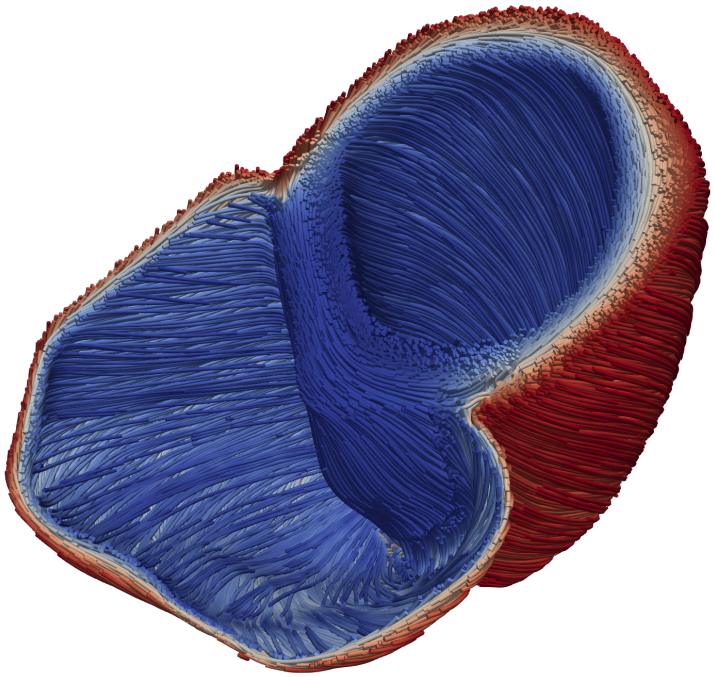
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Iver Håkonsen

GPU-enabled Laplace-Dirichlet Rule-Based Method for Cardiac Fiber Computations

Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
Co-supervisor: James D. Trotter, PhD, Simula Research Laboratory
February 2023

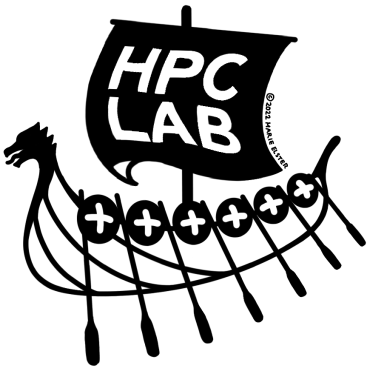


NTNU
Norwegian University of Science and Technology

simula

Iver Håkonsen

GPU-enabled Laplace-Dirichlet Rule-Based Method for Cardiac Fiber Computations



Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
Co-supervisor: James D. Trotter, PhD, Simula Research Laboratory
February 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Project description

Computer simulations of cardiac physiology includes studying underlying mechanisms of the heart under normal as well as diseased conditions. However, a realistic and relevant model of cardiac tissue must include anisotropic features. Moreover, the anisotropy depends on the orientations of cardiac fibres, which are typically obtained by some form of preprocessing before simulations can be carried out. To deduce cardiac fibre orientations, recent approaches employ established facts about the heart's overall structure and geometry and apply rule-based methods. Rule-based methods are, however, very computationally demanding.

The goal of this project is to develop and implement a GPU-accelerated version of the LDRB (Laplace-Dirichlet Rule-Based) method. This is an important step toward simulations of cardiac electrophysiology and mechanics with more elaborate detail in terms of geometry as well as cellular and subcellular details.

The work is to be based on an existing implementation of the algorithm `ldrb` (`https://github.com/finsberg/ldrb`), which is uses FEniCS, an open-source platform for solving partial differential equations. It uses a direct solver, SuperLU, on the Laplace equations that are part of the LDRB algorithm, which works fine for smaller meshes, but does not scale to larger meshes.

The subgoals of the project are as follows:

- Benchmark the existing CPU based implementations of the LDRB algorithm on a selection of heart meshes.
- Write an implementation of the LDRB algorithm using the HIP programming environment such that it can be run on AMD GPUs, and hopefully NVIDIA GPUs as well. Initially the goal is only to look at running the LDRB algorithm itself on GPU, and not the FEM assembly that is required in advance.
- Use the GPU based numerical solvers and preconditioners offered by the *hypre* software library to solve the differential equations, in a hope that it will be both faster, as well as making it possible to use finer grained meshes.
- Finally, as time permits, we wish to look at a GPU based assembly of the finite element matrices, such that the whole computation can be performed on GPU.

Abstract

Cardiology is a major field in medicine, where the ability to model and simulate cardiac functions accurately is of great interest. Knowledge about cardiac fiber orientations is essential in many computational cardiac models. One of the most widely used methods for deriving these cardiac fiber orientations is the Laplace-Dirichlet Rule-Based method (LDRB). The main advantage of this method is that it utilizes partial differential equations (PDEs) to properly capture the complex geometry of the heart. This allows for more anatomically correct fiber orientations than other methods, but it is also computationally intensive.

This thesis describes how we implemented a distributed-memory parallel version of the LDRB algorithm for computing myocardial fiber orientations. The presented implementation also leverages the processing power of one or more many-core GPUs to improve performance compared to only using a high-end multi-core CPU. In addition to being able to use GPUs to solve the PDEs involved in the algorithm, we show how the subsequent steps needed to define fiber orientation can be offloaded to GPU. By limiting ourselves to tetrahedral meshes, we are able to find closed-form solutions for the gradient calculation, projection, and interpolations needed, such that a GPU kernel can efficiently compute them.

Further, we present extensive benchmarking of our implementation of the LDRB algorithm. In comparison with an existing Python-based implementation of LDRB, we show a minimal speedup of 15x on a single CPU core. The performance of our implementation is studied further by scaling it up to 512 CPU cores and by using 1 to 4 GPUs. Our benchmarks show that the PDEs may be solved $\sim$ 1.5x faster on a GPU compared to a 64-core CPU. In the other parts of the algorithm we offloaded to GPU, we show between 1.75x and 19x speedup on one GPU compared to a 64-core CPU. In both cases, it is shown that increasing the number of GPUs results in a close-to-linear speedup. In addition, we compare the performance using GPUs from AMD and NVIDIA. All these results are obtained by processing a set of high-resolution computational meshes with up to 255 million elements, created using an open dataset from a study on acute myocardial ischemia.

Finally, an addition to the method is presented in the form of a heuristic that allows for more accurate reasoning about where in the heart geometry an arbitrary point is, which makes it possible to more robustly define fiber orientations. Directions for future work is also included.

Sammendrag

Kardiologi er et viktig felt innenfor medisin, og muligheten til å presist simulere hjertefunksjoner er av stor interesse i modellering av hjerte. Informasjon om orienteringen til hjertefibre er en essensiell del av mange beregningsbaserte hjertemodeller. En av de mest brukte metodene for å utlede orienteringen til hjertefibre er Laplace-Dirichlet Rule-Based metoden (LDRB). Hovedfordelen ved å bruke denne metoden er at den benytter seg av partielle differensiallikninger (PDEer) for å fange opp den komplekse hjertegeometrien. Dette muliggjør mer anatomisk korrekte hjertefiberorienteringer enn andre metoder, men er også beregningsintensivt.

Denne oppgaven beskriver hvordan vi har implementert en distribuert-minne-parallell versjon av LDRB algoritmen for beregning av hjertefiberorienteringer. Den presentere implementasjonen benytter seg også av beregningskraften til en eller flere GPUer for å øke ytelsen sammenliknet med å kun bruke en flerkjerners CPU. I tillegg til at være stand til å bruke GPUer for å løse PDE'ene som er involvert i algoritmen, viser vi også hvordan de påfølgende stegene som trengs for å definere hjertefiberorienteringer kan flyttes over til GPUer. Ved å begrense oss til tetraediske mesh, er vi i stand til å utlede lukkede løsninger for gradient-beregningene, projeksjonen, og interpolasjonen som kreves, slik at de effektivt kan beregnes i en GPU-kernel.

Videre presenterer vi omfattende referansemåling av vår implementasjon av LDRB algoritmen. I en sammenlikning med en eksisterende Python-basert implementasjon av LDRB viser vi en minimal speedup på 15x på en enkel CPU-kjerne. Ytelsen til vår implementation er studert videre ved å skalere opp til 512 CPU-kjerner, og ved å bruke 1 til 4 GPUer. Våre referansemålinger viser at PDE'ene kan løses $\sim$ 1.5x raskere på én GPU sammenliknet med en CPU med 64 kjerner. I de andre delene av algoritmen var har flyttet over på GPU, viser vi en speedup på mellom 1.75x og 19x på én GPU, sammenliknet med en CPU med 64 kjerner. I begge tilfeller viser vi at når vi øker antall GPUer som tas i bruk, så får vi en tilnærmet linear speedup. I tillegg sammenliker vi ytelsen til GPUer fra både AMD og NVIDIA. Alle disse resultatene er samlet ved å prosessere et sett med høyoppløste mesh med opp til 255 millioner elementer, laget ved bruk av et åpent datasett fra en studie om akutt myokardiskemi.

Til slutt presenter vi et tillegg til metoden i form av en heuristikk som gjør det enklere å presist kunne resonnere om hvor hjertegeometrien et vilkårlig punkt

ligger, som gjør det mulig å definere hjertefiberorienteringer på en mer robust måte. Forslag til fremtidig arbeid blir også presentert.

Acknowledgments

I would first like to thank my supervisor, Professor Anne C. Elster, for her guidance, encouragement, and support, primarily in virtual form from Trondheim. Thank you also for taking your time to visit me on your way through Oslo.

I would like to thank Simula for providing me with a place to work and write after moving back to Oslo, and for giving me access to all the hardware I could ever want from the eX^3 cluster, which has been vital to this thesis.

I would like to express my sincere gratitude to my co-supervisor from Simula Research Laboratory, James D. Trotter. Thank you for tailoring this project for specifically for me, and for spending hours discussing, debugging, and explaining. Thank you for showing a genuine interest in me and my work, first throughout my internships at Simula, and now throughout this thesis.

I would also like to thank Henrik Finsberg, for helping me fill the significant gaps in my knowledge of cardiac physiology in the beginning of this thesis project, and for giving me valuable input on both my work and in the writing of this thesis.

Finally, I want to show gratitude to my family and to my significant other, Oda Wagle, for being my main source of support throughout the entire duration of the five and a half years it took to complete this degree.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

Computer simulations of cardiac physiology are used to study the mechanics of the heart. Many of these models require accurate information about the anisotropic features of cardiac tissue, one of the most important of which is the cardiac fiber orientations. Knowledge about cardiac fiber orientations is, therefore, an important prerequisite when running these models. However, patient-specific cardiac fiber orientations can not be gathered in routine clinical practice and therefore have to be estimated if they are to be included in individualized heart models. There are two main approaches for estimating fiber orientations. The first is an atlas-based approach, where *ex vivo* MRI and diffusion tensor imaging (DTI) is used to generate an estimated model of a patient's cardiac fiber orientation[1, 2]. The second is a rule-based approach, which uses a set of mathematical rules derived from observations made from existing histological and DTI data in order to generate the fiber orientations for a patient-specific anatomical geometry [3]. The rule-based approaches are often used when the DTI data is either missing or lossy. Rule-based methods are also used when the individualized models are created from geometrical data obtained *in vivo*, using methods that do not capture fiber orientation.

Several rule-based methods have been proposed [4, 5], but the most widely used is the Laplace-Dirichlet Rule-Based method (LDRB) [6]. One of the distinguishing features of this method is that it uses partial differential equations (PDEs) to capture the complex geometry of patient-specific anatomical models properly. These PDEs can be solved numerically by using discretization methods such as the finite element method (FEM). This enables the computation of cardiac fibers that are very close to those captured by DTI data. However, solving these PDEs using FEM is very computationally intensive, especially as the granularity of the anatomical model is increased.

One way to reduce computational time is to use hardware accelerators, such as the graphics processing unit (GPU). The use of GPUs is becoming increasingly popular in scientific computing, and the use is especially prevalent in simulations involving finite element methods. Due to their ubiquity and constant need for reduction in simulation time, FEM simulations were one of the first areas where

GPUs were used for general-purpose computing [7].

Recently, focus is also put on being able to utilize hardware from both of the major graphics card vendors, namely NVIDIA and AMD. For the last decade, NVIDIA has been the leading contender when it comes to GPUs in the scientific and high-performance computing world, both on the hardware side and on the software side. The space is, however, changing, and vendors such as AMD are starting to release hardware that is comparable with that of NVIDIA when it comes to performance. While nearly all the top supercomputers in the world use NVIDIA hardware, new supercomputers such as the LUMI system in Finland have chosen to use GPUs from AMD [8]. Being able to support hardware from different vendors is therefore important if we want to keep utilizing the full potential of these hardware accelerators in the future.

1.1 Goals and contributions

In this section, we present the main goals that were outlined at the start of this thesis project, as well as some contributions made beyond the project goals.

The additional sub-goal of doing assembly of the FEM-matrices on the GPU was not carried out due to a lack of time. We do, however, show a way to reduce the time spent in this CPU-bound assembly step when running the rest of the algorithm on GPUs (see Subsection 4.2.7), as well as some discussion of the topic (see Section 5.1).

Goal 1: Benchmarking the existing CPU-based implementations of the LDRB algorithm on a selection of heart meshes.

We show that we successfully run the Python and FEniCS-based [9] `ldrb` [10] implementation of the LDRB algorithm and compare it to our own implementation on a series of meshes. Attempts are made at building other implementations, namely `life`$^{\text{x}}$ and `cardioid` (see Section 1.2). However, strict requirements for dependencies and an outdated build system prevented us from achieving noteworthy results.

Goal 2: Writing an implementation of the LDRB algorithm using the HIP programming environment such that it can be run on AMD GPUs, and hopefully NVIDIA GPUs as well.

We are able to offload all parts of the LDRB algorithm to one or more GPUs, except the final assembly of the FEM matrices. By using the extended GPU programming model offered by the MFEM [11] library, we are able to target both AMD and NVIDIA GPUs through their native HIP and CUDA programming models, respectively.

Goal 3: Using the GPU-based numerical solvers and preconditioners offered by the *hypre* software library to solve the differential equations, to see if it will be both faster, as well as making it possible to use finer-grained meshes.

We show how we utilize the PCG solver and BoomerAMG preconditioner from the *hypre* library to efficiently solve the PDEs that make up the most computationally intensive parts of the LDRB algorithm, both using GPUs and many-core CPUs. The speedups we get from solving these PDEs on GPUs are presented in Subsection 4.2.3.

Additional Contribution: A region-heuristic for LDRB

In addition to fulfilling the original goals for this thesis project, we also created a heuristic that can be added to the LDRB algorithm so that we are able to more accurately reason about where in the myocardium an arbitrary point is, thereby allowing for the definition of more robust fiber orientations (see Section 3.7).

1.2 Related works

The Laplace-Dirichlet Rule-Based method was originally presented in [6] and since then, several open-source implementations of the method have been developed. One example is `ldrb` [10], which is written in Python and utilizes the finite element platform FEniCS [9, 12]. Another example is the Cardioid project [13], an open-source cardiac multiscale simulation suite developed at Lawrence Livermore National Laboratories, which has an implementation of the LDRB algorithm written in C++ that utilizes the MFEM [11] finite element library. The most recent implementation is part of `life`^{x} [14, 15], a library for mathematical problems and numerical methods related to cardiac applications, which is also written in C++ and uses the deal.II [16] finite element core.

In addition to several implementations, there are several published works that investigate alterations or improvements to the method. In [17], the authors present a revised rule-based method called the outflow tract rule-based method (OT-RBM), also referred to as the Doste RBM, which uses separate rules for each ventricle that allows for fiber generation that not only matches histological data in the left ventricle but also in other regions such as the right ventricle endocardium, the septum, and in outflow tracts. In [18], the authors present a version of LDRB that targets fiber generation in the atrium of the heart, an area of the heart not covered by the original algorithm.

To our knowledge, none of these existing implementations or published works have investigated GPU acceleration of the LDRB algorithm.

1.3 Thesis outline

The rest of this thesis is structured as follows:

- **Chapter 2**: Provides background information on myocardial fiber orientations, quaternions, the LDRB method, GPU programming, and FEM.
- **Chapter 3**: Presents the implementation of the algorithm, the approaches taken to offload the individual parts of the algorithm to the GPU, as well as outlining the heuristic we have developed.
- **Chapter 4**: Presents the numerical results from benchmarking of the presented implementation by comparing with an existing implementation and by comparing CPU and GPU performance.
- **Chapter 5**: Discusses the numerical results, as well as some other aspects of the presented implementation of the algorithm.
- **Chapter 6**: Concludes the work and outlines suggested future work.
- **Appendix A**: Contains extra code listings.
- **Appendix B**: Contains the build scripts for the libraries used.

Chapter 2

Background

This chapter contains the background knowledge needed for the rest of this thesis. First, we look at what myocardial fibers are and some of the computational models they are used in. Then we will dive into the theoretical background for the Laplace-Dirichlet Rule-Based method. Next, we will look at the most important parts of the finite element method and the MFEM finite-element library in the context of how they are used to solve the differential equations in the LDRB algorithm. Finally, we will look at the main building blocks and programming models of GPUs, which we will be using to speed up the fiber computations.

2.1 Myocardial fiber orientations

A diagram showing the main components of the heart can be found in Figure 2.1. The regions of interest in the context of this thesis are the ones marked in blue, namely the two lower chambers of the heart (the left and right ventricle), the septum, and the epicardium. The left ventricle is the heart's main pumping chamber, which is responsible for transporting oxygen-rich blood to the rest of the body. The main responsibility of the right ventricle is to pump oxygen-depleted blood to the lungs. The septum is the wall that separates the left ventricle and the right ventricle. In between the left ventricle endocardium and the epicardium is the wall which is often referred to as the "LV free wall", and similarly the wall between the right ventricle endocardium and the epicardium is referred to as the "RV free wall" [19]. We are especially interested in the *myocardium*, colloquially known as the cardiac muscle, that forms these walls.

The myocardium consists of layers of interconnected sheets of tissue. The cardiac muscle fibers lie on the plane of these sheets, and each sheet is loosely connected to other sheets on either side. For this reason, the myocardium is often referred to as having a *sheet laminar* structure. A direct result of this is that the fibers that are adjacent in the same plane, i.e., on the same sheet, are more strongly coupled together than those in the plane of the sheet facing it [21]. These observations form the background for the following axis system of the sheet microstructure. There is one axis pointing in the longitudinal direction of the fibers,

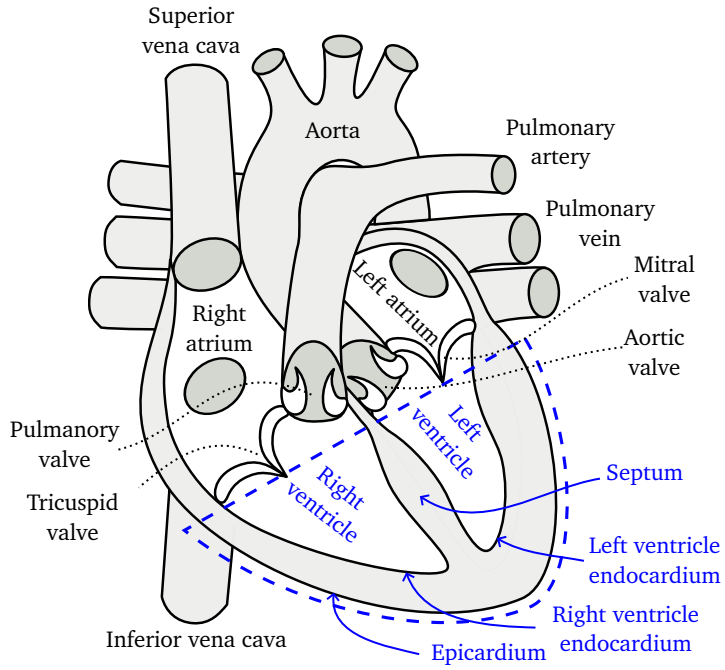


Figure 2.1: Diagram of the human heart, with the parts of interest to this thesis work outlined and marked in blue. Source: Adapted from [20], ©Creative Commons.

the *fiber axis*, denoted by F. Orthogonal to F, but still in the plane of the same sheet, is the *sheet axis*, often also called the *transverse axis*, denoted by T. The final axis is orthogonal to the latter two and is directed normal to the sheet plane, known as the *sheet normal* axis, and is denoted by S [22]. The orientation of the $(F,\ S,\ T)$ axis system in a section of myocardial tissue is illustrated in Figure 2.2.[1]

As mentioned in Chapter 1, knowledge about the myocardial fiber orientations for a given heart model, which is often based on patient-specific data, is important in several computational models aiming to describe how the heart functions. One use case is in passive mechanical models of the myocardium, which aim to determine how the myocardial tissue behaves under mechanical stress [23]. A second use case is in the modeling of how the muscle fibers in the heart contract [22, 24]. A third use case is in the modeling of the action potential, as the electrical conductivity is stronger in the longitudinal direction of fibers compared to the cross-fiber direction. The conductivity is also affected by the sheet structure of the fibers, as the conductivity is higher in the sheet-parallel direction [25].

[1] In the literature the axes are often denoted $\mathbf{f}_0$, $\mathbf{n}_0$ and $\mathbf{s}_0$, but we will stick to the F, S, and T notation as used in [6].

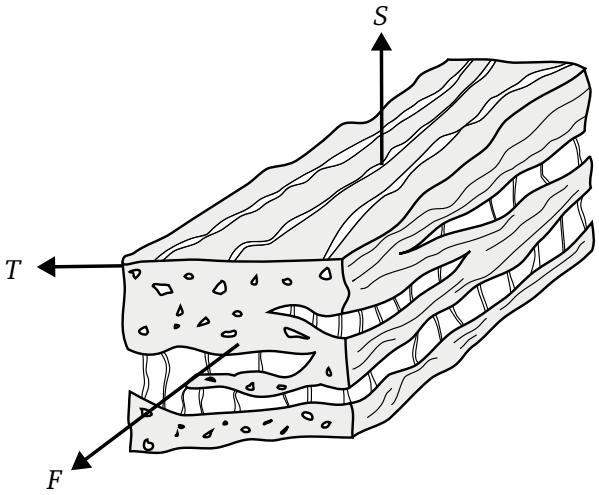


Figure 2.2: The orthogonal $(F,\ S,\ T)$ axis system visualized on a section of the sheet laminar structure of the myocardium. The F axis points in the longitudinal direction of the myocardial fibers. The T axis point across the sheets, orthogonal to the length of the fibers. The S axis point in the sheet normal direction.

2.2 Quaternions

Before diving into the LDRB algorithm, we will first take a small detour to explain some key mathematical aspects of *quaternions*. As we will see in Subsection 2.3.3, the LDRB algorithm uses quaternions to represent the orientations of axis systems.

