective of the MICROCARD[2] project to be able to combine instances of optimized CPU and GPU kernels, that will eventually be dynamically scheduled by a runtime on the varied computing resources of a supercomputer. The main contributions of this work are: (i) a code generator from a DSL to efficient heterogeneous code by leveraging MLIR; (ii) an integration of this code generator in the compilation flow of a cardiac electrophysiology simulator (openCARP); (iii) a performance improvement of openCARP beneficial to the electrophysiologists, paving the way to larger scale experiments.

The paper is organized as follows. Section 2 details the extension we propose for the openCARP compilation flow. Section 3 presents our GPU code generation using MLIR. A discussion about the challenges we faced and the reusability of our work is provided in section 4. Performance and energy efficiency is evaluated on CPU and GPUs in section 5 on all 48 ionic models available in openCARP. Related work is covered in section 6, and finally section 7 concludes the paper.

[2] https://microcard.eu

```
1  Vm; .external(Vm); .nodal();
2  Iion; .external(); .nodal();
3  group {  mu1 = 0.2;  mu2 = 0.3; }.param();
4
5  t_norm  =  12.9; vm_norm = 100;  vm_rest = -80;
6  touAcm2=100/12.9;
7  V_init = 0; Vm_init=vm_rest; K = 8; epsilon = 0.002; a = 0.15;
8
9  U = (Vm-vm_rest)/vm_norm;
10 diff_V = -(epsilon+mu1*V/(mu2+U))*(V+K*U*(U-a-1))/t_norm;
11
12 Iion = (K*U*(U-a)*(U-1)+U*V)*touAcm2;
```

Listing 1: AlievPanfilov ionic model written in EasyML

2 Compilation Flow in openCARP

2.1 EasyML: Description of Ionic Models

In openCARP, biomedical mathematicians use EasyML [20] as a DSL to write ionic models (as mathematical equations) that represent the current that flows through a cell of cardiac tissue from a given cell state. Many other languages (e.g. CellML, SBML, MMT) used to write ionic models can be easily translated to/from EasyML through scripts available in openCARP and Myokit [7]. Some characteristics of EasyML are as follows:

1. SSA (static single assignment) [8] representation, so all variables are defined as mathematical equalities in an arbitrary order;
2. specific variables prefixes/suffixes (such as `_init`, `diff_`, etc.);
3. calls to *math* library functions;
4. *markup* statements to specify various variable properties, such as: which method to use for integrating differential equations (`.method(`*m*`)`), whether to pre-compute a lookup table of predefined values over a given interval (`.lookup`), which variables to output (`.trace`), etc.;
5. it is not Turing-complete since it cannot express loops, control flow, or sequence of elements – but there can be tests expressed as restricted `if/else` statements or as C-like ternary operators.

Example. Listing 1 shows the EasyML code for the very simple AlievPanfilov [1] ionic model. The variables `Vm` and `Iion` (voltage and current) are declared as external on lines 1-2 as they will be used by other parts of the openCARP simulator. Line 3 defines a group of runtime controllable parameters. Lines 5-7 initialize some variables. Line 10 calls the *Forward Euler (fe)* default integration method to recompute `V` by using the DSL `diff_` prefix, and line 12 computes the `Iion` (current) flow out of the cell.

2.2 Code Generation in openCARP

The openCARP simulation handles a mesh of elements, potentially containing many biological cells, but for simplification purposes we will refer to a mesh element as a *cell* in the following. A simulation step is composed of two stages:

```
1  void compute_AlievPanfilov(...) {
2  #pragma omp parallel for schedule(static)
3  for (int __i=start; __i<end; __i++) {
4    AlievPanfilov_state *sv = sv_base+__i;
5    //Initialize the ext vars to current values
6    Iion = Iion_ext[__i], Vm = Vm_ext[__i];
7    //Compute storevars and external modvars
8    U = ((Vm-(vm_rest))/vm_norm);
9    Iion = (((((K*U)*(U-(a)))*(U-(1.)))+(U*sv->V))*touAcm2);
10   //Complete Forward Euler Update
11   diff_V = (((-(0.002+((p->mu1*sv->V)/(p->mu2+U))))*(sv->V+((8.*U)*((U
       -(0.15))-(1.)))))/12.9);
12   V_new = sv->V+diff_V*dt;
13   //Finish the update
14   Iion = Iion, sv->V = V_new;
15   //Save all external vars
16   Iion_ext[__i] = Iion, Vm_ext[__i] = Vm;
17 }
```

Listing 2: Baseline openCARP generated code snippet of the AlievPanfilov model from Listing 1

- a *compute stage*: the ionic model is used to compute the current (`Iion`) that flows in and out of each cell; all cells share read-only data and each one of them updates its private state variables;
- a *solver stage*: the computed current is passed to a linear solver to recompute each cell membrane electric potential (`V`). OpenCARP uses either PETSc [4] or Ginkgo [2] as a linear solver.