A quaternion q is often represented in the form

$$q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k},$$

where a, b, c, and d are real numbers, and $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ are the *basic quaternions*. The basic quaternions can be interpreted as unit vectors that point in the direction of three spatial axes. Further, a is often referred to as the *scalar* component, and b, c, and d as the *vector* components. Quaternions can be used to represent rotations in 3D space, just like the more commonly used rotation matrices. Quaternions can be used in place of rotation matrices, with the added advantage of being more compact, efficient, and numerically stable. As quaternions and rotation matrices represent the same actions, we can trivially convert a rotation represented by a matrix to a quaternion and vice versa.

The product, often called the Hamilton product, of two quaternions, q_1 and q_2, is defined as

$$\begin{aligned} q_1 q_2 = \ & a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2 \\ & + (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2)\mathbf{i} \\ & + (a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2)\mathbf{j} \\ & + (a_1 d_2 + b_1 c_2 - c_1 v_2 + d_1 a_2)\mathbf{k}. \end{aligned}$$

The quaternion product is non-commutative. The resulting quaternion of the product $q_1 q_2$ represents the rotation equivalent to a rotation q_2 followed by a rotation q_1.

The dot product between two quaternions, q_1 and q_2, is essentially the same as the 4D Euclidean dot product of the scalar components, and is defined as

$$q_1 \cdot q_2 = a_1 a_2 + b_1 b_2 + c_1 c_2 + d_1 d_2.$$

Just as vectors in Euclidean space, the quaternion dot product relates to the angle between the two quaternions. Let θ be the angle between the quaternions q_1 and q_2 treated as 4D vectors, then $\cos\theta = q_1 \cdot q_2$.

2.3 The Laplace-Dirichlet Rule-Based method (LDRB)

In this section, we present an explanation of the LDRB algorithm from the point of view of how to implement it. We do not go into the physiological theory on which it is based. For the full theory, background, and explanation of the algorithm, see the original paper [6]. The rules that the algorithm is formulated upon are observational and of the form "The longitudinal fiber direction in the ventricular walls is parallel to the endocardial and epicardial surfaces". For the full set of rules, see [6]. Throughout this section, we will define a series of functions that, when combined, make up the full algorithm, which is shown in Algorithm 1. Note that all the presented function definitions (Function 1, 2, 3, 4, and 5), as well as the full algorithm (Algorithm 1), are adapted from the definitions made in the supplementary material of [6].

Algorithm 1: `Laplace-Dirichlet Rule-Based method (LDRB)`

Input: Ω – Mesh
Input: α_{endo}, α_{epi}, β_{endo}, β_{epi} – Fiber (α) and sheet angles (β)
Output: F – Fiber orientations along longitudinal axis
Output: S – Fiber orientations along sheet normal axis
Output: T – Fiber orientations along transverse axis
1 Define surfaces $\partial\Omega_{\text{lv}}$, $\partial\Omega_{\text{rv}}$, $\partial\Omega_{\text{epi}}$, $\partial\Omega_{\text{base}}$,$\partial\Omega_{\text{apex}}$
2 $\phi_{\text{epi}} = \texttt{laplace}(\Omega, \partial\Omega_{\text{epi}}, \partial\Omega_{\text{lv}} \cup \partial\Omega_{\text{rv}})$
3 $\phi_{\text{lv}} = \texttt{laplace}(\Omega, \partial\Omega_{\text{lv}}, \partial\Omega_{\text{epi}} \cup \partial\Omega_{\text{rv}})$
4 $\phi_{\text{rv}} = \texttt{laplace}(\Omega, \partial\Omega_{\text{rv}}, \partial\Omega_{\text{epi}} \cup \partial\Omega_{\text{lv}})$
5 $\psi_{\text{ab}} = \texttt{laplace}(\Omega, \partial\Omega_{\text{base}}, \partial\Omega_{\text{apex}})$
6 $(F\ S\ T) = \texttt{definefibers}(\Omega, \phi_{\text{epi}}, \phi_{\text{lv}}, \phi_{\text{rv}}, \nabla\phi_{\text{epi}}, \nabla\phi_{\text{lv}}, \nabla\phi_{\text{rv}}, \nabla\psi_{\text{ab}}, \alpha_{\text{endo}}, \alpha_{\text{epi}}, \beta_{\text{endo}}, \beta_{\text{epi}})$

Let Ω be a cardiac mesh made up of the two lower chambers of the heart, namely the left and the right ventricle, including also the septum and the epicardium. To determine the fiber orientation in an arbitrary point $x \in \Omega$, we first

need to know which ventricular wall x lies in, and how deep into the wall it lies. This information is determined by solving the Laplace equation,

$$\nabla^2 u = 0, \tag{2.1}$$

with different sets of Dirichlet boundary conditions applied to the boundary surfaces of the cardiac geometry. We set the boundary condition $u = 1$ on one part of the boundary, denoted Γ_1, and the boundary condition $u = 0$ on another part of the boundary, denoted Γ_0. The boundaries Γ_1 and Γ_0, along with the mesh Ω, form the inputs of the `laplace` function, which is defined in Function 1 (page 10).

The first step of the LDRB algorithm is therefore to define the surfaces upon which the boundary conditions are applied. The mesh domain Ω is defined by the volume bounded by four surfaces: The epicardium ($\partial\Omega_{\text{epi}}$), the left ventricle endocardium ($\partial\Omega_{\text{lv}}$), the right ventricle endocardium ($\partial\Omega_{\text{rv}}$), and the base ($\partial\Omega_{\text{base}}$). In addition to the boundary surfaces, we are also interested in the point lying on the ventricular apex ($\partial\Omega_{\text{apex}}$). The boundary surfaces and the ventricular apex are visualized in Figure 2.3.

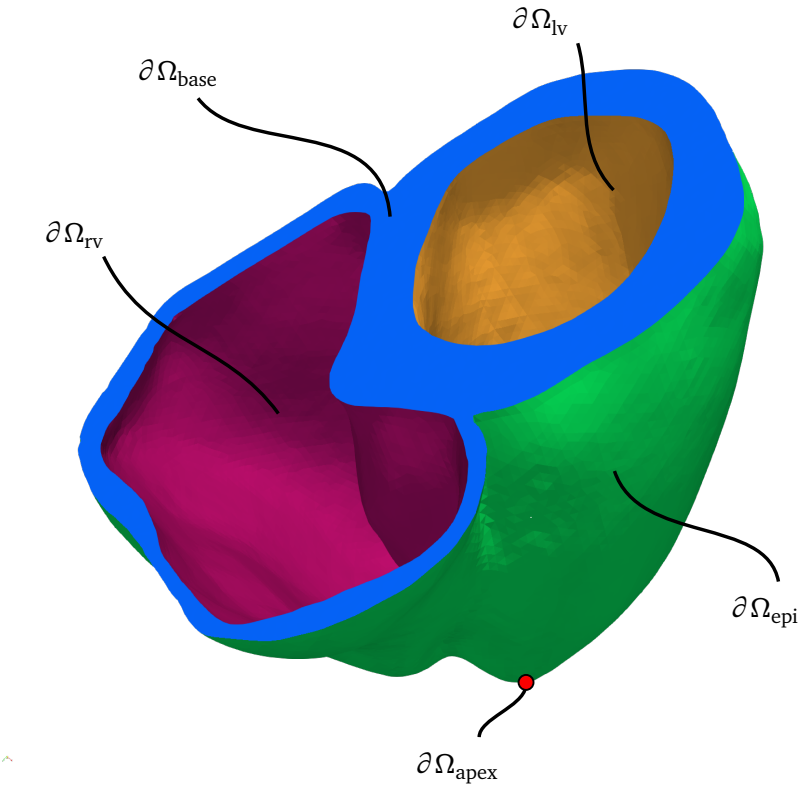


Figure 2.3: The boundary surfaces of interest to the LDRB algorithm marked on a biventricular mesh. The epicardium ($\partial\Omega_{\text{epi}}$) in green, the left ventricle endocardium ($\partial\Omega_{\text{lv}}$) in orange, the right ventricle endocardium ($\partial\Omega_{\text{rv}}$) in purple, and the base ($\partial\Omega_{\text{base}}$) in blue. Also marked is the ventricular apex of the epicardium ($\partial\Omega_{\text{apex}}$).

With the boundary surfaces defined, the next steps of the algorithm concern establishing where every point of interest in the mesh is relative to these boundary surfaces. We need four individual pieces of information:

1. The distance from the epicardium to the left and right ventricle endocardiums, denoted ϕ_{epi}. This is found by solving the Laplace-Dirichlet equation with $\Gamma_1 = \partial\Omega_{\text{epi}}$ and $\Gamma_0 = \partial\Omega_{\text{lv}} \cup \partial\Omega_{\text{rv}}$.
2. The distance from the left ventricle endocardium to the epicardium and right ventricle endocardium, denoted ϕ_{lv}. This is found by setting $\Gamma_1 = \partial\Omega_{\text{lv}}$ and $\Gamma_0 = \partial\Omega_{\text{epi}} \cup \partial\Omega_{\text{rv}}$.
3. The distance from the right ventricle endocardium to the epicardium and left ventricle endocardium, denoted ϕ_{rv}. This is found by setting $\Gamma_1 = \partial\Omega_{\text{rv}}$ and $\Gamma_0 = \partial\Omega_{\text{epi}} \cup \partial\Omega_{\text{lv}}$.
4. The "potential energy" between the apex and the base, denoted ψ_{ab}. This is found by setting $\Gamma_1 = \partial\Omega_{\text{base}}$ and $\Gamma_0 = \partial\Omega_{\text{apex}}$.

The solutions ϕ_{epi}, ϕ_{lv}, ϕ_{rv}, their gradients, and the gradient of ψ_{ab} is all the information that is needed about a particular mesh to be able to compute its fiber orientations. A visualization of the resulting scalar solutions of ϕ_{epi}, ϕ_{lv}, and ϕ_{rv} on a biventricular mesh is shown in Figure 2.4. In addition to the scalar solutions, two sets of input angles are also needed. The first is the angle of the fiber direction F on the epicardium and on the endocardium, denoted α_{epi} and α_{endo}. The second is the angle of the sheet direction, T, on the epicardium and on the endocardium, denoted β_{epi} and β_{endo}. The choice of angles is normally made to fit histological data.

Function 1: `laplace(`Ω, Γ_1, Γ_0`)`

Input: Ω – A biventricular mesh
Input: Γ_1 – Boundary surfaces with boundary condition $u = 1$
Input: Γ_0 – Boundary surfaces with boundary condition $u = 0$
Output: u – Solution of the Laplace-Dirichlet equation

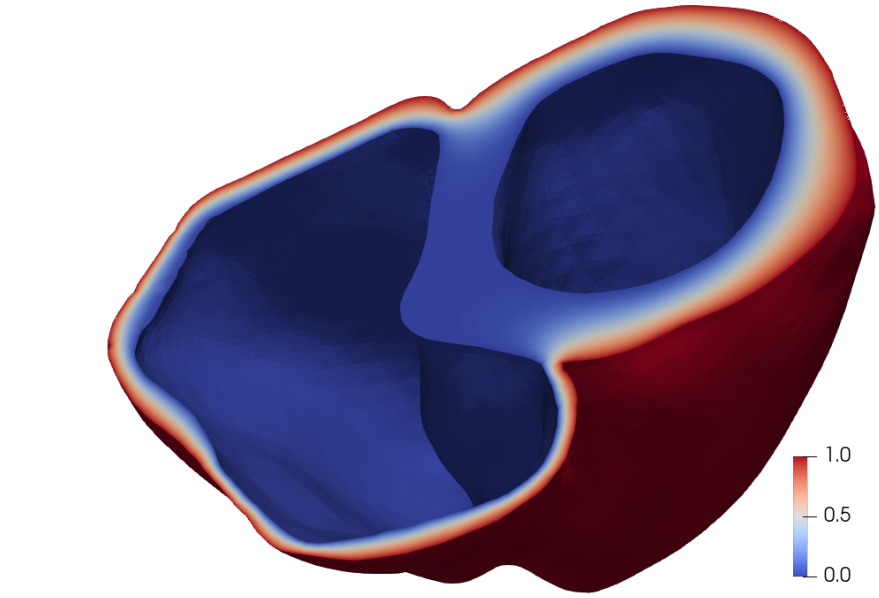
1 Solve

$$\begin{cases} \nabla^2 u = 0 & \text{in } \Omega, \\ u = 1 & \text{on } \Gamma_1, \\ u = 0 & \text{on } \Gamma_0, \\ \nabla u \cdot \mathbf{n} = 0 & \text{on } \partial\Omega - \Gamma_1 - \Gamma_0 \end{cases}$$

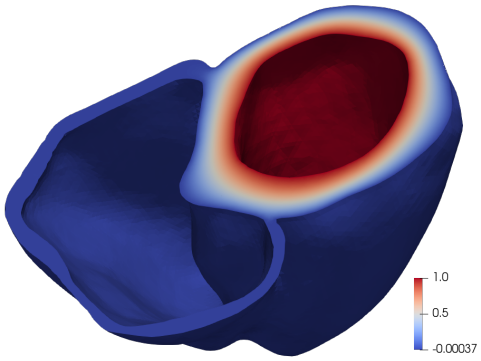
2 **return** u

2.3.1 Creating an axis system

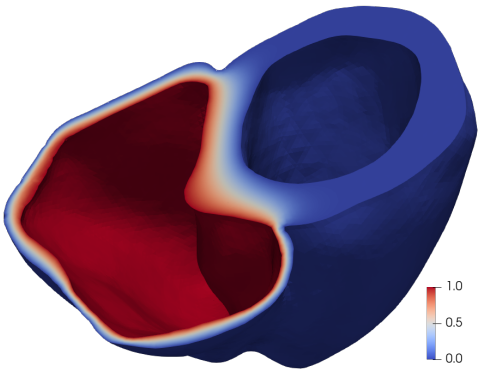
Given the information needed to know the distance to each boundary, the next step is to define an axis system in which we can later define the fiber orientations. The



(a) ϕ_{epi}



(b) ϕ_{lv}



(c) ϕ_{rv}

Figure 2.4: Our visualizations of the scalar field solutions, ϕ_{epi}, ϕ_{lv}, and ϕ_{rv}, of the Laplace-Dirichlet equation on the mesh `heart02` (see Section 4.1). The scalar fields are used in the LDRB algorithm to determine where in the myocardium an arbitrary point of interest is. All scalar fields have values ranging from 0 (blue) to 1 (red). (a) shows ϕ_{epi}, the solution used to determine the distance from the left and right ventricle endocardiums to the epicardium. (b) shows ϕ_{lv}, the solution used to determine the distance from the right ventricle endocardium and the epicardium to the left ventricle endocardium. (c) shows ϕ_{rv}, the solution used to determine the distance from the left ventricle endocardium and the epicardium to the right ventricle endocardium. The data used for this visualization is generated using `cardiac-fibers`, which is presented in Chapter 3. Visualized using ParaView [26].

function `axis`, defined in Function 2, is responsible for creating this axis system. The axes in the resulting right-handed axis system point in the *circumferential*, *apicobasal*, and *transmural* directions. The apicobasal component points, as the name implies, from the apex towards the base, following the shape of the ventricular walls. The transmural component points through the ventricular wall. The circumferential component is orthogonal to the latter two.

Two vectors are needed to construct this axis system. The first vector, $\nabla\psi$, points in the apicobasal direction and is derived by taking the gradient of the solution ψ_{ab}. The second vector, $\nabla\phi$, points in the transmural direction, and is set to the gradient of either ϕ_{epi}, ϕ_{lv}, or ϕ_{rv}, dependent on which ventricular wall we are defining an axis system for.

The first axis, $\hat{e}_1$, of the resulting orthonormal axis system is set to the normalized apicobasal vector. The next axis, $\hat{e}_2$, should point in the general direction of the transmural vector, but it is adjusted so that it is orthogonal to the apicobasal vector by taking the orthogonal projection of the transmural vector onto the plane orthogonal to $\hat{e}_1$. $\hat{e}_2$ is also normalized. The final axis, $\hat{e}_0$, is set to the cross product of the apicobasal vector $\nabla\psi$ and the transmural vector $\nabla\phi$, and ends up pointing in the circumferential direction. The axis system is visualized in Figure 2.5. [2]

Function 2: `axis(`$\nabla\psi, \nabla\phi$`)`

Input: $\nabla\psi$ – Vector pointing in the apicobasal direction.
Input: $\nabla\phi$ – Vector pointing in the transmural direction.
Output: Q – A 3×3 orthonormal matrix representing the axis system for assigning fiber orientations.

1 $\hat{e}_1 = \dfrac{\nabla\psi}{||\nabla\psi||}$
2 $\hat{e}_2 = \dfrac{\nabla\phi - (\hat{e}_1 \cdot \nabla\phi)\hat{e}_1}{||\nabla\phi - (\hat{e}_1 \cdot \nabla\phi)\hat{e}_1||}$
3 $\hat{e}_0 = \hat{e}_1 \times \hat{e}_2$
4 **return** $Q = (\hat{e}_0\ \hat{e}_1\ \hat{e}_2)$

2.3.2 Orienting the axis system

The next step is to take the axis system created by `axis`, and rotate it such that it fits with the angle of longitudinal fiber orientation, α, and the transverse fiber orientation, β. This is handled by the function `orient`, defined in Function 3. The axes of the resulting orthonormal coordinate system $Q' = (F\ S\ T)$ point in the longitudinal direction (F), the sheet normal direction (S), and the transverse

[2] In [6], the second axis of the orthonormal axis system is determined by the expression $\hat{e}_2 = \frac{\nabla\phi - (\hat{e}_0 \cdot \nabla\phi)\hat{e}_0}{||\nabla\phi - (\hat{e}_0 \cdot \nabla\phi)\hat{e}_0||}$. However, $\hat{e}_0$ is undefined at this point. As the goal is to create an axis orthogonal to $\hat{e}_0$ by removing the components of the transmural vector that point in the apicobasal direction, we assume that this is a typo, and that the $\hat{e}_0$ in the expression for $\hat{e}_2$ actually should be $\hat{e}_1$.

direction (T). The rotations applied to produce the longitudinal fiber direction (F) and the transverse direction (T) are visualized in Figure 2.5.

Function 3: `orient`(Q, α, β)

Input: Q – An orthonormal axis system.
Input: α, β – Fiber orientation angles.
Output: Q' – An orthonormal axis system, (F S T).

1 **return** $Q' = Q \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & \sin\beta \\ 0 & -\sin\beta & \cos\beta \end{bmatrix}$

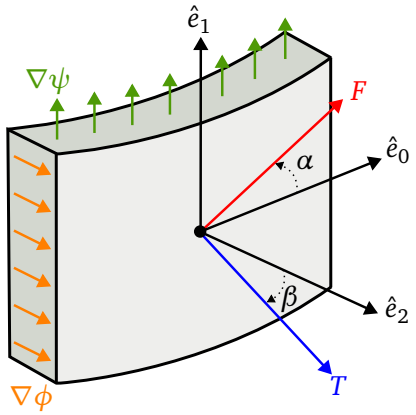


Figure 2.5: The axis system ($\hat{e}_0$, $\hat{e}_1$, $\hat{e}_2$) visualized on the plane of a ventricular wall, along with the longitudinal (F) and transverse (T) orientations produced from applying rotations of α and β. Not shown is the third fiber orientation, the sheet normal direction S, which is orthogonal to F and T.

2.3.3 Interpolating between two axis systems

In some regions of the myocardium, where the point x is close to all three ventricular walls, each of the gradients, $\nabla\phi$, will serve as a transmural vector in its own axis system. This is visualized in Figure 2.6. To find the proper fiber orientation, we need to blend these axis systems, which is done by interpolation. This is performed by the `bislerp` function, defined in Function 4 (page 14).

To interpolate between two axis systems, we first convert them from 3×3 rotation matrices to quaternions. With the axis systems in quaternion form, we perform a spherical linear interpolation (Slerp) [27] between the two. Slerp is used in place of normal linear interpolation, as it guarantees smooth interpolation between two quaternions by rotating with uniform angular velocity. When we interpolate between two quaternions, q_A and q_B, we want to rotate the smallest angle possible. Because the fibers are bidirectional, a quaternion q_M is selected from all the rotations of q_A that are bidirectionally equivalent, such that the chosen

quaternion q_M is the smallest rotation away from q_B. This is taken care of on line 3 in Function 4 (page 14).[3] By bidirectional equivalence, we mean that any rotation of the fiber axis system by 180° around any of its primary axes corresponds to the same fiber orientation. The factor of interpolation is decided by a value $t \in [0, 1]$. After interpolation, the resulting quaternion is converted back to a 3×3 matrix.

Function 4: `bislerp`(Q_A, Q_b, t)

Input: Q_A, Q_B – 3×3 orthogonal matrices.
Input: $t \in [0, 1]$ – Interpolation factor.
Output: Q_{AB} – Resulting matrix of interpolation between Q_A and Q_B.
1 $q_A =$ `rot2quat`(Q_A)
2 $q_B =$ `rot2quat`(Q_B)
3 Find $q_M \in \{\pm q_A, \pm\mathbf{i}\, q_A, \pm\mathbf{j}\, q_A, \pm\mathbf{k}\, q_A\}$ that maximizes $|q_M \cdot q_B|$
4 **return** $Q_{AB} =$ `quat2rot(slerp`$(q_M, q_B, t))$

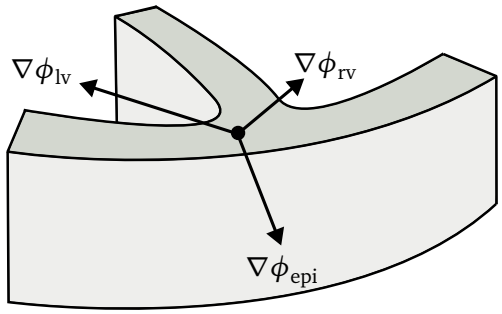


Figure 2.6: Visualization of the gradients in a point that lies in the junction between the septum, LV free wall and RV free wall. Here, there is more than one gradient (transmural vector) contributing, and we have to interpolate between the resulting axis systems to derive the final fiber orientation.

2.3.4 Defining the fibers

With the functions `axis`, `orient`, and `bislerp` defined, we have all the functions that are needed to explain the function `definefibers`, which makes up the last step of Algorithm 1. It is defined in Function 5.

The function loops over every point x in a mesh Ω for which we want to compute the fiber orientations. First, the depth into the septum is determined, denoted by d, which ranges from 0 on the left ventricle endocardium to 1 on

[3] In [6], line 3 of Function 4 is formulated as "Find $q_M \in \{\pm q_A, \pm\mathbf{i} \cdot q_A, \pm\mathbf{j} \cdot q_A, \pm\mathbf{k} \cdot q_A\}$ that maximizes $||q_M \cdot q_B||$", which causes some ambiguity in what the $(\cdot)$-operator refers to. In the cases like "$\pm\mathbf{i} \cdot q_a$", it is to be understood as a Hamilton product. In "$||q_M \cdot q_B||$", however, it refers to a dot product, and the norm $||\ldots||$ is to be interpreted as an absolute value. To get rid of this ambiguity, we have restated the `bislerp` function using the notation established in Section 2.2, where the $(\cdot)$-operator refers to the dot product, and the concatenation $q_1\, q_2$ refers to the Hamilton product.

the right ventricle endocardium. This is used to determine the α and β angles, denoted α_s and β_s, at this depth in the septum. Likewise, the value of ϕ_{epi} in this point is used to determine the same angles for the LV and RV free walls, denoted α_w and β_w.