In this paper, we only discuss the first stage compilation flow, code generation, and optimization opportunities. The solver is out of the scope of this paper.

The upper part of Figure 1 (so excluding the dashed line box) shows the original code generation flow in openCARP. A python code generator called `limpet_fe` takes an EasyML model description as input and generates an Abstract Syntax Tree (AST) from it. From the AST, `limpet_fe` emits `C/C++` output code with (i) functions to initialize parameters, lookup tables, and state variables, and (ii) a `compute` function that scans all cells in a *for* loop, to calculate the output `Iion` current and update the state variables. Finally, the generated code is compiled using a standard `C/C++` compiler and the object file is used for simulation. We call this original compilation flow of openCARP the *baseline*.

Listing 2 shows a snippet of the `compute` function emitted by the openCARP baseline version for the AlievPanfilov model. The `for` loop at line 3 iterates across all cells. Notice the preceding `omp parallel for` directive (line 2) as there is no loop-carried dependency between iterations. Line 4 retrieves the state variables pointer. Line 9 and 14 are the calculation of the new current (`Iion`) and its flow. Lines 11-12 integrate the potential with the *Forward Euler* method.

2.3 Vectorized CPU Code Generation using *limpetMLIR*

The fact that the main loop independently computes the cell's states suggests that we can parallelize computations further using different types of parallel hardware. One possibility is to exploit the CPU SIMD units by vectorizing the

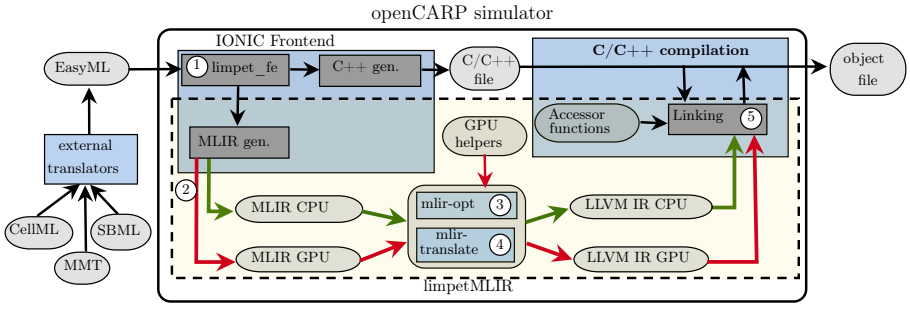

Fig. 1: Overview of the code generation, from the EasyML model to an object file. The dashed line box shows how *limpetMLIR* fits into the original code generation process, to emit optimized code for CPU and GPU.

code. While the current mainstream compilers try to automatically generate vectorized code, they fail to do so in the presence of complex control-flow operations, external function calls, or pointer arithmetic used in complex stride-based memory accesses, which is the case in the openCARP generated code.

We have presented in a previous paper [18] how this limitation can be dealt with by generating code represented in specific intermediate representations (IR) suited to express vectorization. For that purpose, we used MLIR [11] from the LLVM [10] compiler infrastructure. Using some of the conventional MLIR dialects (namely `vector`, `scf`, `arith`, `math`, `memref`, and `openmp`) we have integrated an extension called *limpetMLIR* in the openCARP code generator. The most computationally intensive part of the code (the *compute* function's main loop) is generated in MLIR. The advantage of using MLIR lies in its abstraction with respect to the final target: for example, instructions in the `vector` dialect might be compiled for different hardware (`SSE`, `AVX2`, `AVX-512`, or even `SVE`, `NEON`, ...) while being on par regarding performance with the equivalent code using target specific intrinsics.

Another advantage is that MLIR can target completely different hardware, like GPUs. We present in the following the challenges that we faced to generate efficient GPU code from a real application DSL using the MLIR infrastructure.

3 Optimized GPU Code Generation

3.1 Overview of GPU Code Compilation Flow

Figure 1 shows the compilation flow of *limpetMLIR*. The process is as follows:

1. From the EasyML description the `limpet_fe` python program creates an AST, which serves as a common entry point for the baseline openCARP and *limpetMLIR*.
2. Using python bindings, our *limpetMLIR* code generator emits MLIR code using the `scf`, `arith`, `math` and `memref` dialects; the control flow expressed

in `scf` allows the latter MLIR passes to lower it to a parallel control flow in the `gpu` dialect. In listing 3, lines 2-3 show the two `scf.for` loops, which will be translated into an outer loop and an inner loop iterating over the GPU `blocks` and `threads` within a block respectively.

③ The MLIR lowering pass converts the MLIR code into a GPU device code part using a specific GPU low-level dialect. This low-level dialect can be either `nvvm` (the Nvidia CUDA IR) or `rocdl` (the AMD ROCm IR) depending on the target GPU architecture. This pass finally outputs a binary blob that will be integrated by the next step.

④ The MLIR translator pass converts the MLIR code into a CPU host code part (represented using the `llvm` dialect) to LLVM IR, and that calls the kernel embedded in the binary blob.

⑤ Last is the linking phase, where `C/C++` and LLVM IR GPU files are linked together into an object file using LLVM.

3.2 *LimpetMLIR* for GPU

Some specific features of openCARP were adapted or extended by the CPU version of *limpetMLIR* for their integration with MLIR and optimized code generation: lookup tables (LUTs), integration methods, multi-model support, and data layout transformation. Similarly, the GPU version of *limpetMLIR* provides support for host/device memory management, integration methods, lookup tables, and multi-model support. They are provided as a set of GPU helper functions and as specific MLIR emitted code, and are described hereafter.

Memory Management. One well-known pitfall regarding performance is the data transfers between host and device because of the PCIe bus bottleneck. These necessary transfers are not part of the MLIR code generation process, but are inserted into the code that wraps the ionic model computation for the following reasons: (i) we want to keep the structure of the MLIR code as similar as possible for all types of devices, so we focus on generating MLIR for the compute function only, (ii) other openCARP software parts access this memory (e.g solvers), and (iii) we want to precisely control the data movements behavior regarding performance. We implement the memory management preferably using unified memory with `cudaMallocManaged` or `hipMallocManaged`. As a side note, it happened on our AMD test platform that `hipMallocManaged` is not supported and falls back to inefficient data transfers. In that case, we could easily change it to explicit allocation and memory copies between host and device.

Integration Methods. The complex mathematical functions and equations in the integration methods are represented using MLIR. The MLIR code that we generate for GPU has the same structure as the one generated for the vectorized version. They differ in the data type they use (`vector<?xf64>` vs. `f64` data type) and their respective memory load/store primitives. We only need the `arith` and `math` dialects to represent the following methods: *Forward euler*, *Runge-Kutta* with 2 and 4 steps (*rk2* and *rk4*), *Rush-larsen*, *Sundnes*, and *Markov_be*. Brief

```
%7 = memref.load %6[%c0_3] : memref<?xf64>
scf.for %arg9 = %0 to %3 step %c1 { // iterate over blocks
 scf.for %arg10 = %c0 to %c512 step %c1 { // iterate over threads
 // ... code skipped for space
  scf.execute_region {
   %11 = arith.cmpi slt, %9, %1 : index
   cf.cond_br %11, ^bb1, ^bb2
  ^bb1:  // pred: ^bb0
   // ... code skipped for space
   %cst_11 = arith.constant 8.000000e+00 : f64
   %cst_12 = arith.constant 1.500000e-01 : f64
   %cst_13 = arith.constant 1.000000e+00 : f64
   %cst_18 = arith.constant 2.000000e-03 : f64
   %35 = arith.mulf %5, %31 : f64
   %36 = arith.addf %7, %24 : f64
   %37 = arith.divf %35, %36 : f64
   %38 = arith.addf %cst_18, %37 : f64
   %39 = arith.negf %38 : f64
   %40 = arith.mulf %cst_11, %24 : f64
   %41 = arith.subf %24, %cst_12 : f64
   %42 = arith.subf %41, %cst_13 : f64
   %43 = arith.mulf %40, %42 : f64
   %44 = math.fma %40, %42, %31 : f64
   %45 = arith.mulf %39, %44 : f64
   %cst_19 = arith.constant 1.290000e+01 : f64
   %46 = arith.divf %45, %cst_19 : f64
   // ... code skipped for space
```

Listing 3: MLIR code snippet for GPU generated by *limpetMLIR* for the AlievPanfilov model from Listing 1

information on these integration methods can be found in [18, Sect. 3.3.2]. In listing 3 lines 10-26 show the MLIR representation for *Forward euler*. Line 10 in listing 1 and lines 11-12 in listing 2 represents the same *Forward euler* code in EasyML and in the baseline generated openCARP code, respectively. The *Rosenbrock* integration method using function calls was implemented using GPU helper functions.