`axis` is then used to create axis systems for the left ventricle endocardium by using $\nabla\phi_{\text{lv}}$ as the transmural vector, which is given the correct orientation by `orient` using the angles α_s and β_s, producing the matrix Q_{lv}. The same is done for the right ventricle endocardium by using ϕ_{rv} as the transmural vector, producing the matrix Q_{rv}. `bislerp` is then used to determine the final matrix representing the endocardium, by interpolating between Q_{lv} and Q_{rv}, using d as the interpolation factor. The same approach is used to define Q_{epi}, which defines the orientation in relation to the epicardium, by using $\nabla\phi_{\text{epi}}$ as the transmural vector and orienting with α_w and β_w. Finally, the fiber orientations are found by interpolating between Q_{endo} and Q_{epi} by a factor determined by the value of ϕ_{epi}.

Function 5: `definefibers`$(\Omega, \phi_{\text{epi}}, \ldots, \nabla\phi_{\text{epi}}, \ldots, \nabla\psi_{\text{ab}}, \alpha_{\text{endo}}, \alpha_{\text{epi}}, \beta_{\text{endo}}, \beta_{\text{epi}})$

Input: Ω – A biventricular mesh

Input: ϕ_{epi}, ϕ_{lv}, ϕ_{rv} – Scalar fields

Input: $\nabla\phi_{\text{epi}}$, $\nabla\phi_{\text{lv}}$, $\nabla\phi_{\text{rv}}$, $\nabla\psi_{\text{ab}}$ – Vector fields

Input: α_{endo}, α_{epi}, β_{endo}, β_{epi} – Fiber (α) and sheet (β) angles

Output: F, S, T – Fiber orientations

for *each point* x *in* Ω **do**

$d = \frac{\phi_{\text{rv}}(x)}{\phi_{\text{lv}}(x)+\phi_{\text{rv}}(x)}$ {Septum depth }

$\alpha_s = \alpha_{\text{endo}}(1-d) - \alpha_{\text{endo}}d$ {Fiber angle in septum}

$\beta_s = \beta_{\text{endo}}(1-d) - \beta_{\text{endo}}d$ {Sheet angle in septum}

$\alpha_w = \alpha_{\text{endo}}(1-\phi_{\text{epi}}(x)) + \alpha_{\text{epi}}\phi_{\text{epi}}(x)$ {Fiber angle in wall}

$\beta_w = \beta_{\text{endo}}(1-\phi_{\text{epi}}(x)) + \beta_{\text{epi}}\phi_{\text{epi}}(x)$ {Sheet angle in wall}

$Q_{\text{lv}} = \texttt{orient}(\texttt{axis}(\nabla\psi_{\text{ab}}(x),\ -\nabla\phi_{\text{lv}}(x)),\ \alpha_s,\ \beta_s)$

$Q_{\text{rv}} = \texttt{orient}(\texttt{axis}(\nabla\psi_{\text{ab}}(x),\ \nabla\phi_{\text{rv}}(x)),\ \alpha_s,\ \beta_s)$

$Q_{\text{endo}} = \texttt{bislerp}(Q_{\text{lv}},\ Q_{\text{rv}},\ d)$

$Q_{\text{epi}} = \texttt{orient}(\texttt{axis}(\nabla\psi_{\text{ab}}(x),\ \nabla\phi_{\text{epi}}(x)),\ \alpha_w,\ \beta_w)$

$(F(x)\ S(x)\ T(x)) = \texttt{bislerp}(Q_{\text{endo}},\ Q_{\text{epi}},\ \phi_{\text{epi}}(x))$

end

2.4 The Finite Element Method

In this section, we will go through some key aspects of the finite element method, which we will be using for discretization of the Laplace-Dirichlet equations when

implementing the LDRB algorithm. The details of how it is used are left to Chapter 3. We will then describe the MFEM software library, which will be used to program the FEM-related parts. For an in-depth explanation of the finite element method, see for instance [28] or [29].

Finite element methods are one of the most important and commonly used tools for solving partial differential equations. It is especially powerful when it is applied to problems in which the domain is an unstructured mesh. In FEM, partial differential equations are restated to what is known as a *weak* formulation. We will explain this by deriving the weak formulation of the Laplace-Dirichlet equation presented in Function 1. The following explanation is an adaption of [11, Section 2].

To simplify, we assume that the Dirichlet boundary conditions are zero on both Γ_0 and Γ_1, and we assume a homogeneous Neumann boundary condition, i.e., $\nabla u \cdot \mathbf{n} = 0$, on $\partial\Omega - \Gamma_0 - \Gamma_1$. The implications of having non-zero Dirichlet boundary conditions will be explained at the end. The weak formulation of the Laplace-Dirichlet equation is obtained by multiplying Equation 2.1 by a *test function* v, and integrating over the domain Ω:

$$\int_\Omega \nabla^2 u \cdot v \, dx = \int_\Omega 0 \cdot v \, dx = 0. \tag{2.2}$$

Integration by parts gives

$$\int_\Omega \nabla u \cdot \nabla v dx - \int_{\Gamma_0} (\nabla u \cdot \mathbf{n}) v dS - \int_{\Gamma_1} (\nabla u \cdot \mathbf{n}) v dS - \int_{\partial\Omega - \Gamma_1 - \Gamma_0} (\nabla u \cdot \mathbf{n}) \cdot v dS = 0, \tag{2.3}$$

where Γ_0 and Γ_1 are the boundaries with boundary condition $u = 0$, and $\mathbf{n}$ is the outward normal on the boundary $\partial\Omega$. By imposing $v = 0$ on $\Gamma_0 \cup \Gamma_1$, and knowing that $\nabla u \cdot \mathbf{n} = 0$ on $\partial\Omega - \Gamma_1 - \Gamma_0$, the related terms disappear and we end up with the weakly formulated boundary value problem

$$\int_\Omega \nabla u \cdot \nabla v \, dx = 0. \tag{2.4}$$

The goal is to find $u \in V$ such that Equation 2.4 holds for all test functions $v \in V$, where V is the space of admissible functions. In this example, V consists of functions that fulfill the requirements that $v = 0$ on $\Gamma_0 \cup \Gamma_1$, and have well-defined first order derivatives. This turns out to be what is known as the H^1 space [28]. The space of admissible functions can then be defined as

$$V = \left\{ v \in H^1(\Omega), \; v = 0 \text{ on } \Gamma_0 \cup \Gamma_1 \right\}. \tag{2.5}$$

The weak formulation in Equation 2.4 can further be rewritten as a combination of a *bilinear form*,

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v, \tag{2.6}$$

and the *linear form*

$$l(v) = 0. \tag{2.7}$$

The problem can then be described as finding $u \in V$ such that

$$a(u, v) = l(v) \; \forall v \in V. \tag{2.8}$$

The weak formulation in Equation 2.4 is continuous, so to be able to approximate the solution on a computer we approximate the solution by the discretization

$$u \approx u_h = \sum_j^N c_j \phi_j.$$

Here, ϕ_j are the basis functions of the discrete subspace $V_h \subset V$, called *trial basis functions*, and c_j are scalar unknowns, called *degrees of freedom* (DoFs). By substituting u_h into Equation 2.4, we get

$$\sum_j c_j \int_\Omega \nabla \phi_j \cdot \nabla v = 0, \; \forall v \in V. \tag{2.9}$$

By selecting test functions $\phi_i \in V_h$, we get the final finite-dimensional problem

$$\sum_j c_j \int_\Omega \nabla \phi_j \cdot \nabla \phi_i = 0, \; \forall \phi_i \in V_h. \tag{2.10}$$

The finite-dimensional problem can then be formulated as a discrete linear algebra problem, $Ax = b$, by setting

$$\begin{aligned} A_{ij} &= \int_\Omega \nabla \phi_j \cdot \nabla \phi_i, \\ b_i &= 0, \\ x_j &= c_j. \end{aligned}$$

A common method for enforcing non-zero Dirichlet boundary conditions, as we actually have on Γ_1, is to modify the linear system so that the related components of the right-hand side b have a given value, in our case one, rather than zero, and to modify the corresponding row of the matrix A to be one on the diagonal and zero on the off-diagonal.

2.4.1 MFEM

MFEM [11] is a C++ library for finite element methods. According to its developers, the distinguishing features of MFEM are "massively parallel scalability, HPC efficiency, support for arbitrary high-order finite elements, generality in mesh type and discretization methods, and the focus on maintaining a clean, lightweight

code base" [11]. The serial version of MFEM only relies on a (modern) C++ compiler. The parallelized version minimally depends on an MPI C++ compiler, as well as the *hypre* [30] and Metis [31] libraries. It has support for several mesh formats, including VTK [32], Gmsh [33], and a native MFEM format. MFEM supports a wide range of mesh elements, including segments, triangles, quadrilaterals, tetrahedra, hexahedra, prisms, and pyramids. Parallelism is mainly targeted through MPI, and automatic parallel mesh partitioning is supported through Metis.

Finite element model

MFEM has a series of objects and methods that use the concepts of the finite element method presented at the beginning of this section, which ties together to produce a linear system of equations. The overall structure is as follows:

- A `Mesh` object is used to hold the elements that make up the domain Ω, and the physical coordinates of all the vertices.
- A `FiniteElementCollection` object is used to define the type of basis functions to use on the reference element.
- A `FiniteElementSpace` object gives the mapping between the reference element and each of the physical elements from the mesh.
- Given a finite element space, a `LinearForm` object is used to describe the right-hand side linear form, $l(\cdot)$.
- Similarly, a `BilinearForm` object is used to describe the left-hand side bilinear form, $a(\cdot,\cdot)$.
- Transitioning to the linear algebra side, a `GridFunction` is used to hold the initial guess of the solution vector x with the known essential boundary conditions.
- Finally, the linear system $Ax = b$ can be formed with the `FormLinearSystem` method of `BilinearForm`, which produces the left-hand side matrix A and the right-hand side vector b, which can be passed to a solver along with the `GridFunction`.

2.5 GPU programming

A graphics processing unit is a hardware accelerator designed for high bandwidth and instruction throughput. Compared to a CPU, the transistors in a GPU are mainly used for data processing, rather than complex layers of caches and flow control. Instead, GPUs try to hide memory latency by using a high number of threads. As a result of this, it has a much higher parallel performance than that of a traditional CPU, but also a much lower sequential performance. This kind of hardware architecture fits well with computations where the same operation is applied to a large number of individual elements, as takes place in the finite element method.

There are several programming models used to program GPUs, some of which are able to run on hardware from different vendors, such as OpenCL [34], and

some of which are targeted for a single vendor, such as CUDA. The most commonly used programming models for GPUs in scientific computing are CUDA [35] for NVIDIA hardware and HIP [36] for AMD hardware.[4]

In the HIP and CUDA programming models, a parallel workload is divided into fine-grained tasks, each of which is assigned to a *thread*. The workload is defined by a small program called a *kernel*, which is formulated as a C++ or Fortran function that has a set of inputs and outputs. A group of threads then executes the kernel in a lock-step fashion, with each thread working on some individual or shared piece of data. This approach to parallelism is often referred to as single instruction, multiple threads (SIMT). SIMT allows a combination of both the multithreading and single instruction, multiple data (SIMD) execution models.

When GPUs are used in general-purpose, heterogeneous computations, we refer to the CPU as the *host* and the GPU as the *device*. The two have separate memory regions, which are referred to as the *host memory* and *device memory*, respectively. If we want to offload some computation to the device, and the data to be processed resides on the host, we first have to allocate space for the data in device memory. Then, the data is transferred to the device. When the data has arrived, a kernel can be launched to process the data on the device. When the data is processed, the result can be transferred back to the host, or be kept on the device for further processing.

GPU programming in MFEM

As well as serving as an abstraction layer that allows for easily formulating finite element problems in code, the MFEM library also has some well-established support for executing parts of these problems on GPUs.

MFEM supports GPU execution mainly through two interfaces: an internal memory manager and the `MFEM_FORALL` macro. The main goal of these two interfaces is to enable performance portability by hiding device-specific code. The internal memory manager wraps all pointers of type `T*` in a `Memory<T>` object, which handles both host and device side pointers, as well as memory allocation and data synchronization. If the memory object is intended to reside on the device, the `void UseDevice(bool)` method can be called on the object. To access the underlying type, `T*`, the `Read()`, `Write()` and `ReadWrite()` methods provide a read-only pointer, a write-only pointer, and a read-and-write pointer to the underlying memory, respectively. The location of the returned pointers (i.e. on the host or device) will depend on how `UseDevice` was called in advance (`true` for device, `false` for host).

Given a pointer to the underlying type, the `MFEM_FORALL` macro serves as a wrapper around a regular `for`-loop. If the pointers being manipulated reside on the host, the macro expands to a `for`-loop. That is, `MFEM_FORALL(i, n,{ ... });` expands to `for (int i = 0; i < n; i++) {...}`. But, if the pointers reside

[4]HIP code can also be compiled for NVIDIA hardware through a CUDA backend.

Code listing 2.1: Example use of the `MFEM_FORALL` macro used for offloading to GPU. Adapted from [37].

```
Vector u;
Vector v;
// ...
auto u_data = u.Read();      // Express the intent to read u
auto v_data = v.ReadWrite(); // Express the intent to read and write v
MFEM_FORALL(i, u.Size(),     // Abstract the loop: for(int i=0; i<u.Size(); i++)
{
   v_data[i] *= u_data[i];   // This block of code is executed on the chosen device
});
```

on the device, the macro will expand to either a HIP or CUDA kernel which is launched on the device with `n` threads. The `MFEM_FORALL` macro also allows for thread-level parallelized execution on the host through OpenMP. An example use pattern is shown in Code listing 2.1. The macro expects a lambda, i.e., an anonymous function, as its third argument, but it is also possible to call a function directly by marking it with either the `MFEM_HOST_DEVICE` or `MFEM_DEVICE` macros, which makes sure that said functions are compiled for host and device execution or device execution only, by expanding to the `__host__` `__device__` and `__device__` function attributes. These attributes tell the compiler whether a function should be compiled to host or device execution, or both.

The upside of this approach to GPU execution is that these two interfaces make it easy for MFEM users to introduce GPU execution into their programs, and allows for highly portable code between different hardware configurations. For example, if some piece of code is developed to target NVIDIA GPUs, a port of said code to target AMD GPUs should be as simple as building the MFEM library with HIP support, and changing the compiler from `nvcc` to `hipcc`. If MFEM is built without any GPU support at all, the code still works and will be executed on the host.

The downside of this approach is that it hides all the data transfers between the host and device in the internal memory manager. Since such data transfers are expensive, they are often carefully and explicitly managed by the developer. Making the memory allocation, transfers, and deallocation opaque to the developer may therefore hide bottlenecks.

In addition to supporting both CUDA and HIP execution natively, MFEM also offers an interface for the *hypre* [30] software library, which delivers high-performance preconditioners and solvers for linear systems, some of which are implemented with GPU support.

Chapter 3

`cardiac-fibers`: LDRB on GPU

In this chapter, we describe `cardiac-fibers`, our GPU-enabled implementation of the LDRB algorithm developed as a part of this thesis. One of the goals of `cardiac-fibers` is to improve upon the existing `ldrb` [10] implementation, to be able to process larger meshes and to use GPUs in an attempt to do it faster. As alluded to in Chapter 2, we utilize the finite element method and use the MFEM finite element library to do much of the heavy lifting. The full source code of `cardiac-fibers` is available in a repository on the author's GitHub [38].

This chapter is structured as follows. In Section 3.1 we will explain what the input meshes to the program look like. In Section 3.2, we will explain some considerations that have to be made about the space in which the computed fibers live, and formulate the approach taken from input mesh to output fibers. In Section 3.3, we look at how we get the scalar field solutions to the Laplace-Dirichlet equations. In Section 3.4, we explain how the gradients of the scalar fields are computed. In Section 3.5, we look at how the space considerations introduced in Section 3.2 are applied to the scalar and vector fields. In Section 3.6 we describe how the functions that make up the actual fiber computation are implemented. Finally, in Section 3.7, we describe the heuristic we have developed that allows for a more robust fiber definition.

Note that this chapter contains a number of listings that are adapted from the implementation code, but that are slightly altered for clarity or brevity. Refer to the source code for the actual implementation [38].

3.1 Defining the input meshes

The finite element mesh generator software Gmsh [33] is used to generate the tetrahedral meshes. The legacy Gmsh binary format 2.2 is used, as it is the only Gmsh file format that is supported by MFEM [11].

The input mesh for the program contains all the vertices and tetrahedral elements that form either a single- or biventricular mesh, Ω. The boundary surfaces base ($\partial\Omega_{\text{base}}$), left ventricle endocardium ($\partial\Omega_{\text{lv}}$), right ventricle endocardium

($\partial\Omega_{\text{rv}}$) and epicardium ($\partial\Omega_{\text{epi}}$) are given an ID in the `$PhysicalNames` section of the mesh file, and all boundary elements (triangles) that are part of one of the boundary surfaces are marked with the respective ID. In the case of a single ventricle mesh, the ID of the right ventricle endocardium is left unset.

With a mesh in this format, we have covered the first step of the LDRB algorithm (Algorithm 1), namely the definition of the boundary surfaces. Note that the apex ($\partial\Omega_{\text{apex}}$) is not defined in the input mesh. How we handle the apex is explained in Subsection 3.3.1.

Although MFEM supports a wide variety of element types, we limit ourselves to only considering tetrahedrons, first and foremost because the irregular structure of the biventricular heart meshes are best represented by tetrahedral elements, but also because it allows for optimizations that will become evident later in this chapter.

3.2 Selecting a solution space and outlining the approach

Before diving into the next steps of the LDRB algorithm (Algorithm 1), which involves solving the Laplace-Dirichlet equations, we have to make some choices about what sort of space we want the final fiber orientations to be defined in. When this is decided, we can outline the rest of the approach.

As described in Section 2.4, the weak formulation of the Laplace-Dirichlet equation has its trial basis functions, ϕ_j, and the degrees of freedom, c_j, in a discrete subspace of H^1. In our implementation, we will only be considering first-order linear basis functions, i.e., $\phi_j \in \mathbb{P}_1$. For piecewise linear elements in H^1, the degrees of freedom are located in the nodes (vertices) of the domain Ω. If we were to apply the LDRB algorithm directly on the scalar field solutions to the Laplace-Dirichlet equation, we would end up with fibers that are also defined in the vertices of Ω. However, there are many use cases for the fiber orientations where we would much rather have them be defined *per-element* than *per-vertex*. One of the goals of the implementation was, therefore, to be able to generate fibers both per-element and per-vertex.

If we want the fibers to be defined per-element, we have to make sure that both the scalar fields and the gradient vector fields are defined per-element. In the case of the gradients, this simplifies things because the gradients of the H^1 space are, by definition, defined in the L^2 space, a space in which the degrees of freedom are located in the center of each element. The gradient calculation is explained in Section 3.4. The scalar fields, however, have to be evaluated at the center of each element. This can be viewed as projecting each field onto a space spanned by piecewise constant elements. This is further explained in Subsection 3.5.1.

If we want the fibers to be defined per-vertex, the scalar fields are already defined in the correct space. However, as we just explained, the gradients are not. When the gradients have been calculated, they then have to be interpolated from L^2 to H^1. This is further explained in Subsection 3.5.2.

With a solution space selected, we can define all the steps that need to take place from when the input mesh is loaded to the resulting fiber orientations. For each of the Laplace-Dirichlet equations, ϕ_{epi}, ϕ_{lv}, ϕ_{rv}, and ψ_{ab}, we first have to assemble the linear system of equations corresponding to the weak formulation, as described in Section 2.4. We then have to solve the linear system of equations to produce a scalar field. Next, we calculate the gradients of the scalar fields. If we choose to have the final fibers on a per-element basis, we project the scalar fields to L^2. If we instead want the fiber per-vertex, we interpolate the gradients to H^1. Finally, we compute the fiber orientations using `definefibers` (Function 5, page 15). The execution flow is outlined in Figure 3.1

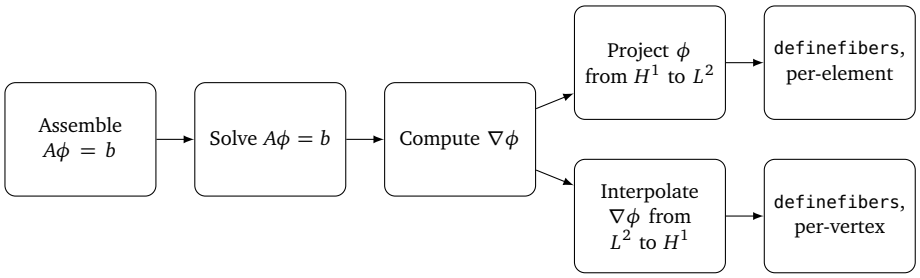


Figure 3.1: The execution flow when generating fibers either on a per-element or a per-vertex basis. For clarity, only one of the four solutions is considered, denoted by ϕ. In the case of ψ_{ab}, the projection from H^1 to L^2 is not necessary.

Out of all the steps presented in Figure 3.1, two were initially targeted for offloading to GPU. The first is the solving of the linear systems, which is offloaded by using a GPU-enabled solver. This is discussed in Subsection 3.3.2. The second is the fiber computation performed by the `definefibers` function, the implementation of which is discussed is Section 3.6. If these two are the only parts offloaded to the GPU, the execution and transfer pattern between the host and the device will be as shown in Figure 3.2, in the case where fibers are computed per-element.

3.3 Solving the Laplace-Dirichlet equations

With the input mesh and the surfaces defined and the solution space taken into consideration, we can start solving the Laplace-Dirichlet equations. As Figure 3.1 shows, getting the solutions of the Laplace-Dirichlet equations is a two-step process. First, we have to construct a linear system of equations; then we have to solve it.

3.3.1 Constructing the linear systems of equations

For each of the four Laplace-Dirichlet equations we want the solution of, we have to do the following:

1. Determine the list of *essential* degrees of freedom.

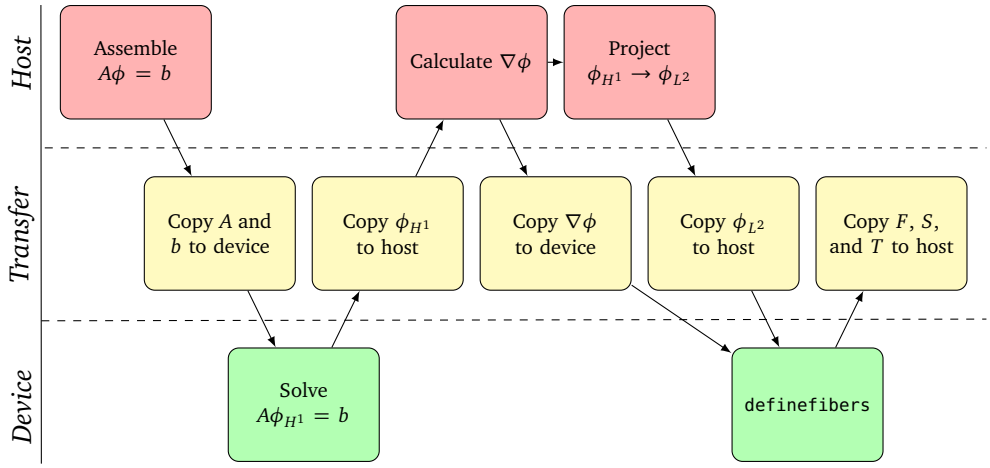


Figure 3.2: The execution and data transfer pattern when calculating fiber orientations on a per-element basis and offloading the solving of linear systems and fiber calculation to GPU. For clarity, only one of the four solutions is considered, denoted by ϕ. In the case of ψ_{ab}, the projection from H^1 to L^2 is not necessary.