GPU Helper Functions. During the compute stage, openCARP performs function calls to (i) lookup table based interpolation (to use pre-computed LUT values for complex mathematical functions), and (ii) the *Rosenbrock* integration method (to perform LU decomposition and integration). MLIR cannot inline a function call and it is very hard to automatically generate MLIR code for those function calls. So, we add those function calls during the MLIR code generation and we write their respective implementations in GPU device code such that they are called and executed on GPU without any call back to CPU. For example in listing 4, lines 5 and 7 are the function calls. Also, we provide `accessor` functions that assist in loads and stores of external variables of ionic models and state variables of cells.

Implementation Effort. For our implementation, we wrote about 10k source lines of code (39% python, 26% MLIR, 23% `C++`, and some GPU kernel and CMake code). The total auto generated lines of code for all 48 ionic models are as follows: *baseline*: 39621; vectorized *limpetMLIR*: 111883; GPU: 78025.

```
// ... code skipped for space
scf.execute_region {
 // ... code skipped for space
 %147 = memref.view %146[%c0_127][] : memref<?xi8> to memref<1xi8>
 func.call LUT_interpRow(%144,%122,%147):(f64,i32,memref<1xi8>)->()
 // ... code skipped for space
 func.call rosenbrock_StepX(%814,%830,%c3):(memref<1xi8>,f32, i32)->()
 // ... code skipped for space
```

Listing 4: MLIR code snippet generated for GPU by *limpetMLIR* for the BONDARENKO model

4 Discussion

We have explained above the overall software architecture of openCARP and how we have fit into it our optimizations of the critical part concerning the ionic model computation. We now discuss in this section the general principles and caveats to consider when envisaging such an approach.

Writing Abstract Optimizations. The identification of a general pattern of optimization can be an incentive to use MLIR to describe this pattern. For instance in our case, the loop that carries no dependencies between iterations can trigger the idea that it can be represented as a parallel loop, whatever the available hardware to execute it in parallel. Generating only the loop control flow in a language-independent representation enables to later generate specialized code for different programming models or accelerators, as we do in our case for OpenMP, CPU vector processing units or GPUs. Although writing this representation still requires to precisely understand the code semantics and the optimization potential like an expert would do to optimize for a given device, MLIR offers a more abstract and therefore portable way of coding these optimizations. The programmer is indeed relieved from the burden of writing the eventual implementation for each specific target architecture as he/she can rely on the lowering passes included in MLIR. And as MLIR continues to evolve, relying on LLVM as a back-end, it is expected that optimization improvements and support for new targets will be integrated over time.

Choosing Dialects. One challenge lies in how to represent the input problem using the large number of available dialects in MLIR. In our example, the control flow expressed by the loop can be represented in the `scf` dialect or in the `affine` dialect. The `affine` dialect is more specific than `scf` as it represents the particular case of affine loops (which matches our case). As MLIR offers a transformation from `affine` into the `gpu` dialect it would be a possible choice. However, our objective to have the most abstract representation makes us choose `scf` because it allows to derive both GPU and vectorized CPU code. For the GPU code, we wrote a simple pass to transform `scf` to `affine` and rely afterwards on the `affine` to `gpu` pass provided by MLIR. For the vectorized version, we used the MLIR pass that lowers `scf` to the `cf` dialect, which represents the control

flow using SSA blocks.

A New Dialect? MLIR can be extended by defining new dialects along with associated transformation passes. Hence, a legitimate question is to assess if the problem would be better expressed in a new dialect, especially if optimizations are more tractable using operations and types at this level. Some works [17,9] have proposed such higher level dialects while interacting with other domains. In this work, we have not felt the need for a new dialect as the set of existing ones is expressive enough to represent the statements and mathematical operations that we need to support heterogeneous code generation.

5 Experimental Results

We evaluate *limpetMLIR* on an A100 Nvidia GPU (9,700 GFLOP/s peak performance on *double*s, 400W), on an AMD Radeon Instinct MI50 GPU (6,600 GFLOP/s, 300W), and on a 2x 18-core Cascade Lake Intel Xeon Gold 6240 @2.6GHz (850 GFLOP/s, 2x 150W), turbo boost and hyperthreading disabled, 192GB of RAM @2933MT/s. On CPU we ran (i) the 36 OpenMP threads baseline openCARP, and (ii) the 36 OpenMP threads `AVX-512` *limpetMLIR* version.

We implemented *limpetMLIR* on top of the openCARP source from the git repository. We compiled them using the LLVM compiler infrastructure tag `15.0.2`, which has all necessary compilation tools including Clang and MLIR. We run all 48 ionic models available in the openCARP benchmarks, using the `bench` executable to run the compute step alone and get a trace every 100 steps. We used 819,200 cells with a 10,000 step simulation. Each model is run five times, the two extreme measures are eliminated and the remaining three are averaged.

The total number of floating point operations necessary to run each ionic model was measured with the hardware counters on the CPU. The GPU probably does less operations due to mathematical functions being optimized, but we used the same baseline value measured on CPU for a fair comparison.

For the GPU execution we chose a block and thread dimension of 1, a `number of threads per block` (CTA size) of 64, and a `number of blocks` of {number of cells/64}. We empirically determined that this provides the best performance results on our platforms for all models. This value can be easily adjusted if running on different hardware.

5.1 Performance

Nvidia CUDA Performance. Figure 2 shows the floating-point operations executed per second by the CPU baseline, *limpetMLIR* `AVX-512` vectorized, and A100 GPU versions of openCARP. The x-axis lists all 48 ionic models and the y-axis is the GFLOP/s performance. On the `x-axis`, we sorted the ionic models from the shortest to the longest execution time (of the baseline). We classified them into three categories: 17 ionic models executing in less than one minute into the *small* category, 19 ionic models executing in 1-5 min into the *medium*

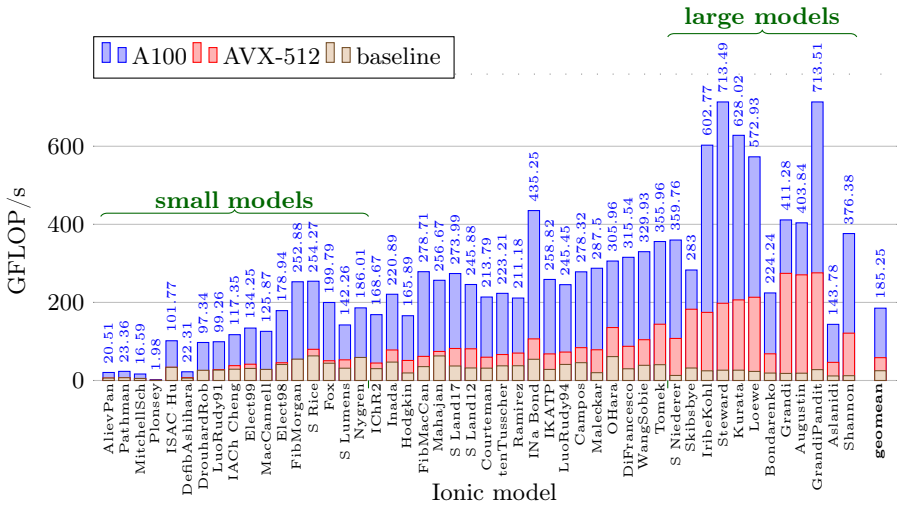

Fig. 2: Performance on Nvidia A100, in giga floating operations per second

category and the remaining 12 with more than 5 min execution time into the *large* category. Note that the large models are usually also the most realistic ones (they are the closest to the physics) and are widely used in realistic physiological simulations.

From fig. 2 we can observe with no surprise that the GPU code performs better than the CPU openCARP versions in all ionic models. GPU optimized codes report the highest GFLOP/s for the large models, that perform the most computations. Overall, considering the geometric mean, we reach 185 GFLOP/s on this platform, the GPU optimized code executes 3.17× faster than the vectorized CPU code, and 7.4× faster than the baseline openCARP.

However, the model that exhibits the best performance reaches 713 GFLOP/s, that is only 1/13 of the raw performance the A100 can deliver. Also, the A100 has 11.4x the raw performance of our test bed CPU (850 GFLOP/s) so the average gain of 3.17x seems pretty low. The reason is that those ionic models, taken from a real simulation application, have a pretty low compute intensity: a geomean of 0.35 flop/byte. This means that they execute many memory operations along with floating point calculations. Better performance on large models is explained by their greater compute intensity: a geomean of 3.02 flop/byte if we exclude Bondarenko and Aslanidi. Those two specific models have lower performance results than the other ones, as they have in common to call a memory intensive integration method (Rosenbrock). Overall, the low compute intensity explains that the GPU performance is far from the maximal hardware performance, and that the CPU with multiple levels of fast and large caches is better at this.