2. Assemble the right-hand side linear form, $l(\cdot)$.
3. Assemble the left-hand side bilinear form, $a(\cdot,\cdot)$.
4. Set the values of the essential DoFs as an initial guess to the solution, x.
5. Form the linear system of equations, $Ax = b$.

The first step is to determine what kind of basis functions are to be used when mapping to a reference element. As we just saw, the weak form of the Laplace equation has solutions in the H^1 space, so that is the space we choose. The next step is to define a mapping between each element in the mesh to the reference element. With the space and the mapping defined, we have to figure out which degrees of freedom on the boundary are *essential*, i.e. have a known value. This is done by selecting the DoFs that, in our case, are part of either Γ_0 or Γ_1. Next, we define the linear and bilinear forms, as per Equation 2.7 and Equation 2.8. The initial guess to the solution, x, is then defined, which is of the length defined by the number of essential DoFs. The known values $x = 0$ on Γ_0 and $x = 1$ on Γ_1 are set in the respective DoFs. Given the bilinear and linear forms, as well as the vector x, we can finally form the linear system $Ax = b$. Some example code showing how this is done in MFEM, exemplified for creating the linear system to solve ϕ_{epi}, is shown in Code listing A.1

Handling of the boundary conditions for ψ_{ab}

The last set of boundary conditions is used to find the solution of the Laplace-Dirichlet equation between the base and the apex, ψ_{ab}. This case is different from the previous three in that the apex is a single point on the epicardium surface, rather than a disjoint surface. This makes the enforcement of the boundary con-

ditions a little different.

The first step is to find the actual apex, which is loosely defined to be the point on the epicardium that is the furthest from the base. Different approaches can be taken to find this apex. `ldrb`[10] takes the approach of solving a Laplace-Dirichlet equation similar to the ones used in Algorithm 1, namely

$$\begin{cases} \nabla^2 u = 1 & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega_{\text{base}}, \end{cases} \tag{3.1}$$

to create a heuristic for where the apex is. The heuristic has the apex in the degree of freedom (vertex) with maximal value in the solution to Equation 3.1. The upside of this approach is that no preprocessing has to take place to find the apex. The downside is that an additional Laplace-Dirichlet equation has to be solved, which is costly, and the location of the apex can not be manually selected.

The approach taken in our implementation is to take in a user-specified apex coordinate and then find the vertex closest to said coordinate. In practice, this is done by having each MPI rank go through its submesh and find its local apex candidate, which is the vertex in its submesh which has the smallest Euclidean distance from the prescribed apex. Then the global apex is found by selecting the local apex candidate with the smallest Euclidean distance.

MFEM does not directly support enforcing boundary conditions in a single vertex, so some fiddling with the DoFs has to happen to make it work. Since the Laplace-Dirichlet equation is being solved in H^1, we know that there is a DoF for each vertex, and hence there is a DoF in the apex vertex. The rank which has the apex in its submesh does a lookup to find the local id of the apex DoF, which is then added to the list of essential DoFs. The value in this DoF is then set to 0. Some example code showing how this is done is shown in Code listing A.2.

Reusing the linear and bilinear forms

Since the left-hand sides and right-hand sides which are formed for ϕ_{epi}, ϕ_{lv} and ϕ_{rv} are based on the same list of essential DoFs, namely from the boundary surfaces $\partial\Omega_{\text{epi}}$, $\partial\Omega_{\text{lv}}$, and $\partial\Omega_{\text{rv}}$, these actually end up being identical for all three linear systems. The only thing that changes between the three is what the value on each surface is set to, but this does not change which DoFs end up in the resulting linear system. This means the linear form, l, and bilinear form, a, can be reused for the three, and the `Assemble()` method only has to be called once for each form.

As the essential boundaries are different in ψ_{ab}, the list essential of essential DoFs change as well. This means that the assembled linear form and bilinear form can not be reused, and they both have to be reassembled with the new list of essential DoFs.

By only having to assemble two sets of bilinear and linear forms instead of four, we effectively reduce the work and the time spent doing assembly by a factor of two.

3.3.2 Solving the linear systems

When solving a sparse linear system of equations, $Ax = b$, there are two main approaches: A direct solver or an iterative solver. When using an iterative solver, it is common to also make use of a *preconditioner*. A preconditioner performs either an implicit or explicit modification of the linear system, which makes it easier to solve with an iterative method afterward [39].

In [6], the method used for solving the linear systems is the conjugate gradient method (CG) [40], preconditioned with block-Jacobi method and incomplete factorization (ILU(0)) to solve each block.

In `ldrb`, the Laplace-Dirichlet equations are solved using SuperLU [41], specifically the SuperLU_DIST version for distributed memory systems [42]. SuperLU is a direct solver, which solves a sparse linear system $Ax = b$ using Gaussian elimination with partial pivoting (GEPP), and is made primarily to solve sparse unsymmetric linear systems.

Like the authors of [6], we choose to use the conjugate gradient method (CG) as the solver of choice for the linear systems. The conjugate gradient method is an iterative algorithm that is designed to solve linear systems where the left-hand side matrix A is symmetric, positive definite (SPD). It is part of a family of methods known as *Krylov subspace* methods. We also choose to employ a preconditioner, but our preconditioner of choice is algebraic multigrid (AMG) [43, 44]. Using a combination of CG and AMG, we can expect that the number of iterations needed to solve the linear systems is independent of the problem size [44]. Specifically, we choose to use the preconditioned conjugate gradient (PCG) solver implemented in the *hypre* library. We combine the PCG solver with *hypre*'s implementation of the algebraic multigrid (AMG) method [43, 44], BoomerAMG, as the preconditioner. The choice of this combination of solver and preconditioner is twofold. First, the algebraic multigrid method ensures that the number of iterations required in the solver remains more or less constant as the problem size increases. Second, both PCG and BoomerAMG have GPU support, meaning that the whole solve step can be offloaded to GPU. Using this combination of solver and preconditioner for solving linear systems on GPU has been shown to perform well, both for NVIDIA and AMD hardware [45, 46].

3.4 Calculating the gradients of the scalar fields

Given the resulting scalar field solutions to the Laplace-Dirichlet equations, the next step is to find the gradients. In this section, we look at how the gradients are calculated, and how the calculations are offloaded to the GPU.

We could use functionality built into MFEM to calculate the gradients, as shown in Code listing A.3, but we run into the issue of having to perform the computations on the host, as shown in Figure 3.2. Instead, we can use some knowledge about the finite element space we are working with, combined with the fact that we are considering strictly tetrahedral meshes, to calculate the gradients using a

simple formula instead.

We know that the gradient of a scalar field in H^1 is, by definition, a vector field in L^2. In L^2, it is possible to use a single degree of freedom for each element, placed in the centroid of the element. As mentioned in Section 3.2, we have chosen to use first-order basis functions, $\phi_i \in \mathbb{P}_1$, and we, therefore, know that the basis functions in the gradient space have to be constant. So, there has to be one gradient per element, and the value is constant over the whole element. This also means that we are able to calculate the value of the gradient in each element purely based on the scalar field values in each of its vertices. We have also restricted the input to only consist of tetrahedral meshes.

With these observations and restrictions in control, we choose to use the method of per-cell linear estimation (PCE) of the gradient in a tetrahedron. This method is outlined in Equation 2 in Section 3.1 of [47]. Given vertices $p_{i,0}$, $p_{i,1}$, $p_{i,2}$ and $p_{i,3}$ that form a tetrahedron T_i, with known, constant values at each vertex, $u_{i,0}$, $u_{i,1}$, $u_{i,2}$ and $u_{i,3}$, the gradient can be shown to be

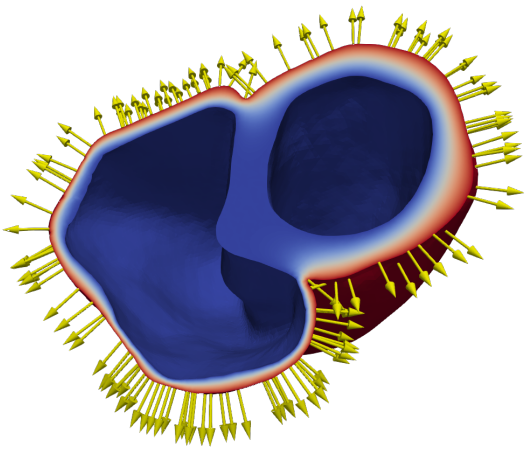
$$\begin{aligned}\nabla u_{T_i} =&(u_{i,1}-u_{i,0})\frac{(p_{i,0}-p_{i,2})\times(p_{i,3}-p_{i,2})}{6V_{T_i}}\\ &+(u_{i,2}-u_{i,0})\frac{(p_{i,0}-p_{i,3})\times(p_{i,1}-p_{i,3})}{6V_{T_i}}\\ &+(u_{i,3}-u_{i,0})\frac{(p_{i,2}-p_{i,0})\times(p_{i,1}-p_{i,0})}{6V_{T_i}},\end{aligned} \tag{3.2}$$

where V_{T_i} is the volume of T_i.[1]

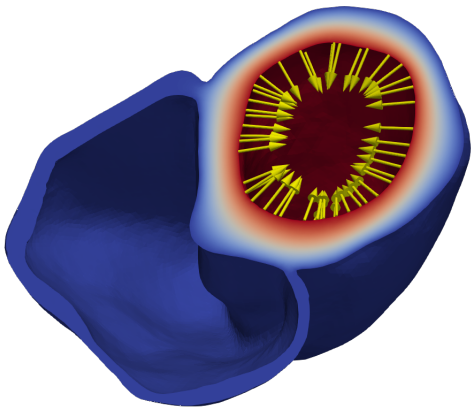
The formula for the gradient requires the H^1 DoFs, from which we get $u_{i,j}$, and the L^2 DoF, where we store the result ∇u_{T_i}, for each element. The function `FiniteElementSpace::GetElementToDofTable()` is used to retrieve a mapping from the index i of each element T_i in a given space, to the DoFs associated with said element. Additionally, the physical coordinates of each vertex $p_{i,j}$ of T_i are needed. For this, we use `Mesh::GetVertices()`, which gives an array of the coordinates of every vertex in the input mesh. Both the element-to-DoF tables and vertices are transferred to the GPU. A straightforward kernel that computes the gradient according to Equation 3.2 is then used, with one thread assigned per element. The implementation is shown in Code listing 3.1. Note that the code uses the same notation as in [47, Equation 2], where the vertices of the tetrahedron are denoted by v_i, v_j, v_k, and v_k, with scalar values f_i, f_j, f_k, and f_h. A visualization of the resulting gradients is shown in Figure 3.3. Each gradient is visualized as a vector field (in yellow), on top of the scalar field it is a gradient of. For clarity, only a subset of the vector field is shown, and the vectors are normalized.

independent and no communication between ranks is necessary.

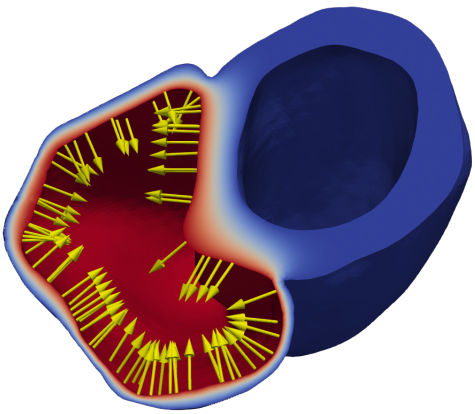
[1] In [47, Equation 2], the denominator is set to $2V$ rather than $6V$. When using said formula, the gradients produced were off by a factor of three when compared to the gradients produced by MFEM (as calculated with the approach shown in Code listing A.3). After our correspondence with the authors of [47], it was discovered that the original formula was incorrect and that the denominator should in fact be $6V$. A correction has since been published in [48].



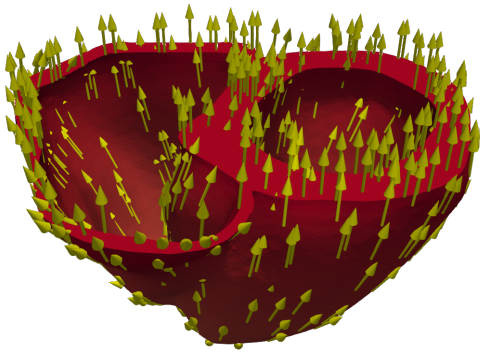
(a) $\nabla\phi_{\text{epi}}$



(b) $\nabla\phi_{\text{lv}}$



(c) $\nabla\phi_{\text{rv}}$



(d) $\nabla\psi_{\text{ab}}$

Figure 3.3: Visualization of gradients of the Laplace-Dirichlet scalar fields, $\nabla\phi_{\text{epi}}$, $\nabla\phi_{\text{lv}}$, $\nabla\phi_{\text{rv}}$, and $\nabla\psi_{\text{ab}}$, on the mesh `heart02`. The gradients are used to create axis systems for defining fiber orientations in the LDRB algorithm. Each gradient is visualized as a vector field, in yellow, on top of the scalar field for which it is a gradient. For clarity, only a subset of each vector field is shown, and all vectors are normalized. The data is generated by using the `compute_gradient` function in `cardiac-fibers`, shown in Code listing 3.1, and visualized using ParaView [26].

Code listing 3.1: GPU-enabled kernel for gradient computation.

```
void compute_gradient(
    double *gradient,           // Gradient in L2
    const double *laplace,      // Laplace-Dirichlet solution in H1
    const double *vert,         // Vertex coordinates
    const int ne, const int nv, // Number of elements and  vertices
    const int *h1_table_col,    // H1 element-to-DoF table columns
    const int *h1_table_row,    // H1 element-to-DoF table rows
    const int *l2_table_col,    // L2 element-to-DoF table columns
    const int *l2_table_row)    // L2 element-to-DoF table rows
{
    mfem::MFEM_FORALL(i, ne, {
        const int *h1_dofs = &h1_table_row[h1_table_col[i]];

        // The vertices are in SoA-format: x0 x1 ... xn y0 y1 ... yn z0 z1 ... zn
        vec3 v_i, v_j, v_k, v_h;
        v_j = {vert[0*nv+h1_dofs[0]], vert[1*nv+h1_dofs[0]], vert[2*nv+h1_dofs[0]]};
        v_k = {vert[0*nv+h1_dofs[1]], vert[1*nv+h1_dofs[1]], vert[2*nv+h1_dofs[1]]};
        v_h = {vert[0*nv+h1_dofs[2]], vert[1*nv+h1_dofs[2]], vert[2*nv+h1_dofs[2]]};
        v_i = {vert[0*nv+h1_dofs[3]], vert[1*nv+h1_dofs[3]], vert[2*nv+h1_dofs[3]]};

        // Get the values of the scalar H1 DoFs at each vertex of the tetrahedron.
        const double f_j = laplace[h1_dofs[0]];
        const double f_k = laplace[h1_dofs[1]];
        const double f_h = laplace[h1_dofs[2]];
        const double f_i = laplace[h1_dofs[3]];

        // Get all the needed vertex-to-vertex vectors.
        vec3 v_ik = v_i - v_k; vec3 v_hk = v_h - v_k;
        vec3 v_ih = v_i - v_h; vec3 v_jh = v_j - v_h;
        vec3 v_ki = v_k - v_i; vec3 v_ji = v_j - v_i;

        // Calculate the three needed cross-products.
        vec3 v_ik_x_v_hk; vec3_cross(v_ik_x_v_hk, v_ik, v_hk);
        vec3 v_ih_x_v_jh; vec3_cross(v_ih_x_v_jh, v_ih, v_jh);
        vec3 v_ki_x_v_ji; vec3_cross(v_ki_x_v_ji, v_ki, v_ji);

        // For the volume we need one more vertex-to-vertex vector, v_hi.
        // We divide the final gradient by six times the volume, so no need to
        // do the division by six to get the real volume.
        vec3 v_hi = v_h - v_i;
        const double six_vol = abs(vec3_dot(v_hi, v_ki_x_v_ji));

        // Scale each component by the difference in scalar values in the
        // vertices.
        v_ik_x_v_hk *= (f_j - f_i);
        v_ih_x_v_jh *= (f_k - f_i);
        v_ki_x_v_ji *= (f_h - f_i);

        // Calculate the final gradient, which is the sum of each component
        // divided by the six times the volume.
        vec3 grad = v_ik_x_v_hk;
        grad += v_ih_x_v_jh;
        grad += v_ki_x_v_ji;
        grad *= (1.0 / six_vol);

        const int l2_dof = l2_table_row[l2_table_col[i]];
        gradient[3*l2_dof+0] = grad[0];
        gradient[3*l2_dof+1] = grad[1];
        gradient[3*l2_dof+2] = grad[2];
    });
}
```

3.5 Transferring the fields to the solution space

Having calculated the scalar field solutions to the Laplace-Dirichlet equations, which live in H^1, and their gradients, which live in L^2, we have to make a choice of which of these spaces the final fibers should be calculated in. As explained at the beginning of this chapter, either the scalar fields have to be projected to L^2, or the gradient vector fields have to be interpolated to H^1, depending on whether we want the fibers to be defined per-element or per-vertex. We will now look at the approach taken to make sure both cases are handled, and how we make sure it is done on the GPU.

3.5.1 Per-element: Project scalar fields to L^2

As outlined in Section 3.2, all the scalar fields that are needed in the LDRB algorithm, that is ϕ_{epi}, ϕ_{lv}, and ϕ_{rv}, have to be projected from H^1 to L^2. Since we only need the gradient of the ψ_{ab}, it does not have to be projected. Just like when calculating the gradients, this could very simply be done by going the route of using MFEMs functionality. An example of this is shown in Code listing A.4. The problem with this approach is that this operation is (at the time of writing) not implemented with GPU support. Therefore, we use a different approach. We know that we are working with a tetrahedral mesh with first-order elements, i.e., four DoFs per element with basis functions in $\mathbb{P}_1$, located in each vertex. We want to transform a scalar field defined in each vertex to L^2, i.e., one DoF in the centroid of each element. Based on these restrictions, we can bypass the generalized projection scheme used by MFEM, and instead, do the projection "by hand".

Inside a tetrahedral element T_i, with vertices $p_{i,0}$, $p_{i,1}$, $p_{i,2}$, and $p_{i,3}$, all the basis functions except the ones in its vertices will be zero. So the value in any point $(x,y,z) \in T_i$ is given by

$$u(x,y,z) = c_{i,0}\phi_{i,0}(x,y,z) + c_{i,1}\phi_{i,1}(x,y,z) + c_{i,2}\phi_{i,2}(x,y,z) + c_{i,3}\phi_{i,3}(x,y,z), \tag{3.3}$$

where $\phi_{i,j}$ is the basis function in vertex j. To evaluate u in the centroid of T_i, we first look at the reference tetrahedron $\hat{T}$ with vertices $\hat{p}_0 = (0,0,0)$, $\hat{p}_1 = (1,0,0)$, $\hat{p}_2 = (0,1,0)$, and $\hat{p}_3 = (0,0,1)$. In the reference tetrahedron, it is straight forward to show that the $\mathbb{P}_1$ basis functions are defined as

$$\begin{cases} \hat{\phi}_0(\hat{x},\hat{y},\hat{z}) = 1-\hat{x}-\hat{y}-\hat{z}, \\ \hat{\phi}_1(\hat{x},\hat{y},\hat{z}) = \hat{x}, \\ \hat{\phi}_2(\hat{x},\hat{y},\hat{z}) = \hat{y}, \\ \hat{\phi}_3(\hat{x},\hat{y},\hat{z}) = \hat{z}. \end{cases} \tag{3.4}$$

See for example [29, Example 1.33]. To transfer from the reference tetrahedron $\hat{T}$ to T_i, the affine transformation operator

$$F_{T_i}(\hat{x},\hat{y},\hat{z}) = p_{i,0} + \hat{x}(p_{i,1}-p_{i,0}) + \hat{y}(p_{i,2}-p_{i,0}) + \hat{z}(p_{i,3}-p_{i,0}) \tag{3.5}$$

is used to map a point $(\hat{x}, \hat{y}, \hat{z}) \in \hat{T}$ to the corresponding point $(x, y, z) \in T_i$. The transformation is visualized in Figure 3.4.

The expression for the value in a point $(x, y, z) \in T_i$ in Equation 3.3 can then be rewritten using a combination of the affine transformation in Equation 3.5 and an application of the basis functions on the reference tetrahedron from Equation 3.4:

$$\begin{aligned} u(x,y,z) = c_{i,0}(\hat{\phi}_0 \circ F_{T_i}^{-1})(x,y,z) + c_{i,1}(\hat{\phi}_1 \circ F_{T_i}^{-1})(x,y,z) \\ + c_{i,2}(\hat{\phi}_2 \circ F_{T_i}^{-1})(x,y,z) + c_{i,3}(\hat{\phi}_3 \circ F_{T_i}^{-1})(x,y,z). \end{aligned} \tag{3.6}$$

The affine transformation will "preserve" the centroid of the tetrahedron, so instead of finding the centroid γ of T_i and mapping it to the reference tetrahedron, we can just find the centroid $\hat{\gamma}$ of the reference tetrahedron directly, which is at

$$\hat{\gamma} = \frac{\hat{p}_0 + \hat{p}_1 + \hat{p}_2 + \hat{p}_4}{4} = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right). \tag{3.7}$$

Insertion into Equation 3.5 shows we in fact derive the definition of the centroid γ of T_i, which is the average of the four vertices:

$$\begin{aligned} F_{T_i}(\hat{\gamma}) &= p_{i,0} + \frac{1}{4}(p_{i,1} - p_{i,0}) + \frac{1}{4}(p_{i,2} - p_{i,0}) + \frac{1}{4}(p_{i,3} - p_{i,0}) \\ &= \frac{1}{4}(p_{i,0} + p_{i,1} + p_{i,2} + p_{i,3}) \end{aligned}$$

By using its definition we can unwrap the composition operator, and see that

$$(\hat{\phi}_j \circ F_{T_i}^{-1})(x,y,z) = \hat{\phi}_j(F_{T_i}^{-1}(x,y,z)) = \hat{\phi}_j(\hat{x}, \hat{y}, \hat{z}). \tag{3.8}$$

Finally, we can insert Equation 3.8 into Equation 3.3, and express the value in the centroid γ of T_i as

$$\begin{aligned} u(\gamma) &= c_{i,0} \cdot \hat{\phi}_0(\hat{\gamma}) + c_{i,1} \cdot \hat{\phi}_1(\hat{\gamma}) + c_{i,2} \cdot \hat{\phi}_2(\hat{\gamma}) + c_{i,3} \cdot \hat{\phi}_3(\hat{\gamma}) \\ &= c_{i,0} \cdot \frac{1}{4} \quad + c_{i,1} \cdot \frac{1}{4} \quad + c_{i,2} \cdot \frac{1}{4} \quad + c_{i,3} \cdot \frac{1}{4} \end{aligned} \tag{3.9}$$

The conclusion is then that a projection from the finite element space H^1 with basis functions in $\mathbb{P}_1$ and DoFs in the vertices, to the finite element space L^2 with a DoF in the centroid, on a strictly tetrahedral mesh is simply an average of the four H^1 DoFs. So, to perform the projection, all that is needed is a mapping from each element T_i to its H^1 DoFs, and a mapping to its L^2 DoF.