We also report that one of those best performing model (Steward) was manually written in CUDA by our HPC expert, using explicit memory copies. We measured very similar performance between this code and the *limpetMLIR*

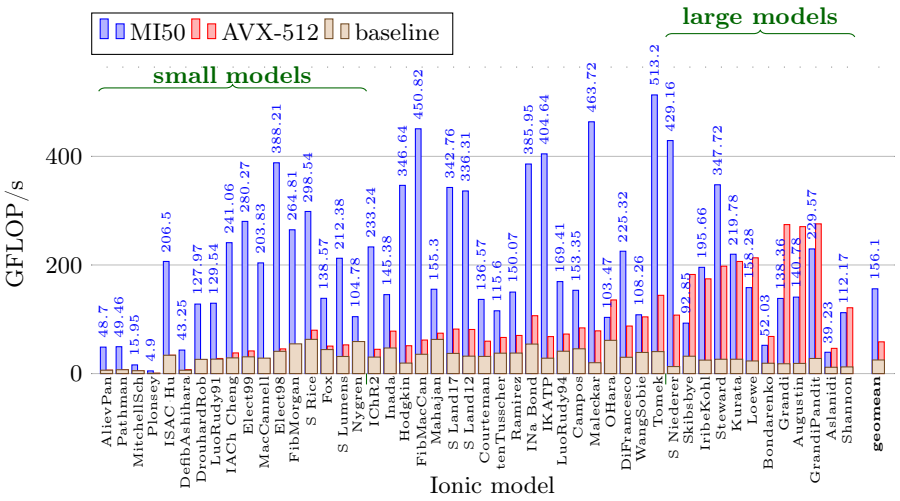

Fig. 3: Performance on AMD MI50, in giga floating operations per second

generated code (the handwritten code is less than 5% faster).

AMD ROCm Performance. We did the same experiments on the AMD MI50, as reported in fig. 3. We reach a geomean performance of 156 GFLOP/s on this platform, and the overall results are pretty similar except for one point: the MI50 performs better than the A100 on small and medium ionic models while we can observe the opposite for the large models. The AMD version is sometimes even outperformed by the CPU vectorized version (for example on Grandi and Augustin). The difference in memory management (see section 3.2) between the CUDA and ROCm implementations is the main reason for this lower performance on large models and better performance on small models.

Considering the geometric mean, the AMD ROCm *limpetMLIR* code executes 2.67× faster on MI50 than the vectorized CPU version. This number compares to 3.17× on A100, since the A100 has almost 50% more maximal raw performance than the MI50.

5.2 Energy Efficiency

We reported GFLOP/s raw performance results as it is good practice, but those numbers are not very meaningful when comparing completely different architectures with very different raw computing power. The FLOP per consumed Joule is a much better scale to compare them with the perspective of running on energy-aware supercomputers. We measured the total energy consumption by running the benchmarks on the CPU using the hardware counters, as the sum of package and RAM consumption; on the Nvidia GPU, we used the `nvidia-smi` command

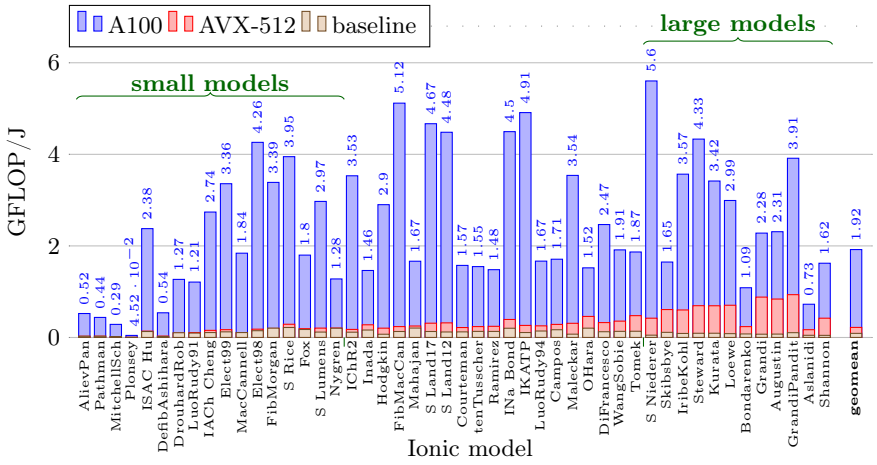

Fig. 4: Energy efficiency on Nvidia A100, in GFLOP per Joule

to regularly poll instant power consumption during the kernels execution, and averaged them; a similar command (`rocm-smi`) was used on AMD GPUs.

Figure 4 shows the energy efficiency (`y-axis`) of the *limpetMLIR* generated code on the A100 GPU compared to the CPU baseline and vectorized versions, for all 48 ionic models (`x-axis`). A first remark from this figure is that the difference between small, medium, and large models is much less significant than on the previous figure. Only the very small models performing very few floating point operations, and in general the ones that have a small compute-intensity (e.g., as already noticed, Bondarenko and Aslanidi), have a low efficiency on GPU. The numbers reported in this plot pretty closely relate to the compute-intensity of those different benchmarks. For example, ISAC Hu (col. 4) has an intensity of 1.6 flop/byte, while DefibAshihara (col. 5) has only 0.28 flop/byte and is much less power efficient.

Not reported in our previous work [18], the geomean energy gain of the CPU `AVX-512` vectorized version compared to the baseline openCARP CPU version is 2.3×, and it is especially significant on the large models (7.1×): on CPU, the largest benchmarks have the most energy gain when vectorized. On the other hand, for all benchmarks, the efficiency of the GPU is consistently higher than the best CPU version. Considering all ionic models the geomean energy gain of the A100 GPU over the vectorized CPU version is 8.72×.

We performed the same measurements (not shown in the plot) on the AMD MI50 GPU and obtained similar results: as reported before, the MI50 is faster and has also better energy efficiency than the A100 on small models, but worse efficiency on the medium and large ones. The geomean power efficiency of the MI50 is 1.54 GFLOP/J, a bit lower than the A100 1.92 GFLOP/J, but still much better than the vectorized CPU version 0.22 GFLOP/J.

6 Related Work

Since we integrated our GPU code generation in *limpetMLIR* on top of the vectorized code generator [18], it resembles our very close work and most of related work intersects.

Fried Lionetti et al. [13,12] generated CUDA kernels for electrophysiology simulations in cardiac modeling using *python* tools. With the help of python `sympy` [6] they represented mathematical equations of cell models and translated them into a `C` code. With python `pycparser` they build an AST from the emitted `C` code. Finally, they traverse the AST to emit equivalent CUDA kernels, and applied source-to-source optimizations on the CUDA kernels.

Myokit [7] is a python-based software used for the cardiac simulation of myocytes (cardiac muscular cells). Myokit accepts inputs in multiple formats and can generate heterogeneous source code (`C, python, Matlab, CUDA,` and `OpenCL`), relying on the standard compilers for optimizations. Though openCARP and Myokit share very similar characteristics for ionic cardiac simulation, their purpose is different, the first one targeting simulation of (parts of) the whole organ, the second one considering individual biological cells.

Campos et al. [5] used GPU technology to utilize the parallelism in Lattice Boltzmann method for cardiac simulations using the monodomain model, a different ionic model. Zhang et al. [21] developed a GPU system for cardiac simulation and visualization.

All these code generators rely on external systems or scripts to generate GPU code, and perform source-to-source translation or depend on standard compilers for optimizations. Our code generator differs from theirs, as we incorporate the cardiac simulation code into the compiler IR, and then give hints to the compiler for optimization and code generation for various architectures.

With respect to MLIR, there are works that generate GPU or heterogeneous optimized code. Polygeist [14] acts as a `C/C++` frontend to MLIR and generates `affine` dialect code to better utilize the polyhedral optimization and code generation available in MLIR. As said in previous work [18], Polygeist cannot handle complex codes (like ionic model descriptions) as input. Gysi et al. [9] propose a new dialect for GPU-based stencil computations in weather and climate domains. Both those works followed a similar approach to ours but we have different input requirements.

7 Conclusion

In this paper, we have presented how the quite recent *multi-level intermediate representation* concept that arose from the research community in compilation can be applied to a production-level scientific application, namely openCARP, a cardiac electrophysiology simulator. The challenge is to integrate into the existing code base the generation of highly optimized code both for CPU and GPU. We have explained the modifications we have brought to the code generation process which originally generated C/C++ code from models expressed with a

DSL. The paper discusses the design choices that arise when it comes to choose among the available dialects to represent the code structures and statements at the appropriate abstraction level. We show that we were able to factorize a large part of the MLIR generated code that was used for vectorization in CPU, and explain how the necessary additions to generate GPU code are implemented through the lowering passes. Finally, MLIR allows us to produce code that has the same level of performance as native code but in a more portable way. The evaluation of our GPU version is carried out on the full set of models shipped with openCARP and shows it brings a significant performance improvement over the CPU vectorized version both in terms of execution time and energy efficiency. As a perspective, we want to further extend this work by integrating the code generator with a task-based runtime system like StarPU [3], in order to exploit simultaneously CPU and GPU so able to run experiments at larger scales.