Using the same element-to-DoF tables as in the gradient computation, as well as a read-only pointer to the H^1 DoFs and a write-only pointer to the L^2 DoFs, the projection can be performed using the kernel shown in Code listing 3.2 and assigning one thread to each element.

With both the projection and gradient computation offloaded to GPU, we have moved all the steps of the program, outlined in Figure 3.1, that take place after assembly of the linear systems to the GPU. Instead of having to move the solutions and gradients to and from the GPU, as we had to in the baseline implementation shown in Figure 3.2, they now remain on the device. The execution flow and data transfer now look like what is shown in Figure 3.5.

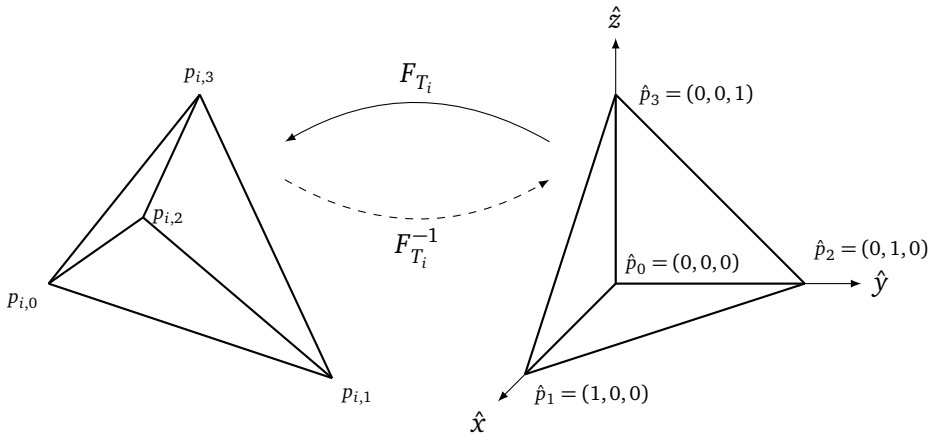


Figure 3.4: The affine transformations between the reference element, $\hat{T}$, and an arbitrary element, T_i. The F_{T_i} operator maps a point inside an arbitrary element T_i to the corresponding point on the reference element $\hat{T}$. The $F_{T_i}^{-1}$ operator maps the other way, from the reference element onto T_i.

Code listing 3.2: GPU-enabled kernel for projection of scalar field from H^1 to L^2.

```
void project_h1_to_l2(
    double *l2_vals,          // Solution in L2
    const double *h1_vals,    // Solution in H1
    const int ne,             // Number of elements
    const int *h1_table_col,  // H1 element-to-DoF table columns
    const int *h1_table_row,  // H1 element-to-DoF table rows
    const int *l2_table_col,  // L2 element-to-DoF table columns
    const int *l2_table_row)  // L2 element-to-DoF tableorows
{
    mfem::MFEM_FORALL(i, ne, {
        const int *h1_dofs = &h1_table_row[h1_table_col[i]];
        double avg = h1_vals[h1_dofs[0]] + h1_vals[h1_dofs[1]]
                   + h1_vals[h1_dofs[2]] + h1_vals[h1_dofs[3]];
        avg *= 0.25;
        l2_vals[l2_table_row[l2_table_col[i]]] = avg;
    });
}
```

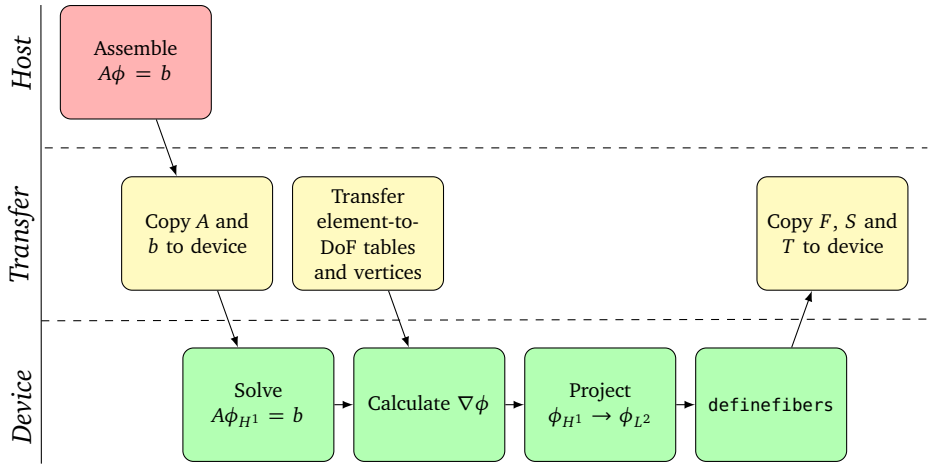


Figure 3.5: The execution and data transfer pattern when calculating fiber orientations on a per-element basis, and offloading solving of the linear systems, gradient computation, projection, and fiber calculation to the GPU, compared to the baseline shown in Figure 3.2. For clarity, only one of the four solutions is considered, denoted by ϕ. In the case of ψ_{ab}, the projection from H^1 to L^2 is not necessary.

3.5.2 Per-vertex: Interpolate gradients to H^1

As explained in Section 3.2, we have to transfer the values of the gradients to each vertex if we want the final fibers to be defined per-vertex. The problem with getting the gradients into H^1 is that they are not well-defined. The gradients are constant over each element, but technically undefined in the nodes. For this reason, we have to estimate.

For every node in the mesh (H^1 DoF), we look up which elements have this node as one of its vertices. Then, we loop over each of these elements and set the gradient in the node to be the average of the gradients in each element. The mapping from each node i to the elements that have this node as a vertex is obtained by the method `Mesh::GetVertexToElementTable()`. In addition to the vertex-to-element table, we also need the mapping between each element index i and the corresponding L^2 DoF. This is the same table as used in the gradient computation kernel and is therefore already in the device memory. With all the needed information, the interpolation is performed using the GPU-enabled kernel that shown in Code listing 3.3, by assigning one thread to each node/vertex of the mesh.

With this part offloaded to the GPU, we get the same execution and data transfer pattern as we got when computing fibers per-element as shown in Figure 3.5, with the only difference being that the projection step is swapped out for the interpolation explained here.

Code listing 3.3: GPU-enabled kernel interpolating a vector field from L^2 to H^1.

```
void interpolate_gradient_to_h1(
    double *h1_vals,          // Gradient in H1
    const double *l2_vals,    // Gradient in L2
    const int nv,             // Number of vertices
    const int *v2e_table_col, // Vertex-to-element table columns
    const int *v2e_table_row, // Vertex-to-element table rows
    const int *l2_table_col,  // L2 element-to-DoF table columns
    const int *l2_table_row)  // L2 element-to-DoF table rows
{
    // We interpolate the gradients in a DoF ‘i‘ in H1 by taking the average of
    // the gradients in the elements that ‘i‘ is a part of.
    mfem::MFEM_FORALL(i, nv, {
        // Find the elements connected to vertex i
        const int *element_indices = &v2e_table_row[v2e_table_col[i]];

        // The number of elements associated with vertex ‘i‘ is given by the
        // distance between the index to the first element associated with ‘i‘
        // and the index to the first element associated with ‘i+1‘. No need to
        // worry about overflowing, as there are ‘nv+1‘ columns in ‘v2e_table_col.
        const int num_elements = v2e_table_col[i+1] - v2e_table_col[i];

        // Loop over all the elements associated with vertex ‘i‘, and set the
        // gradient to the average of the element gradients
        vec3 grad = {0};
        for (int j = 0; j < num_elements; j++) {
            // Find the L2 DoF of this element
            const int l2_dof = l2_table_row[l2_table_col[element_indices[j]]];

            // Get the gradient of this element.
            vec3 element_grad = {
                l2_vals[3*l2_dof+0],
                l2_vals[3*l2_dof+1],
                l2_vals[3*l2_dof+2],
            };
            grad += element_grad;
        }
        grad *= (1.0 / (double) num_elements);

        h1_vals[3*i+0] = grad[0];
        h1_vals[3*i+1] = grad[1];
        h1_vals[3*i+2] = grad[2];
    });
}
```

3.6 Computing the fiber orientations

When we have the scalar field solutions to the Laplace-Dirichlet equations and their gradients, both on the same space, the fiber orientations can finally be calculated. As mentioned in Section 3.2, the fiber computation, which is performed by implementing the `definefibers` function (Function 5, page 15), was targeted for offloading to the GPU from the start. The reason is that the computations needed are exactly the kind that the GPU is designed to perform well on: The same operations performed on a large number of independent values.

We will now look at how the subroutines of `definefibers` are implemented.

3.6.1 Implementing the `axis` and `orient` functions

The `axis` and `orient` functions, defined in Function 2 and Function 3, respectively, are straight forward to implement in code. The only change made in the implementation is that the two matrices that are multiplied in `orient` to orient the axis system created by `axis` so that it has the correct fiber angle (α) and sheet angle (β) are merged into a single matrix, as the matrices are "constant", depending only on the evaluation of cos and sin of the angles.

The code implementing `axis` and `orient` is shown in Code listing A.5 and Code listing A.6, respectively.

3.6.2 Implementing the `bislerp` function

The implementation of the bidirectional spherical linear interpolation (bislerp) is mostly a straightforward translation of the pseudocode (Function 4, page 14) to C++. The implementation is shown in Code listing 3.5 (page 37). There are however some functions called by `bislerp` that we have not yet explained, namely `quat2rot`, `rot2quat`, and `slerp`. The implementation of these three functions is, as is also suggested in [6], based on the seminal paper "Animating Rotation with Quaternion Curves" by Ken Shoemake [27].

The `quat2rot` function is used, as the name implies, to convert a quaternion to its corresponding 3×3 rotation matrix. The implementation is based on the formula in Section I.1 in Appendix 1 of [27]. The code implementing this function is shown in Code listing A.7.

The `rot2quat` function performs the inverse of `quat2rot`, by converting a 3×3 rotation matrix to its corresponding quaternion. The implementation is based on the pseudocode in Section I.2 in Appendix 1 of [27]. The code implementing this function is shown in Code listing A.8.

The `slerp` function is the spherical linear interpolation routine that `bislerp` wraps with some logic to fulfill bi-directionality. The function takes two quaternions, q_1 and q_2, and produces a quaternion that is the result of spherically interpolating between the two by a factor $t \in [0, 1]$. The implementation is mostly a straightforward adaption of the formula in Section 3.3 of [27]. The implementation is shown in Code listing 3.4.

Code listing 3.4: Implementation of `slerp`, based on the formula in [27].

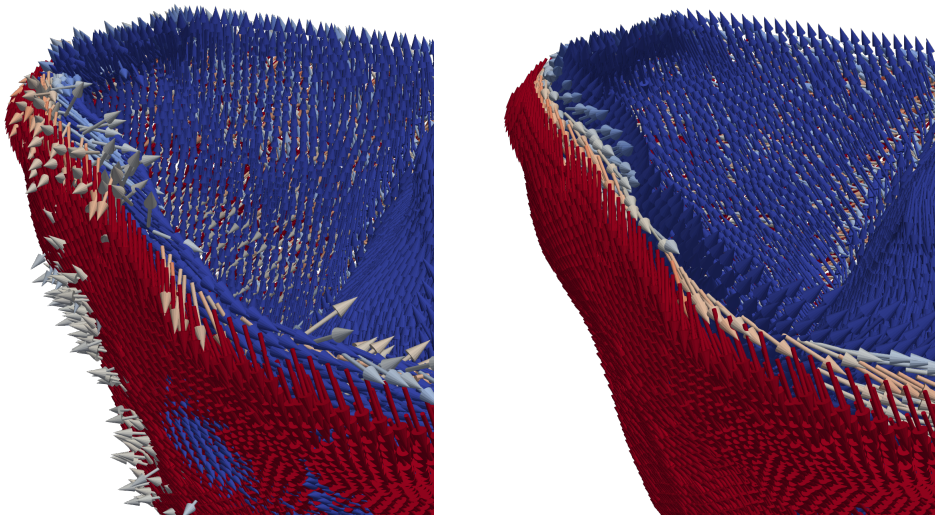
```
MFEM_DEVICE
static void slerp(quat& q, quat& q1, quat& q2, double t)
{
    double dot = quat_dot(q1, q2);
    // If the dot product is close to one, the angle approaches zero, and slerp
    // reduces to a regular linear interpolation.
    if (dot > 1-1e-12) {
        // Slerp(q1, q2, t) = ((sin(1-t)*theta)/sin(theta))q1
        //                  + ((sin(t)*theta)/sin(theta))q2
        // where theta = acos(q1 dot q2)
        const double angle = acos(dot);
        const double a = sin(angle * (1-t))/sin(angle);
        const double b = sin(angle * t)/sin(angle);

        q2b = q2
        q2b *= b;
        q = qa;
        q *= a;
        q += q2b;
    } else {
        q = q1;
        q *= (1-t);
        quat q2t = q2;
        q2t *= t;
        q += q2t;
    }
}
```

Code listing 3.5: Implementation of `bislerp`, as presented in Function 4.

```
MFEM_DEVICE
void bislerp(mat3x3& Qab, mat3x3& Qa, mat3x3& Qb, double t)
{
    // Translate the rotation matrices Qa and Qb into quaternions
    quat qa = {0}, qb = {0};
    rot2quat(qa, Qa);
    rot2quat(qb, Qb);

    // Find qm in { +qa, -qa, +i*qa, -i*qa, +j*qa,-j*qa, +k*qa, -k*qa}
    // that maximizes | qm dot qb |
    const double a = qa[0], b = qa[1], c = qa[2], d = qa[3];

    // Calculate the Hamilton products with the basic quaternions i,j, and k by
    // hand.
    quat i_qa = {-b,  a, -d,  c};
    quat j_qa = {-c,  d,  a, -b};
    quat k_qa = {-d, -c,  b,  a};
    quat minus_qa   = {-a, -b, -c, -d};
    quat minus_i_qa = {-i_qa[0], -i_qa[1], -i_qa[2], -i_qa[3]};
    quat minus_j_qa = {-j_qa[0], -j_qa[1], -j_qa[2], -j_qa[3]};
    quat minus_k_qa = {-k_qa[0], -k_qa[1], -k_qa[2], -k_qa[3]};

    quat *quat_array[8] = {
        &qa,   &minus_qa,   &i_qa, &minus_i_qa,
        &j_qa, &minus_j_qa, &k_qa, &minus_k_qa,
    };

    quat qm = {0};
    double max_abs_dot  = -1.0;
    for (int i = 0; i < 8; i++) {
        quat *v = quat_array[i];
        const double abs_dot = abs(quat_dot(*v, qb));
        if (abs_dot > max_abs_dot) {
            max_abs_dot = abs_dot;
            qm = *v;
        }
    }

    // We have found the candiate qm that requires the smallest rotation angle.
    // Interpolate with slerp.
    quat q = {0};
    slerp(q, qm, qb, t);
    quat_normalize(q);
    quat2rot(Qab, q);
}
```

3.7 Heuristics for a more robust definition of fiber orientations

With all the subroutines defined, the `definefibers` function (Function 5, page 15), which makes up the last part of the LDRB algorithm, could be implemented by copying the pseudocode verbatim. However, when doing so, we observe fibers in some vertices/elements that do not match the rules defined in [6], and that are visibly "unsmooth". This is especially visible in the center of the RV free wall, where the fibers point in seemingly random directions, which is shown in Figure 3.6a. We found that a probable cause of this is when a gradient with a magnitude that is approaching zero is used as the transmural component in an axis system that is input to `bislerp`. For example, the misdirected fibers in the RV free wall are produced by first performing a `bislerp` between Q_{lv} and Q_{rv} to produce Q_{endo}, which is then used in a `bislerp` with Q_{epi} to produce the final Q_{fiber}. In this area, the `depth` variable holds a value very close to one, and the first `bislerp` should yield $Q_{\text{endo}} = Q_{\text{lv}}$. However, the gradient $\nabla\phi_{\text{lv}}$, which is used as the transmural component of Q_{lv}, is very small and points in a seemingly arbitrary direction, which will cause even the slightest interpolation factor in `bislerp` to produce a wrong result. Based on this observation, we have created a set of heuristics that mitigate the use of gradients with small magnitudes in the interpolations.



(a) Without region heuristics **(b)** With region heuristics

Figure 3.6: The fibers generated with and without using region heuristics, on the section of the RV free wall furthest away from the septum of the mesh `heart02`. (a) shows the fibers that are generated when implementing Function 5 verbatim, where the fibers in the middle of the wall are directed in seemingly random directions. (b) shows the fibers generated when using the added heuristics presented here, which are smoothly interpolated from an angle α_{epi} on the epicardium to α_{endo} on the endocardium when increasing the distance from the epicardium.

We observe that in the septum, $\nabla\phi_{\text{epi}}$ has a magnitude that approaches zero, an observation that is also noted in [6]. Based on this information, we can use $||\nabla\phi_{\text{epi}}|| < \varepsilon_{\text{epi}}$ as a heuristic for when we are in the septum, for some small value ε_{epi}. An example of using $\varepsilon_{\text{epi}} = 0.01$ as a threshold for the septum is shown in Figure 3.7, where the septum is highlighted in red. When the heuristic is fulfilled, we can simply set $Q_{\text{fiber}} = Q_{\text{endo}}$. Furthermore, the transmural vectors for Q_{lv} and Q_{rv} are the same in the septum, which means that the `bislerp` used to find Q_{endo} is not needed, and we can set Q_{fiber} to either of Q_{lv} or Q_{rv}.

The same effects take place in both the LV and RV free walls. In the LV free wall, the direction of $\nabla\phi_{\text{rv}}$, which determines the transmural component of Q_{rv}, has a magnitude that approaches zero, but a direction that may affect Q_{endo} when interpolating between Q_{lv} and Q_{rv}. We can use $||\nabla\phi_{\text{lv}}|| > \varepsilon_{\text{lv}}$, $||\nabla\phi_{\text{rv}}|| < \varepsilon_{\text{rv}}$, and $||\nabla\phi_{\text{epi}}|| > \varepsilon_{\text{epi}}$ as a heuristic for when we are in the LV free wall. An example of using this heuristic with the thresholds $\varepsilon_{\text{epi}} = 0.01$, and $\varepsilon_{\text{lv}} = \varepsilon_{\text{rv}} = 0.05$ is shown in Figure 3.7, where the region that fulfills the heuristic is highlighted in green. Furthermore, when $\nabla\phi_{\text{rv}}$ approaches zero, $\nabla\phi_{\text{lv}}$ and $\nabla\phi_{\text{epi}}$ are equal, but with opposite signs. So, we can simply set $Q_{\text{fiber}} = Q_{\text{epi}}$.

As we saw in the introduction to this issue, the exact same argument can be used for the RV free wall, but in this case, it is $||\nabla\phi_{\text{lv}}||$ that approaches zero. The heuristic for the RV free wall is then that $||\nabla\phi_{\text{rv}}|| > \varepsilon_{\text{rv}}$, $||\nabla\phi_{\text{lv}}|| < \varepsilon_{\text{lv}}$, and $||\nabla\phi_{\text{epi}}|| > \varepsilon_{\text{epi}}$. An example of using this heuristic with the thresholds $\varepsilon_{\text{epi}} = 0.01$, and $\varepsilon_{\text{lv}} = \varepsilon_{\text{rv}} = 0.05$ is shown in Figure 3.7, where the region that fulfills the heuristic is highlighted in yellow. As for the LV free wall, we can set $Q_{\text{fiber}} = Q_{\text{epi}}$.

If none of the three aforementioned heuristics are satisfied, then the point lies in the junction between the septum, the LV free wall, and the RV free wall. The region is shown in blue in Figure 3.7. In the junction, all the gradients have a magnitude above their respective thresholds. For points in this region, we have to use the full LDRB algorithm. The observation that the weighted averaging of axis systems primarily occurs in this junction is also noted in [6].

We have adapted the implementation of `definefibers` function to use these heuristics and observations about simplifications that can be made in some of the regions. The code is shown in Code listing 3.6. The fibers generated for the right ventricle after adding this adaptation are shown in Figure 3.6b. Note that what values to use for ε_{epi}, ε_{lv}, and ε_{rv} will vary based on the gradients, which in turn is affected by factors such as input geometry and wall thicknesses. The values presented here are what we found worked well for this specific mesh.

A visualization of the generated longitudinal (F), sheet normal (S), and transverse (T) fiber orientations for a whole biventricular mesh are shown in Figure 3.8, and a streamlined visualization of the longitudinal fiber orientation (F) is shown in Figure 3.9.