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

Data availability. An artifact (docker image file) for reproducing the results presented in figs. 2 to 4 is provided [19].

References

1. Aliev, R.R., Panfilov, A.V.: A simple two-variable model of cardiac excitation. Chaos, Solitons & Fractals **7**(3), 293–301 (1996). https://doi.org/10.1016/0960-0779(95)00089-5
2. Anzt, H., Cojean, T., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., Ribizel, T., Tsai, Y.M., Quintana-Ortí, E.S.: Ginkgo: A modern linear operator algebra framework for high performance computing. ACM Trans. Math. Softw. **48**(1) (feb 2022). https://doi.org/10.1145/3480935
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 **23**, 187–198 (Feb 2011). https://doi.org/10.1002/cpe.1631
4. Balay, S., et al.: PETSc Web page (2022), `https://petsc.org/`
5. Campos, J., Oliveira, R., dos Santos, R., Rocha, B.: Lattice boltzmann method for parallel simulations of cardiac electrophysiology using gpus. J. Comput. Appl. Math. **295**(C), 70–82 (mar 2016). https://doi.org/10.1016/j.cam.2015.02.008
6. Certik, O.: Sympy python library for symbolic mathematics (2008)

7. Clerx, M., Collins, P., de Lange, E., Volders, P.G.: Myokit: A simple interface to cardiac cellular electrophysiology. Progress in Biophysics and Molecular Biology **120**(1), 100–114 (2016)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 25–35. POPL '89, ACM (1989). https://doi.org/10.1145/75277.75280
9. Gysi, T., Müller, C., Zinenko, O., Herhut, S., Davis, E., Wicky, T., Fuhrer, O., Hoefler, T., Grosser, T.: Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation. ACM Trans. Archit. Code Optim. **18**(4) (sep 2021). https://doi.org/10.1145/3469030
10. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Int. Symp. on Code Generation and Optimization, 2004. pp. 75–86 (2004). https://doi.org/10.1109/CGO.2004.1281665
11. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO). pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308
12. Lionetti, F.V.: Gpu accelerated cardiac electrophysiology. Masters Thesis, University of California, San Diego (2010)
13. Lionetti, F.V., McCulloch, A.D., Baden, S.B.: Source-to-source optimization of cuda c for gpu accelerated cardiac cell modeling. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. pp. 38–49. Springer, Berlin, Heidelberg (2010)
14. Moses, W.S., Chelini, L., Zhao, R., Zinenko, O.: Polygeist: Raising C to polyhedral MLIR. In: 30th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT). pp. 45–59 (2021). https://doi.org/10.1109/PACT52795.2021.00011
15. Plank, G., Loewe, A., Neic, A., Augustin, C., Huang, Y.L., Gsell, M.A., Karabelas, E., Nothstein, M., Prassl, A.J., Sánchez, J., Seemann, G., Vigmond, E.J.: The openCARP simulation environment for cardiac electrophysiology. Computer Methods and Programs in Biomedicine **208**, 106223 (2021). https://doi.org/10.1016/j.cmpb.2021.106223
16. Potse, M., Saillard, E., Barthou, D., Coudière, Y.: Feasibility of whole-heart electrophysiological models with near-cellular resolution. In: 2020 Computing in Cardiology. pp. 1–4 (2020). https://doi.org/10.22489/CinC.2020.126
17. Sommer, L., Axenie, C., Koch, A.: SPNC: An open-source MLIR-based compiler for fast sum-product network inference on CPUs and GPUs. In: 2022 IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO). pp. 1–11 (2022). https://doi.org/10.1109/CGO53902.2022.9741277
18. Thangamani, A., Trevisan, T., Loechner, V., Genaud, S., Bramas, B.: Lifting Code Generation of Cardiac Physiology Simulation to Novel Compiler Technology. In: 21st ACM/IEEE Int. Symp. on Code Generation and Optimization (CGO). ACM, Montréal Québec, Canada (2023). https://doi.org/10.1145/3579990.3580008
19. Trevisan Jost, T., Thangamani, A., Colin, R., Loechner, V., Genaud, S., Bramas, B.: Artifact for GPU code generation of cardiac electrophysiology simulation with mlir. `https://doi.org/10.6084/m9.figshare.23546157` (2023)
20. Vigmond, E.: EasyML. `https://opencarp.org/documentation/examples/01_ep_single_cell/05_easyml` (2021)
21. Zhang, L., Wang, K., Zuo, W., Gai, C.: G-heart: A gpu-based system for electrophysiological simulation and multi-modality cardiac visualization. Journal of Computers **9**(2), 360–367 (2014)