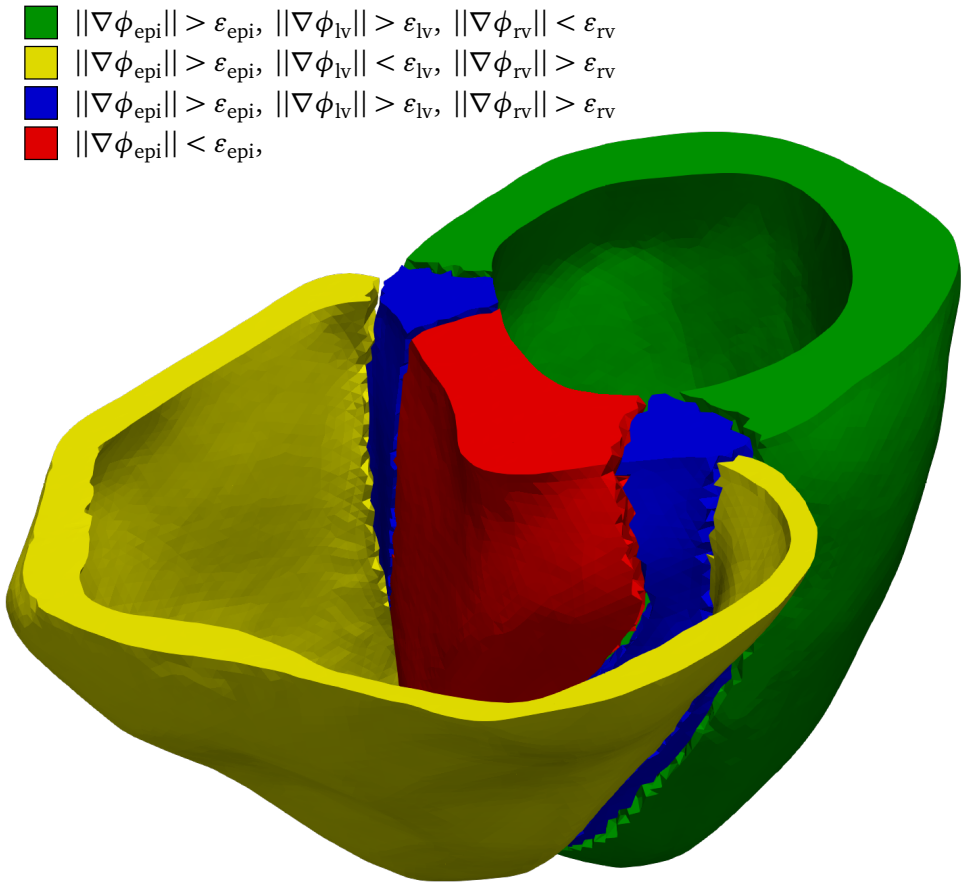
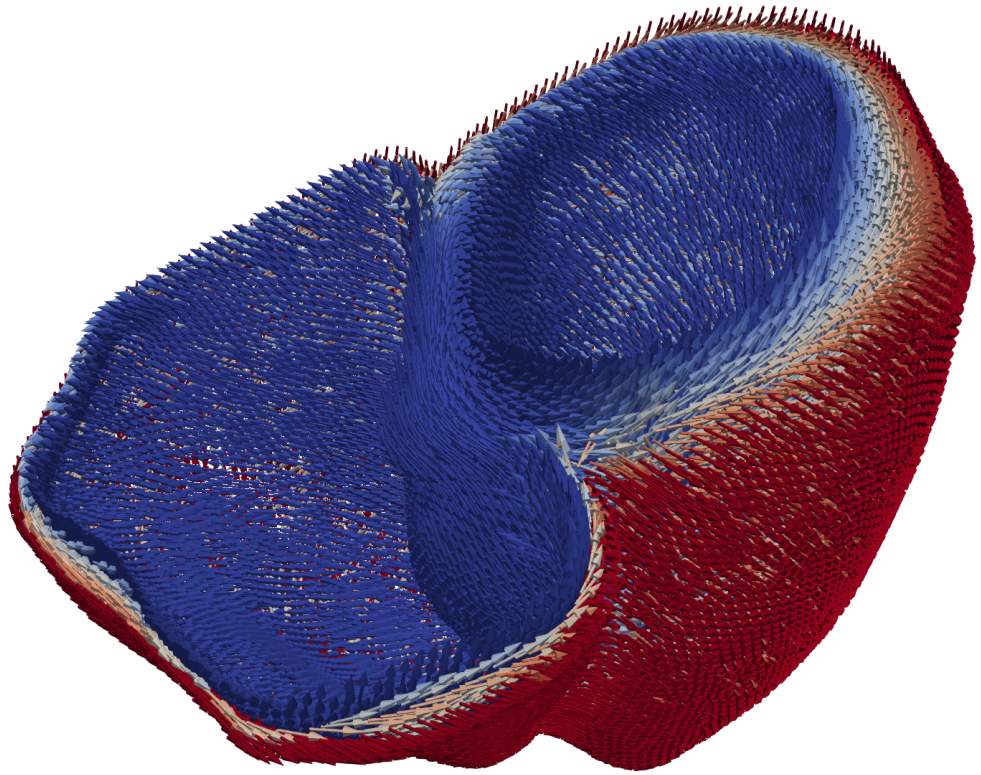
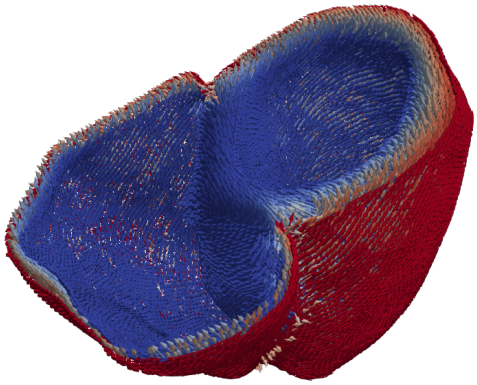


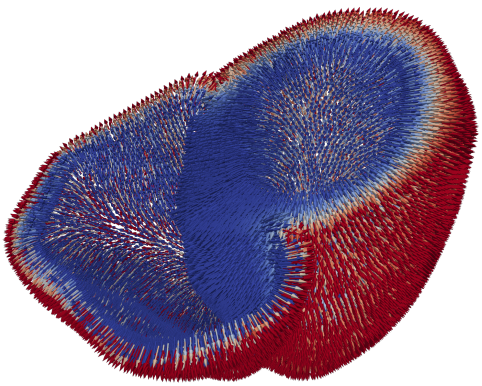
Figure 3.7: A visualization of the regions that are formed when using different combinations of thresholds on the magnitude of the gradients as a heuristic for location, on the mesh `heart02`. Here, the values $\varepsilon_{\text{epi}} = 0.01$ and $\varepsilon_{\text{lv}} = \varepsilon_{\text{rv}} = 0.05$ are used as the thresholds.



(a) Fiber, F



(b) Sheet normal, S



(c) Transverse, T

Figure 3.8: Fiber orientations generated by `cardiac-fibers`, using the fiber angles $\alpha_{\text{endo}} = 60°$ and $\alpha_{\text{epi}} = -60°$, and sheet angles $\beta_{\text{endo}} = \beta_{\text{epi}} = 0°$, on the mesh `heart02`. The tolerances used for the region heuristics are $\varepsilon_{\text{epi}} = 0.01$, and $\varepsilon_{\text{lv}} = \varepsilon_{\text{rv}} = 0.05$. Visualized using ParaView [26].

Code listing 3.6: Implementation of `definefibers`, as presented in Function 5.

```
void define_fibers(
    int n,
    const double *phi_epi, const double *phi_lv, const double *phi_rv,
    const double *grad_phi_epi, const double *grad_phi_lv,
    const double *grad_phi_rv, const double *grad_psi_ab,
    double alpha_endo, double alpha_epi, double beta_endo, double beta_epi,
    double *F, double *S, double *T,
    const double eps_epi, const double eps_lv, const double eps_rv)
{
    MFEM_FORALL(i, n,
    {
        // Clamp the Laplace-Dirichlet solutions to make sure they are in the
        // range [0,1]
        const double epi = CLAMP(phi_epi[i], 0.0, 1.0);
        const double lv  = CLAMP(phi_lv[i],  0.0, 1.0);
        const double rv  = CLAMP(phi_rv[i],  0.0, 1.0);

        // Convert the gradients to ‘vec3‘ from the input 1D arrays.
        vec3 grad_epi, grad_lv, grad_rv, grad_ab;
        vec3_set_from_ptr(grad_epi, &grad_phi_epi[3*i]);
        vec3_set_from_ptr(grad_lv,  &grad_phi_lv[3*i]);
        vec3_set_from_ptr(grad_rv,  &grad_phi_rv[3*i]);
        vec3_set_from_ptr(grad_ab,  &grad_psi_ab[3*i]);

        // Calculate the magnitude of the gradients so that we can do the region
        // heuristics.
        const double grad_epi_mag = vec3_magnitude(grad_epi);
        const double grad_lv_mag  = vec3_magnitude(grad_lv);
        const double grad_rv_mag  = vec3_magnitude(grad_rv);

        // Calculate the depth. In the cases where it is ill-defined, e.g. along
        // the epicardium, we just set it to 0, as it will not be used anyway.
        double depth = (lv > 0 || rv > 0) ? rv / (lv + rv) : 0.0;

        // Calculate the fiber and sheet angles for the septum and outer walls
        const double alpha_s = alpha_endo*(1.0-depth) - alpha_endo*depth;
        const double alpha_w = alpha_endo*(1.0-epi)   + alpha_epi*epi;
        const double beta_s  = beta_endo*(1.0-depth) - beta_endo*depth;
        const double beta_w  = beta_endo*(1.0-epi)   + beta_epi*epi;

        mat3x3 Q_lv = {{{0}}};
        {
            vec3 grad_lv_neg = grad_lv;
            vec3_negate(grad_lv_neg);

            mat3x3 T = {{{0}}};
            axis(T, grad_ab, grad_lv_neg);
            orient(Q_lv, T, alpha_s, beta_s);
        }

        mat3x3 Q_epi = {{{0}}};
        {
            mat3x3 T = {{{0}}};
            axis(T, grad_ab, grad_epi);
            orient(Q_epi, T, alpha_w, beta_w);
        }
```

```
        mat3x3 Q_fiber = {{{0}}};
        // Use the established heuristics to find out where in the geometry we are
        if (grad_epi_mag < eps_epi) {
            // We are in the septum. Since the magnitude of ‘grad_epi‘ is
            // under the threshold, we can assume that ‘grad_lv‘ and ‘grad_rv‘
            // are equal but oppositely directed. Can therefore set the fiber
            // orientations to ‘Q_lv‘ directly.
            Q_fiber = Q_lv;
        } else if (grad_epi_mag >= eps_epi
                && grad_lv_mag  >= eps_lv
                && grad_rv_mag  <  eps_rv) {
            // We are in the LV free wall. Since the magnitude of ‘grad_rv‘ is
            // under the threshold, we can assume that ‘grad_lv‘ and ‘grad_epi‘
            // are equal but oppositely directed. We can not use ‘Q_lv‘ for the
            // fibers, as it is not oriented with the epicardial angles, but
            // ‘Q_epi‘ is.
            Q_fiber = Q_epi;
        } else if (grad_epi_mag >= eps_epi
                && grad_lv_mag  <  eps_lv
                && grad_rv_mag  >= eps_rv) {
            // We are in the RV free wall. Since the magnitude of ‘grad_lv‘ is
            // under the threshold, we can assume that ‘grad_rv‘ and ‘grad_epi‘
            // are equal but oppositely directed. We can not use ‘Q_rv‘ for the
            // fibers, as it is not oriented with the epicardial angles, but
            // ‘Q_epi‘ is.
            Q_fiber = Q_epi;
        } else {
            // We are in the junction between the septum and the LV and RV free
            // walls, or in some other place where the heuristics don’t apply.
            // All the gradients (transmural) vectors are big enough to matter,
            // so we have to apply the full algorithm.
            mat3x3 Q_rv = {{{0}}};
            {
                mat3x3 T = {{{0}}};
                axis(T, grad_ab, grad_rv);
                orient(Q_rv, T, alpha_s, beta_s);
            }
            mat3x3 Q_endo = {{{0}}};
            bislerp(Q_endo, Q_lv, Q_rv, depth);
            bislerp(Q_fiber, Q_endo, Q_epi, epi);
        }

        // Set the output arrays (F S T) = Q_fiber
        F[3*i+0] = Q_fiber[0][0];
        F[3*i+1] = Q_fiber[1][0];
        F[3*i+2] = Q_fiber[2][0];
        S[3*i+0] = Q_fiber[0][1];
        S[3*i+1] = Q_fiber[1][1];
        S[3*i+2] = Q_fiber[2][1];
        T[3*i+0] = Q_fiber[0][2];
        T[3*i+1] = Q_fiber[1][2];
        T[3*i+2] = Q_fiber[2][2];
    });
}
```

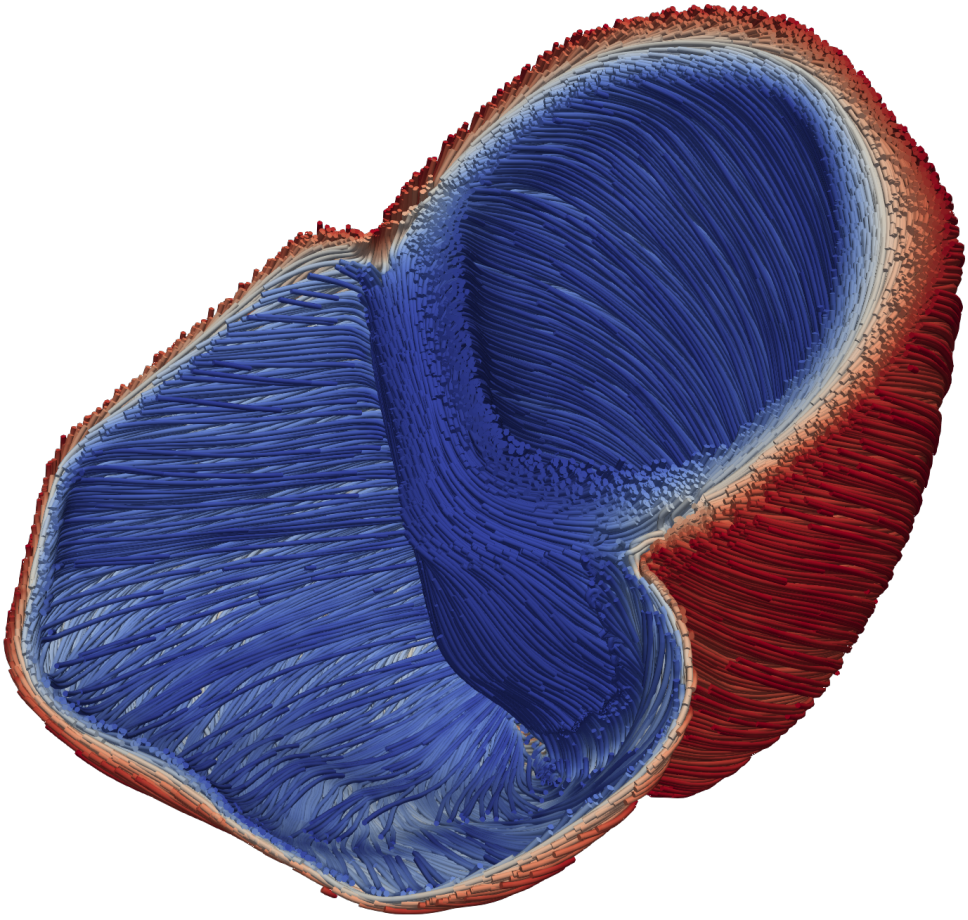


Figure 3.9: Streamlined visualization of fibers generated by `cardiac-fibers` on the mesh `heart02`. The fibers are the same as the ones shown in Figure 3.8a, but are visualized using ParaView's StreamTracer filter, which generates streamlines from a vector field.

Chapter 4

Results

In this chapter, we will dive into the numerical results from benchmarking of `cardiac-fibers`, the GPU-enabled implementation of the LDRB algorithm that was presented in Chapter 3. We will begin by presenting the experimental data that is used, as well as the hardware and software that is used. We will then look at the numerical results related to the performance when processing the experimental data.

4.1 Experimental data and environment

This section describes the data that is used in the experiments, what hardware and software is used, how the timing of the experiments is performed, as well as some information about the job setup.

4.1.1 Experimental data

The meshes used in the experiments were generated by using Gmsh [33]. The raw data used to generate the meshes is from the dataset published alongside [49], a study on acute myocardial ischemia. The meshes are generated with characteristic length parameters from 5 mm down to 0.09 mm, which is used to set the desired size for the mesh edges in Gmsh. In practice, Gmsh is not always able to produce the desired edge lengths, but they will be close. The generated meshes are described in Table 4.1, with information about the (desired) edge length, number of vertices, and number of elements. The mesh named `heart02` in the table is the mesh that has been used in all the figures involving a biventricular heart shown previously in this thesis.

4.1.2 Hardware and software configuration

To compare the performance of the fiber computation on the GPU with running only on the CPU, three different setups were used, using either AMD GPUs, NVIDIA

Table 4.1: Meshes used in the numerical experiments, which are generated from the dataset published alongside [49].

Mesh	Edge length [mm]	Vertices	Elements
`heart01`	5.00	1571	4717
`heart02`	1.00	46 421	210 101
`heart03`	0.50	307 679	1 607 708
`heart04`	0.40	560 180	3 031 704
`heart05`	0.30	1 288 388	7 205 076
`heart06`	0.20	4 066 503	23 595 379
`heart07`	0.15	9 400 914	55 603 164
`heart08`	0.10	30 865 049	186 274 337
`heart09`	0.09	42 093 384	255 047 053

GPUs, or only CPUs. All benchmarks were run on eX^3, an experimental, heterogeneous computational cluster for research in high-performance computing [1]. The same versions of libraries needed by `cardiac-fibers` are used in all three configurations. These are MFEM 4.5, *hypre* 2.25.0, METIS 5.1.0, and OpenMPI 4.1.4. We compile `cardiac-fibers` with the `-O3` and `--march=native` optimization flags for the `milanq` and `mi210q`. Due to instabilities in the timing when using these flags for the `hgx2q` build, they are omitted. Note that this only affects the host side code, as the device code still gets the `-O3` flag by default, and that the libraries are built with the same optimization flags as for `milanq` and `mi210q`.

`milanq` on eX^3

The four nodes of the `milanq` partition were used for the CPU-runs on eX^3. Just like the `mi210q`, each node has two AMD Epyc™ 7763 64-core processors, resulting in a total of 512 available cores across the four nodes.

The 7763 processor is part of the Epyc™7003 series of processors. It is a 64-core processor with 128 hardware threads. It has a base frequency of 2.45 GHz, and a max boost frequency of 3.50 GHz. It has an L3 cache size of 256 MB, and 8 DDR channels. The per-socket theoretical memory bandwidth is 204.8 GB/s. It has 128 PCIe Gen 4 lanes. [50]

In this configuration, `cardiac-fibers` is compiled with the `mpic++` compiler from OpenMPI 4.1.4. The MFEM build script for this configuration is shown in Code listing B.3.

In the tables and figures following in the rest of this chapter, this configuration will be referred to as `CPU-*`, where * is substituted with the number of cores used.

[1] `https://www.ex3.simula.no/`

`mi210q` on eX^3

The two nodes of the `mi210q` partition were used for AMD GPU-runs on eX^3. Each node has two AMD Instinct™ MI210 GPUs and one AMD Epyc™ 7763 64-core processor, for a total of 256 available cores and four available GPUs across the two nodes.

The MI210 GPU has 104 compute units and 6656 stream processors. It has 64 GB of memory, a memory clock of 1.6 GHz, and a peak memory bandwidth of 1.6 TB/s. The bus interface has both PCIe Gen 3 and Gen 4 support. The peak operations per second for the different floating point data types on the MI210 GPU are shown in Table 4.2. [51]

The achievable bandwidth, measured using the TRIAD benchmark in BabelStream [52], is 1288 GB/s.

In this configuration, `cardiac-fibers` is compiled with `hipcc` and uses the libraries from ROCm version 5.1.3. The MFEM build script for this configuration is shown in Code listing B.1.

In the tables and figures following in the rest of this chapter, this configuration will be referred to as `MI210-*`, where * is substituted with the number of GPUs used.

Table 4.2: Peak operations per second for the different floating point data types on the AMD MI210 GPU

Data type	Peak ops/second
`FP64/FP32` Vector	22.6 TFLOPS
`FP64/FP32` Matrix	45.3 TFLOPS
`FP16`	181.0 TFLOPS

`hgx2q` on eX^3

The single node of the `hgx2q` partition was used for NVIDIA GPU-runs on eX^3. The node has eight NVIDIA A100 GPUs and one AMD Epyc™ 7763 64-core processor, for a total of 256 available cores. Due to availability, we only use up to four GPUs.

The A100 GPU has 80 GB of memory, a memory clock of 1.512 GHz, and a peak memory bandwidth of 2.039 TB/s. The bus interface has both PCIe Gen 3 and Gen4 support. The peak operations per second for the different floating point data types on the A100 GPU is shown in Table 4.3. [53]

The achievable bandwidth, measured using the TRIAD benchmark in BabelStream [52], is 1768 GB/s.

In this configuration, `cardiac-fibers` is compiled with `nvcc` and uses the libraries from Cuda Toolkit version 11.8.0. The MFEM build script for this configuration is shown in Code listing B.2.

In the tables and figures following in the rest of this chapter, this configuration will be referred to as `A100-*`, where * is substituted with the number of GPUs used.

Table 4.3: Peak operations per second for the different floating point data types on the NVIDIA A100 GPU

Data type	Peak ops/second
FP64	9.7 TFLOPS
FP64 Tensor Core	19.5 TFLOPS
FP32	19.5 TFLOPS

Code listing 4.1: Timing technique used in numerical experiments, exemplified with the pattern used when timing a function that launches a GPU kernel.

```
double duration(struct timespec t0, struct timespec t1)
{
    return (t1.tv_sec - t0.tv_sec) + (t1.tv_nsec - t0.tv_nsec) * 1e-9;
}

void tick(struct timespec *t, bool barrier, bool device_barrier)
{
    if (device_barrier)
        MFEM_DEVICE_SYNC; // [cuda|hip]DeviceSynchronize
    if (barrier)
        MPI_Barrier(MPI_COMM_WORLD);
    clock_gettime(CLOCK_MONOTONIC, t);
}

void foo(...)
{
    struct timespec t0, t1;
    tick(&t0, /* MPI barrier */ true, /* device barrier */ false);
    this_function_launches_a_GPU_kernel(...);
    tick(&t1, /* MPI barrier */ true, /* device barrier */ true);
}
```

4.1.3 Timing

When timing the sections of interest in the code, we have to make sure that any kernel launched on a GPU is actually finished executing, not just finished launching. We also have to make sure that when we run in parallel with multiple MPI processes, all the processes are done executing. Therefore, the timing of all code that may launch a GPU kernel has an `MFEM_DEVICE_SYNC` after it, to make sure that any launched kernels are done, and an `MPI_Barrier`, to make sure all ranks have reached the same points. Timing is only reported by rank 0. The code used to achieve this is illustrated in Code listing 4.1.

4.1.4 Job setup

All benchmarks were run using the SLURM [54] job scheduler. To make sure there is as little interference from other processes as possible, all jobs were run with the `--exclusive` flag to make sure no other jobs are running on the allocated nodes.

To make sure processes do not get moved between different physical cores by the operating system scheduler during the run, all processes are bound to cores with the `--cpu-bind=rank` flag, meaning rank 0 is bound to core 0, rank 1 to core 1, and so on. This also has the effect of binding each rank to a *physical* core, so that no two ranks use the two hardware threads of the same physical core, in the case of the 7763 processor.

The binding of the available GPUs is done by setting the `ROCR_VISIBLE_DEVICES` environment variable. In the case where multiple processes share a smaller number of GPUs, the GPUs are divided among the processes in such a way that the first $n = \texttt{\#Procs}/\texttt{\#GPUs}$ are assigned to the GPU with ID 0, the next n to the GPU with ID 1, and so on.

4.2 Numerical results

In this section, we will look at the results of when running `cardiac-fibers` on the presented experimental data in the different hardware configurations. We will first look at the performance of the implementation of the algorithm as a whole. We do not consider the time spent loading meshes from disk or saving the results. We compare to an existing implementation, `ldrb`. After that, we look at where time is spent when running `cardiac-fibers` on CPU versus on GPU. We then consider what effect there is from increasing the number of cores in use. From there on, focus is mainly put on the separate parts of the algorithm that have been successfully transferred over to GPU, and compare the performance with the results obtained when running on GPU versus CPU. We will be considering fiber generation on a per-element basis in all experiments.

4.2.1 Performance of the full algorithm

As a baseline, we compare the single core performance of the full algorithm, between our implementation (`cardiac-fibers`) and `ldrb`. We do not consider the time spent loading the input mesh, nor the time spent saving the results, only the steps that are outlined in Algorithm 1. We use `ldrb` version `2022.5.0` with FEniCS verison `2019.1.0.post0`. Only `heart01-06` are considered, as `ldrb` fails to process any larger meshes. All runs are using a single core of the 7763 processor on the `milanq` partition of eX^3. The results are shown in Table 4.4. The source code used for the `ldrb` runs is shown in Code listing A.9. The single-core performance of `cardiac-fibers` is better than that of `ldrb` for all the considered meshes, with a minimal speedup of $\sim$ 15x.

The factors that play into why our implementation is faster are many, but two of the main ones are that `ldrb` is written in Python, an interpreted language, and the fact that it uses a direct solver, SuperLU, to solve the Laplace-Dirichlet equations.

If we consider the total run time of our implementation of the LDRB algorithm, we can split it into seven parts. The first part is the time that is spent setting up

Table 4.4: Comparison of the single core performance of `ldrb` and `cardiac-fibers` on the Epyc 7763 processor.

	`ldrb` [s]	`cardiac-fibers` [s]	Speedup
`heart01`	3.74	0.05	74.8
`heart02`	47.60	1.96	24.29
`heart03`	404.57	26.74	15.13
`heart04`	859.70	57.16	15.04
`heart05`	2652.48	159.06	16.68
`heart06`	18910.68	757.38	24.97

finite element spaces, allocating buffers, and so forth. The second part is the time spent assembling linear and bilinear forms of the Laplace-Dirichlet equations. The third part is the time spent forming linear systems from the linear and bilinear forms. The fourth part is the time spent solving the linear systems. The fifth part is the time spent computing the gradients of the solutions. The sixth part is the time spent projecting the solutions. The final part is the time spent doing the actual fiber calculations.

Table 4.5 shows the portions of time that is spent in each of these seven parts when processing `heart09` on a single core. Most of the time is spent in the assembly (35.03%) and solve (54.27 %) parts. The same table shows the time spent in each part when running on a single core while also using one GPU. In that case, the solve step has been offloaded to the GPU, and the time spent in solve is therefore substantially reduced (2.18%). Most of the remaining time is spent in assembly (85.25 %), a part of the program that is not offloaded to the GPU, and the time spent in this step is more or less equal to what we get when running only on a single core. We look at the solve step in more detail later in this section. Note that had we not been reusing the linear and bilinear forms in the first three linear systems, both the assemble part and form linear system part would take twice as long.

Next, we look at the performance of the full algorithm when increasing the number of cores used on `milanq`. For this, we consider the largest available mesh, `heart09`. The run time, strong scaling, and efficiency when running on 1 through 512 cores is shown in Table 4.6. On one core the full algorithm takes 12059 seconds, roughly 3 hours and 21 minutes. The fastest run, using 512 cores, takes roughly 32 seconds, resulting in a speedup of $\sim$ 378x. Overall, the algorithm shows good scalability, with a parallel efficiency ranging from 0.6 to 0.8 for all core counts.

4.2.2 Forming of linear systems

As Table 4.6 shows, there is a speedup of 1.16 when going from one to two cores, which equates to an efficiency of 0.66. Ideally, doubling the number of cores

Table 4.5: Run time of separate parts of the full algorithm on `heart09` when using a single core and when using a single MI210 GPU.

Part	`CPU-1`		`MI210-1`	
	Time [s]	% of total	Time [s]	% of total
Setup	382.34	3.2 %	361.22	7.2 %
Assemble	4224.24	35.0 %	4276.67	85.3 %
Form	616.97	5.1 %	257.77	5.1 %
Solve	6544.12	54.3 %	109.47	2.2 %
Gradient	145.49	1.2 %	1.42	0.0 %
Projection	22.07	0.2 %	0.28	0.0 %
Fiber	111.79	0.9 %	0.11	0.0 %
Others	11.92	0.1 %	9.92	0.2 %
Total	12058.95	100 %	5016.86	100 %

Table 4.6: Strong scaling of the full algorithm on `heart09`.

	Time [s]	Speedup	Efficiency
`CPU-1`	12058.95	1	1
`CPU-2`	9221.92	1.31	0.66
`CPU-4`	4724.00	2.55	0.64
`CPU-8`	2687.93	4.49	0.56
`CPU-16`	1243.71	9.70	0.61
`CPU-32`	576.81	20.91	0.65
`CPU-64`	295.45	40.82	0.64
`CPU-128`	134.78	89.47	0.70
`CPU-256`	56.14	214.80	0.84
`CPU-512`	31.84	378.73	0.74

should approximately halve the size of each process' submesh, and we should see a doubling in performance. In practice, however, we actually see a *slowdown* in some parts of the algorithm when going parallel, which causes a far-from-ideal speedup. The most significant slowdown is in the forming of the four linear systems. This is done in the `FormLinearSystem` method of the left-hand side `BilinearForm`, which applies transformations to the linear system, such as the elimination of boundary conditions. When running on one core, approximately 5.25% of the time is spent forming linear systems. When going to two cores, this portion bumps up to 40.4%, and the time it takes to form all the linear systems increases almost by a factor of 6. This portion is reduced as the number of cores is increased, as shown in Table 4.7. Note that the forming of the linear systems for ϕ_{lv} and ϕ_{rv} are practically free, as the already assembled left and right-hand sides are reused. This is the first part where we see a proper gain from offloading to GPU. When running on GPU, there is no reduction in performance when going from one to two processes/GPUs.

Table 4.7: Time spent in forming of the four linear systems

Cores	Form linear system [s]					% of total
	ϕ_{epi}	ϕ_{lv}	ϕ_{rv}	ψ_{ab}	Sum	
`CPU-1`	321.52	0.87	0.83	312.91	636.13	5.25 %
`CPU-2`	1867.92	0.43	0.43	1860.86	3729.64	40.44 %
`CPU-4`	970.52	0.23	0.24	968.19	1939.18	41.05 %
`CPU-8`	421.18	0.18	0.18	420.24	841.78	31.32 %
`CPU-16`	171.79	0.10	0.10	170.65	342.64	27.55 %
`CPU-32`	70.87	0.06	0.06	71.26	142.25	24.66 %
`CPU-64`	29.60	0.05	0.05	31.22	60.92	20.62 %
`CPU-128`	22.62	0.02	0.02	22.69	45.35	33.65 %
`CPU-256`	4.26	0.01	0.01	4.21	8.49	15.12 %
`CPU-512`	1.60	0.01	0.01	1.73	3.35	10.52 %
`MI210-1`	134.00	0.23	0.23	123.85	258.31	5.15 %
`MI210-2`	61.34	0.07	0.08	55.60	117.09	5.15 %
`MI210-4`	27.71	0.04	0.04	25.88	53.67	5.04 %

4.2.3 Solving the linear systems

The rest of this section will be spent on the results related to the parts of the program that are GPU-enabled, starting with the solving of the linear systems representing the Laplace-Dirichlet equations. First, we consider the speedup of the solves when running on CPU on `milanq`, by looking at the speedup obtained when increasing the number of cores compared to single-core performance, when running on `heart09`. The results are shown in Figure 4.1. As the figure shows, we get a linear speedup when increasing to two and four cores. From 8 through 64 cores, there is a sublinear speedup. For 128, 256, and 512 cores, the speedup

is close to linear again, at least for ϕ_{lv} and ϕ_{rv}. The superlinear speedup when doubling from 64 to 128 cores could be attributed to the fact that we go from using one to two sockets and therefore double the amount of available cache. The faster solve times for ϕ_{lv} and ϕ_{rv} when using 256 and 512 cores may be related to the fact that the left-hand side matrix from solving ϕ_{epi} is reused, and that we have enough cache when using four and eight sockets so that it is already hot in cache from solving ϕ_{epi}. As the left-hand side changes when solving for ψ_{ab}, there is an expected reduction in performance.

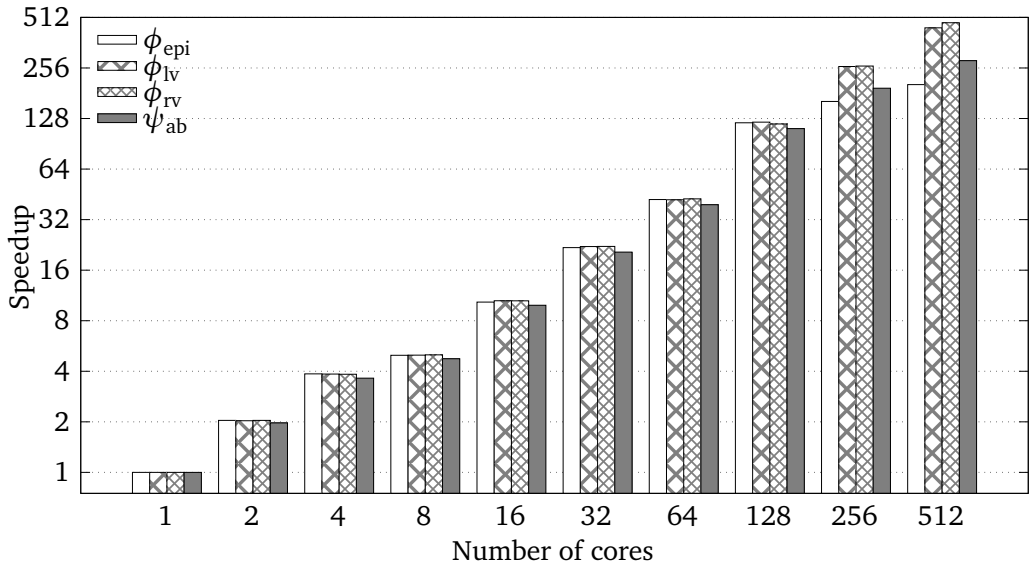


Figure 4.1: Speedup of CPU solve times on `heart09` compared to one core.

Next, we look at the performance when solving the linear systems on GPU. Again, we compare with one socket on `heart09`. The speedups are shown in Figure 4.2. When using one GPU, we see a speedup of $\sim$ 1.5x across all four linear systems. When going to two GPUs, we get a speedup of $\sim$ 2.7x, and using four GPUs we get $\sim$ 5.0x. So, we see a slightly sublinear speedup when increasing the number of GPUs. This may be attributed to the fact that solving the linear systems requires GPU-to-GPU communication. Unlike for the `milanq` runs with high core counts, we do not see any large differences in the run times across the four linear systems for any of the GPU runs, which makes sense when we consider the fact that there is no global caching of the left and right-hand sides.

4.2.4 Computing gradients

We first compare the difference between the naive implementation, outlined in Code listing A.3, and the kernel written with GPU execution in mind, Code listing 3.1. Even though the reason for writing the gradient calculations by hand rather than using the functionality MFEM was to run on the GPU, and not necessarily to speed up the CPU version, we suspect that it will also be faster on the

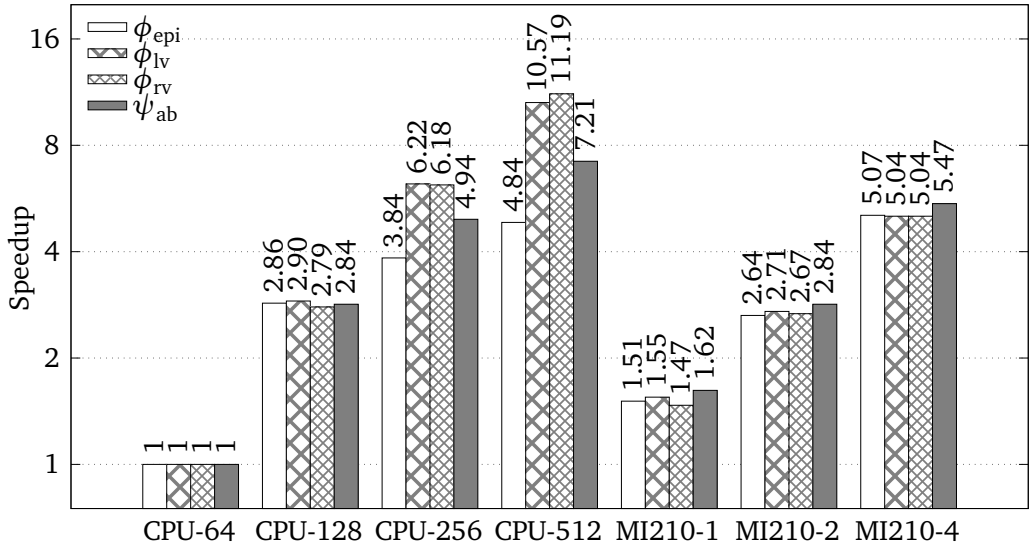


Figure 4.2: Speedup of CPU and GPU solve times on `heart09` compared to one socket (64 cores).

CPU. Table 4.8 shows the "naive" implementation compared to the handwritten one when calculating the gradient $\nabla\phi_{rv}$, running on `heart09` with 1 through 64 cores. As the table shows, the handwritten implementation has a minimal speedup of $\sim$ 2.6x. Backed by these results, we choose to use the handwritten gradient kernel when running on the CPU only as well.

Table 4.8: Naive versus improved implementation of calculation of $\nabla\phi_{rv}$ on `heart09`.

	Naive [s]	Improved [s]	Speedup
`CPU-1`	190.396	36.477	5.22
`CPU-2`	95.059	16.602	5.73
`CPU-4`	47.386	8.299	5.71
`CPU-8`	22.374	6.418	3.49
`CPU-16`	11.912	3.264	3.65
`CPU-32`	6.144	1.751	3.51
`CPU-64`	3.510	1.354	2.59

Next, we consider the performance when doing the gradient computations on GPU. Again, we compare the performance of one through four MI210 GPUs with multiples of the socket size of the Epyc 7763 processor, when running on `heart09`. The speedups are shown in Figure 4.3. One MI210 GPU gets $\sim$ 3.75x speedup compared to one socket, and when we increase the number of GPUs we get a linear speedup. The speedup when increasing the number of CPU sockets is

superlinear, with eight sockets (512 cores) seeing a speedup of up to ∼ 41x. This may be attributed to the increased cache size or the increased amount of memory.

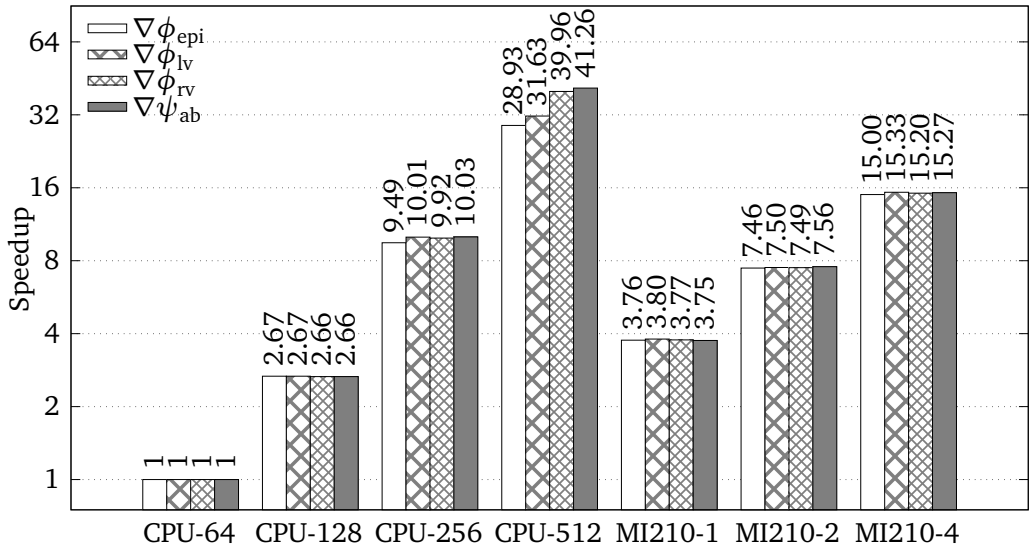


Figure 4.3: Speedup of CPU and GPU gradient computations on `heart09` compared to one socket (64 cores).

4.2.5 Projecting the solutions

Just like with the gradients, the reason for hand-writing a kernel to do projection was not to get speedup on CPU, but rather to make it possible to run on GPU. But, the same argument can be used as with the gradients, and we would expect a speedup when compared to the "naive" implementation in MFEM. Table 4.9 shows the speedup of the handwritten kernel when performing the projection of ϕ_{rv}, running on 1 through 64 cores on `heart09`. The handwritten implementation has a minimal speedup of ∼ 10x compared to the naive. As with the gradient kernel, we choose to use the handwritten projection kernel when running only on the CPU as well.

Next, we consider the performance when doing the projections on GPU. Again, we compare the performance of one through four MI210 GPUs with multiples of the socket size of the Epyc 7763 processor, when running on `heart09`. The speedups are shown in Figure 4.4. The speedup when increasing the number of sockets used is superlinear, with eight sockets (512 cores) seeing a speedup of up to 16x. This may be attributed to the increased cache size or the increased amount of memory. One MI210 GPU gets 1.75x speedup compared to one socket, and when we increase the number of GPUs we get a linear speedup.

Table 4.9: Naive versus improved implementation of the projection of ϕ_{rv} on `heart09`.

	Naive [s]	Improved [s]	Speedup
`CPU-1`	140.141	6.889	20.34
`CPU-2`	69.321	3.373	20.55
`CPU-4`	34.493	1.789	19.28
`CPU-8`	16.208	1.569	10.33
`CPU-16`	8.521	0.725	11.75
`CPU-32`	4.169	0.336	12.41
`CPU-64`	2.355	0.164	14.36

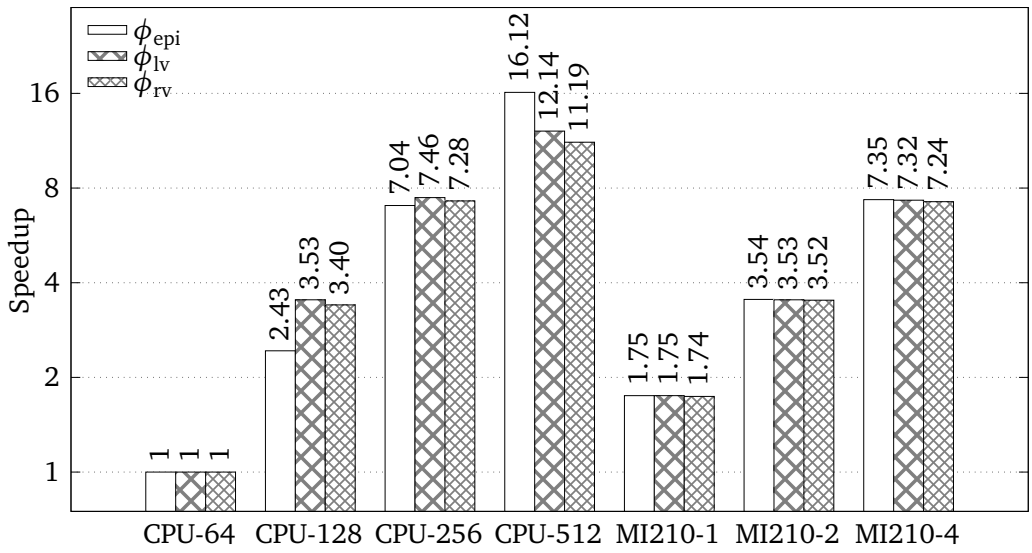


Figure 4.4: Speedup of CPU and GPU projections on `heart09` compared to one socket (64 cores).

4.2.6 Defining the fibers

We now look to the final part of the algorithm, `definefibers`, which is where the actual fiber computation takes place. Again, we compare the performance of one through four MI210 GPUs with multiples of the socket size of the Epyc 7763 processor, when running on `heart09`. Table 4.10 shows the run time, number of elements processed per second, and the speedup compared to one socket (64 cores).

The fastest CPU run, using 512 cores, obtains a peak throughput of $1039 \cdot 10^6$ elements per second, for a total run time of 0.246 seconds on `heart09`. Peak performance is obtained when running on four MI210 GPUs, where we get a throughput of $9073 \cdot 10^6$ elements per second, for a total run time of 0.028 seconds on `heart09`.

Table 4.10: CPU and GPU performance of `definefibers` on `heart09`.

	Time [s]	Melem/s	Speedup
`CPU-64`	2.062	123.67	1.00
`CPU-128`	0.988	258.15	2.09
`CPU-256`	0.518	492.35	3.98
`CPU-512`	0.246	1038.79	8.40
`MI210-1`	0.108	2367.40	19.14
`MI210-2`	0.054	4732.21	38.27
`MI210-4`	0.028	9073.18	73.37

4.2.7 Multiple processes per GPU

The results presented show that there is a performance gain running the parts that are able to run on GPU. But, as Table 4.5 shows, the majority of the time spent after offloading these parts to the GPU is in the assembly of the linear and bilinear forms (85.25%), which still has to happen on the host. This would be a natural part of the program to try to move to the GPU in an attempt to reduce the run time of the full algorithm. But, MFEM does not currently support assembly on GPU for simplex (tetrahedral) elements.

Another way to reduce the time spent in the parts of the program that run host side is to increase the number of processes per GPU, where each process gets a dedicated core. When increasing the number of processes, we hopefully get a speedup of the parts of the program that run on the host (assembly, setup), and at the same time not get a slow-down in the parts that run on the GPU. The run time of the separate parts of the full algorithm on `heart09` when using 1, 2, 4, and 8 processes sharing one GPU are shown in Table 4.11. The table reflects what we suspected. Setup, assembly, and forming of linear systems largely see an ideal speedup when increasing the number of processes. In the case of assembly, it is

even superlinear, with 8 cores getting a speedup of 9.64x compared to one core. At the same time, we see that the time spent on parts that are running on the GPU, i.e., solve, gradient computation, projection, and fiber calculation, stays constant. In some cases, like for the solve, there is even a slight speedup, which may be attributed to the increased number of cores being able to hide the NUMA effects.

Table 4.11: Run time of separate parts of the full algorithm on `heart09` when using multiple processes sharing one GPU.

Part	`MI210-1`, 1 process			`MI210-1`, 2 processes		
	Time [s]	% of total	Speedup	Time [s]	% of total	Speedup
Setup	361.22	7.20 %	1	178.59	7.64 %	2.02
Assemble	4276.67	85.25 %	1	1921.84	82.23 %	2.23
Form	257.77	5.14 %	1	122.41	5.24 %	2.11
Solve	109.47	2.18 %	1	107.69	4.61 %	1.02
Gradient	1.42	0.03 %	1	1.42	0.06 %	1.00
Projection	0.28	0.01 %	1	0.29	0.01 %	0.97
Fiber	0.11	0.00 %	1	0.11	0.00 %	1.00
Others	9.92	0.20 %	–	4.66	0.20 %	–
Total	5016.86	100 %	1	2337.01	100 %	2.15
	`MI210-1`, 4 processes			`MI210-1`, 8 processes		
	Time [s]	% of total	Speedup	Time [s]	% of total	Speedup
Setup	83.36	7.20 %	4.33	40.98	6.56 %	8.81
Assemble	902.19	77.89 %	4.74	443.7	71.04 %	9.64
Form	65.47	5.65 %	3.94	36.8	5.89 %	7.00
Solve	103.42	8.93 %	1.06	100.65	16.12 %	1.09
Gradient	1.33	0.11 %	1.07	1.30	0.21 %	1.09
Projection	0.27	0.02 %	1.04	0.26	0.04 %	1.08
Fiber	0.11	0.01 %	1.00	0.12	0.02 %	0.92
Others	2.10	0.18 %	–	0.75	0.12 %	–
Total	1158.25	100 %	4.33	624.56	100 %	8.03

Our experiments with increasing the number of processes per GPU show that we can use two, four, or eight processes without any reductions in performance in the parts of the program that are offloaded to the GPU, as shown in Table 4.11. Attempts to double once more, for a total of 16 processes per GPU, effectively stalls when solving the linear systems. No further investigation into the reason for why this happens has been conducted.

4.2.8 Comparing performance between AMD and NVIDIA GPUs

Finally, we would like to see how the performance compares when using GPUs from different vendors. For this, we compare the AMD MI210 GPUs from the `mi210q` configuration with the NVIDIA A100 GPUs from the `hgx2q` configuration.

We compare the performance by looking at the run time of the main GPU-enabled parts of the program. That is the solves, gradient computations, projections, and the final fiber computation. We run on `heart07`, using one GPU in each run. The results are shown in Table 4.12. In the gradient, projection, and fiber computation kernels, the performance of the A100 is close to that of the MI210, with the A100 seeing an average speedup of $\sim$ 1.14x. Based purely on the hardware specifications, we would expect the A100 to perform better than the MI210 on bandwidth-limited tasks, as it has a $\sim$ 30% higher theoretical maximum bandwidth.

In solving of the linear systems, the A100 performs $\sim$ 2.2x better on average in the solving of the linear systems. This may be partly attributed to the increased bandwidth. Another reasonable cause is that the sparse linear algebra libraries used on the A100, cuSPARSE, may be more highly optimized than rocSPARSE, which is used on the MI210.

Table 4.12: Run time of the main GPU-enabled parts of the program when running on `heart07` with one AMD MI210 GPU versus one NVIDIA A100 GPU.

Step		`MI210-1` [s]	`A100-1` [s]	Speedup
Solve	ϕ_{epi}	5.4447	2.2922	2.38
	ϕ_{lv}	5.3836	2.3617	2.28
	ϕ_{rv}	5.2754	2.1458	2.46
	ψ_{ab}	6.1407	3.6928	1.66
Gradient	$\nabla\phi_{\text{epi}}$	0.0780	0.0719	1.08
	$\nabla\phi_{\text{lv}}$	0.0769	0.0533	1.44
	$\nabla\phi_{\text{rv}}$	0.0771	0.0610	1.26
	$\nabla\psi_{\text{ab}}$	0.0766	0.0691	1.11
Project	ϕ_{epi}	0.0195	0.0177	1.10
	ϕ_{lv}	0.0195	0.0199	0.98
	ϕ_{rv}	0.0194	0.0200	0.97
`definefibers`		0.0244	0.0208	1.17

Chapter 5

Discussion

In this chapter, we will discuss some key numerical results presented in Chapter 4, as well as some aspects of our implementation of the LDRB algorithm presented in Chapter 3.

5.1 Assembly of the linear systems

As presented in Subsection 4.2.1, most of the time is spent in the assembly of the linear systems after all the subsequent parts of the algorithm have been offloaded to the GPU. In Subsection 4.2.7 we looked at how the time spent in the CPU-bound assembly can be reduced by increasing the number of processes/cores and having them share the GPU for the offloaded parts. The results show that we got the expected speedup in the CPU-bound parts, and saw the same performance, or even some speedup, in the parts running on the GPU. However, this is done only because we try to mitigate the fact the assembly is not being done on the GPU.

MFEM does offer some assembly levels and techniques that can be performed directly on the GPU. One such level is what is called "FULL" in the MFEM terminology, where a global sparse matrix is assembled on the GPU. Another assembly method is the "ELEMENT" form, where the left-hand side is stored as element-level dense matrices, which supposedly should store the data in a more GPU-friendly format. However, these techniques are currently only supported for tensor-product elements (quads and hexes), and not for tetrahedral elements, as we have limited ourselves to in our implementation. [55]

Theoretically, there is no requirement for the element type to support tensor products in order for it to be possible to do the assembly on GPU, and research has been published on efficient ways to do GPU-based assembly on simplex meshes. For example, in [56] the authors present a technique for GPU-based assembly of arbitrary polynomial order finite elements, which uses the restriction of simplex elements to derive combinatorial equations that allow for exact allocation of the sparse matrix, and directly assemble into the matrix on GPU.

5.2 Solving the linear systems

In Subsection 4.2.1 we compared the performance of the full algorithm when using the `ldrb` implementation against our own implementation, where we saw a minimal speedup of $\sim$ 15x. We briefly mention the use of the SuperLU solver in `ldrb` as one of the possible reasons why our implementation is faster. Although we do not have benchmarks of just the solve phase, we can assume that one of the contributions to its lower performance is the solver,. Solving the linear systems this way requires LU-factorization, which in turn requires fill-ins of the upper and lower triangular factors. This fill-in may require an order of magnitude or more memory than just storing the non-zeros, which may contribute to why `ldrb` is not able to process meshes larger than `heart06`.

In Subsection 4.2.3 we saw the performance when using the BoomerAMG preconditioner and PCG solver to solve the linear systems on one MI210 GPU is comparable to using between one and two sockets of the Epyc 7763 processor. We also observed that when we increase the number of GPUs, we get a slightly sublinear speedup, which we can attribute to the introduction of GPU-to-GPU communication needed when using more than one GPU. We will now look at the results in light of what has been observed by others, and then look at some possible areas of improvement.

In [45], the authors investigate the performance of using the BoomerAMG preconditioner and PCG solver on a 7-point 3D finite difference Laplace problem. The performance of four NVIDIA V100 GPUs is compared to 40 MPI tasks running on IBM Power 9 CPUs. They find that on a $600 \times 600 \times 600$ grid, the speedup of the GPU run is 3.2x. Our experiments on `heart09`, a mesh with an element count in that same order of magnitude, show that there is a $\sim$ 5x speedup obtained from using four MI210 GPUs over 64 cores. We, therefore, note that we have observed a speedup on GPUs that is in the same order of magnitude as in [45], while also taking into account that there is different hardware being used and that we are comparing the performance of an unstructured problem with a structured one.

In [46], the authors investigate the use of BoomerAMG and PCG on two nodes of the Spock early access system at Oak Ridge National Labs, where each node has four AMD MI100 GPUs and one 64-core AMD Epyc 7702 (Rome) CPU. Architecturally, this is very similar to the `mi210q` node on eX^3 used in our numerical experiments. The authors use the same kind of 7-point 3D finite difference Laplace problem as in [45]. A set of different parameter combinations are tested, but in relation to our experiments we can focus only on the default one, where the authors present an $\sim$ 8.4x speedup in the base configuration GPU runs compared to the base configuration CPU runs, on a $400 \times 400 \times 400$ grid. In this case, the authors observe a speedup greater than we find here.

An aspect of the solving that we have not elaborated much on is the tuning of the parameters. As shown by both [45] and [46], a lot of the performance of the BoomerAMG preconditioner is found by tuning a number of parameters. In this thesis, we have only considered the default parameters that are set in MFEM's

bindings to *hypre*. An improvement in the performance of the solve may be found by tuning the preconditioner and solver, for example by using the parameters presented in [45] and [46].

One way to further reduce the time spent solving would be to reduce the tolerance. All the linear systems are solved with a stop condition relative error of 10^{-20}, which is a much stricter convergence requirement than, for example, the 10^{-7} used in [6]. Such a strict tolerance was chosen to reduce the likelihood of wrong fiber generation as a result of numerical errors, such as the ones discussed in Section 3.7.

.

5.3 Effects of moving gradient computation and projection to GPU

In Subsection 4.2.4 and Subsection 4.2.5 we saw that the gradient and projection kernels perform well on the GPU, with gradient computations seeing a speedup of $\sim$ 4x per added MI210 GPU, compared to one socket (64 cores) of the Epic 7763 processor, and projection seeing a speedup of $\sim$ 1.7x per added GPU. There is however a second aspect that we have not yet explored, which is the data transfers that are mitigated.

In Section 3.2 we outlined the steps that need to take place from the input mesh to the final fiber orientations, and we also flagged two primary parts which we wanted to offload to the GPU, specifically the solving of linear systems and the final fiber computation. This resulted in the baseline execution flow and data transfer pattern presented in Figure 3.2 (page 24). In Section 3.4 and Subsection 3.5.1 we explained the methods used to offload both gradient computation and projection to the GPU, which improves upon the baseline pattern from Figure 3.2 to the one presented in Figure 3.5. We have also seen from the numerical results that both the gradient computation and projection benefits from being offloaded to the GPU, but what we have not considered is the cost of the data transfers that no longer need to take place.

We assume the solving of the linear systems has been performed on the GPU, and the scalar field solutions to ϕ_{epi}, ϕ_{lv}, ϕ_{rv}, and ψ_{ab} reside in device memory. If the gradient computation takes place on the host, all four have to be copied back to the device. The total size of this transfer is $4 \cdot n \cdot$ `sizeof(double)`, where n is the number of vertices in the mesh. If we consider `heart09`, at 42 093 384 vertices, these transfers have a total size of 1.35 GB. All four of the computed gradients then need to be transferred back to the host. There is one 3D vector per element, so these have a total size of $4 \cdot 3 \cdot m \cdot$ `sizeof(double)`, where m is the number of elements. If we consider `heart09`, at 255 047 053 elements, these transfers have a total size of 24.48 GB. In addition, the projected solutions ϕ_{epi}, ϕ_{lv}, and ϕ_{rv} have to be transferred to the device before the final fiber calculations can take place. These have a total size of $3 \cdot m \cdot$ `sizeof(double)`, which for `heart09` turns out to

be 6.12 GB. So to get the final gradients and projected scalar fields on the GPU, we first have to transfer 1.35 GB of data back to the host, wait for the gradients to be computed, and then receive a transfer of 24.48 GB back from the host, wait for the projections to be calculated and then receive a transfer of 6.12 GB from the host. The transferring of the gradients may of course overlap with the calculation of the projections. The `mi210q` is configured with 16 PCIe lanes per GPU, for a theoretical bandwidth of 32 GB/s. So the data transfers take roughly a second in total. If we now compare this to the run times for one GPU shown in Table 4.5, the data transfer becomes very significant.

If the gradient calculation and projection take place on the device, none of the previously mentioned transfers have to take place. Instead, we only need to transfer the element-to-DoF tables for H^1 and L^2, as well as the vertices. The H^1 element-to-DoF table is of size $m + 4 \cdot m = 5m$. The L^2 element-to-DoF table is of size $m + m = 2m$. The vertex table is of size n. In total, three transfers with a total size of $(7m + n) \cdot$ `sizeof(double)` are needed before calculating gradients and projections can take place on the device. For `heart09`, this results in a total transfer of 14.62 GB. Ideally, if there is enough device memory, we would have the transfer of the tables and vertices overlap with the solving of the linear system. If we assume that they are masked and that the tables are available as soon as the linear systems are solved, the combination of the four gradient computations and projections take roughly 1.7 seconds (as per Table 4.5). This means that just the data transfers needed to do gradient computation and projection on the host are almost as expensive as doing the actual computations on the device.

5.4 Evaluation of the gradient, projection, and fiber computation kernels

As shown in the results, the relative time spent transforming the scalar fields that result from solving the Laplace-Dirichlet equations into the final fibers is small. When doing fiber computation on one MI210 GPU, the total time spent in these steps adds up to less than 2%, as is shown in Table 4.5. The results also show that the performance of all these kernels scales well, both for the number of cores used and for the number of GPUs used. With this in mind, no further effort has been put into increasing the performance of these steps. However, it is worth discussing how well these kernels actually perform in light of theoretical limitations, and consider some possible improvements.

5.4.1 The gradient kernel

Each thread calculates the gradient in a single element. The first thing each thread has to do is to look up in the H^1 element-to-DoF table to find the indices of the four vertices of this element. There is one lookup to find the row for this element and one lookup for each vertex. The indices are stored as `int`, so the total read size is $5 \cdot$ `sizeof(int)` $= 20$ bytes. Each thread then has to read the physical coordinates

of each of the four vertices of the element, which are all `doubles`. This is a total of $4 \cdot 3 \cdot$ `sizeof(double)` $= 96$ bytes. Each thread also has to read the values of the scalar solution from each vertex. This is a total of $4 \cdot$ `sizeof(double)` $= 32$ bytes. All these reads amount to a total of 148 bytes. When storing the resulting gradient, one 3D vector is written, which is of size $3 \cdot$ `sizeof(double)` $= 24$ bytes. The total read-write size for each element is 172 bytes. Using one MI210 GPU on `heart09`, the fastest of the gradient computations takes 0.357 seconds, which equates to an effective bandwidth of 122.9 GB/s, a mere 7.7% of the theoretical bandwidth.

One obvious limitation in the gradient computation is the reading of the vertex coordinates, as the vertices that form an element are placed in an unstructured manner in memory. To get a good bandwidth utilization, GPUs rely heavily on *coalescing*. A coalesced memory access is one where consecutive threads read or write to consecutive memory addresses, and the read/write can be combined into a single transaction. In the case of the vertex reads, consecutive threads may have vertex indices that are far away from each other in memory, and in the worst case, we get no combination of reads into shared transactions.

A simple way to reduce the cost of the vertex lookups would be to fuse the computation of the four gradients into a single kernel. This is possible because the only thing that changes between the gradients is the value of the scalar solution in each vertex, and we can simply take all of the scalar solutions as the input simultaneously. That is, instead of the single `laplace` array as input to the `compute_gradient` function (Code listing 3.1, page 29), we would take all four scalar fields, and have the output be all four gradients. This way, the expensive vertex lookups for each element would only have to happen once. This would likely further improve the performance of the gradient calculations, both on the CPU and GPU.

5.4.2 The projection kernel

The projection kernel performs just a couple of additions and a multiply per element and we, therefore, assume that it is bound by bandwidth. Six array lookups of `ints` are needed to find the indices of the vertex values, and four `doubles` are read with these indices. Before storing the interpolated value, two more `ints` have to be read to find the index to store at. A total of 64 bytes are read by each thread. Each thread then has to store one `double`, which gives a total read-write size of 72 bytes. On `heart09`, this results in a total of $1.84 \cdot 10^{10}$ bytes. The fastest of the projections on one MI210 GPU takes 0.0944 seconds, for an effective bandwidth utilization of 12.2%.

The low bandwidth utilization may be caused by the series of dependent memory accesses. That is, we do a lookup in one array to find the index to use to do a lookup in a second array. In the projection kernel, this is done in both the array that is read from and the one written to. Just like with the gradient kernel, this becomes a coalescing problem.

5.4.3 The fiber computation kernel

When running the fiber computation, we assign one thread to each element. Each thread has to read in one scalar value from each of ϕ_{epi}, ϕ_{lv}, and ϕ_{rv}, as well as one 3D vector from each of $\nabla\phi_{\text{epi}}$, $\nabla\phi_{\text{lv}}$, $\nabla\phi_{\text{rv}}$, and $\nabla\psi_{\text{ab}}$. Storing all values as `double`, this means that each thread has to read $(3+4\cdot 3)\cdot$ `sizeof(double)` $= 120$ bytes. Each thread also has to write one 3D vector into each of F, S, and T, for a total write size of $3{\cdot}3{\cdot}$`sizeof(double)` $= 72$ bytes. As a result, the total size of reads and writes is 192 bytes. For `heart09`, which has 255 047 053 elements, the total read-write size of `definefibers` is $48.97\cdot 10^9$ bytes. As per Table 4.10, the run time of `definefibers` on `heart09` using one MI210 GPU is 0.108s, for a resulting effective bandwidth of $\sim$ 453 GB/s. This equates to 28.3% of the theoretical bandwidth of the MI210 GPU. The same calculations show that the effective bandwidth when using two and four GPUs is also $\sim$ 28%. When running on CPU, we get an effective bandwidth of $\sim$ 12% for one, two, four, and eight sockets.

Unlike the gradient and projection kernels, the reads and writes done in the fiber computation kernel are easily coalesced, in that thread i reads at an offset i in a number of input arrays, and writes at an offset i to a number of output arrays.

Determining why there is such a low bandwidth utilization would require profiling of the kernel, which may show that the kernel is not bandwidth bound after all, which we have assumed here.

5.5 Limitations on the maximal mesh size

In the current version of MFEM, the reading of serial meshes has to happen on every rank when running in parallel. That is, every rank has to read the entire mesh before performing a partitioning to find its submesh. This very quickly becomes a memory problem, which increases both with the mesh size and the number of processes. According to the MFEM developers, the rationale behind this is that each rank should read a coarse mesh, partition it, and then perform uniform refinement to increase the number of elements/DoFs.

To mitigate this, and be able to have the limiting factor of the maximal mesh size instead be the amount of memory in the system, all the meshes were pre-partitioned into submeshes of size $1, 2, 4, \ldots, 512$ using a single process, and stored in MFEM's native `v1.2` format which has support for parallel meshes. When running the fiber computation program, each process then only needs to read in its submesh, which will lead to a total memory footprint which is more or less equal to what you get when running in serial. The functionality of partitioning on a single process is not part of any of the currently released major versions of MFEM, so a separate build of MFEM based on the in-development branch `mesh-partitioner-dev`[1] was used, which adds this capability through the `MeshPartitioner` and `MeshPart` classes. According to the discussion in the pull request

[1] `https://github.com/mfem/mfem/tree/mesh-partitioner-dev`

tracking this branch on GitHub[2], this functionality is milestoned for the 5.0 release of MFEM.

There is still a roof for how big of a serial mesh we are able to pre-partition using MFEM. MFEM uses 32-bit signed integers for indices, and in the process of loading a serial mesh a new array will be allocated which is of length $6 \cdot m$, where m is the number of elements in the mesh. This means that a mesh with more than $2^{31}/6$ elements will cause a signed integer overflow. A way around this would be to use another parallel mesh format supported by MFEM, such as PUMI [57]. Another workaround is to do as the MFEM developers intended and load the coarsest possible mesh and perform refinement of the submeshes on each rank. This is probably the best solution when working with simple geometries such as regular cubes or beams, where the geometry itself is easily represented with a low number of polygons, but it may not be viable for complex geometries such as the biventricular heart models used in this project, especially if fiber generation is done as a pre-processing step for other cardiac simulations that use the same input mesh.

It is worth clarifying that this limitation only applies to the loading of the mesh, and not necessarily to the rest of the program. It is possible to build *hypre* with 64-bit indices, which allows for linear systems with billions of DoFs. The subsequent parts of the algorithm that we have offloaded to the GPU also do not have an inherent limit when it comes to the maximum number of elements/vertices that are possible to process. As long as enough GPUs are available to cover the amount of device memory needed, there should not be a problem.

[2] `https://github.com/mfem/mfem/pull/2669`

Chapter 6

Conclusions and Future Work

Being able to simulate cardiac functions accurately is of great interest to cardiology, a major field in medicine. Many of these simulation models require accurate information about cardiac fiber orientations. The most widely used rule-based method for simulating this is the *Laplace-Dirichlet Rule-Based method (LDRB)* which uses partial differential equations (PDEs) to capture the complex geometry of patient-specific anatomical models properly. These PDEs can be solved numerically on a computer using the finite element method (FEM), which enables the computation of cardiac fibers that are very close to those captured through atlas-based approaches such as DTI. However, solving these FEM-based PDEs is very computationally intensive, especially as the granularity of the anatomical model is increased.

Modern GPUs offer great computational power but are more challenging to program than CPUs. However, in FEM-based PDEs, the same operation is applied to many individual elements. This makes them very suitable for GPUs.

In this thesis, we showed how the LDRB method for cardiac fiber computation could be successfully implemented for both multicore CPUs and be offloaded to GPUs, and run in parallel in a distributed-memory fashion. This was done by leveraging the GPU-enabled solvers and preconditoners from the *hypre* library to solve the Laplace-Dirichlet equations that are involved in this method, as well as writing GPU kernels to perform the operations on the solutions to these equations that are needed to derive the final fiber orientations.

Specifically, we have used the MFEM software library to load a biventricular mesh from a common mesh format and perform all the needed operations to define the linear systems based on the Laplace-Dirichlet equations. We have then used the PCG solver and BoomerAMG preconditioner from *hypre* to efficiently solve these linear systems, both on CPUs and on GPUs.

By limiting ourselves to tetrahedral meshes, we were able to find closed-form solutions for the gradient calculation, projection, and interpolation that needed to be performed on the Laplace-Dirichlet solutions before the final fiber computation could take place. We have implemented these formulas in GPU kernels, so that also these operations can be performed on the GPU. In addition, we have

also implemented the final fiber computation as a GPU kernel. This way, all the operations that have to take place on the GPU – ranging from the linear systems have been assembled up until the fiber computation is finished, avoiding a lot of CPU-GPU memory transfers.

Finally, we proposed a heuristic based on the magnitude of the gradients of the solutions to the Laplace-Dirichlet equations, and showed that it could be used to more precisely reason about what region of the myocardium an arbitrary point is, which consequently makes it possible to define fiber orientations more robustly. This is of great importance for cardiac simulations that depend on these fiber orientations.

Performance results – summary

When comparing our LDRB algorithm implementation to an existing state-of-the-art Python-based version that uses a SuperLU solver, we observed a minimal speedup of $\sim$ 15x when running both on a single CPU core. Furthermore, we were also able to process larger meshes.

Our benchmarking results of the solver phase of the LDRB algorithm showed that we were able to solve these linear systems $\sim$ 1.5x faster using an AMD MI210 GPU, compared to using all 64 cores of an AMD Epyc 7763 CPU. We also saw that when we increased the number of GPUs to two or four, we got close to linear speedup. When comparing these results with work done by others who have used the same solver and preconditioner to solve linear systems on GPUs, we observed that the speedups we have seen are close to that found by others. Note that our solve phase is not just a stand-alone code, but part of a more complex application with real-world meshing and boundary conditions. Furthermore, we outlined some areas of improvement that may lead to faster solve times.

Our benchmarks showed that the by doing gradient calculation and projection of the solutions to the Laplace-Dirichlet equations on GPU, we get speedups between 1.5x and 3.8x when running on one AMD MI210 GPU, compared to all 64 cores of an AMD Epyc 7763 CPU. In addition, we showed that doing these operations on GPU reduces the amount of data transfers when the final fiber computation takes place on the GPU. Doing the final fiber computations on GPU also showed great performance, with a 19x speedup on one MI210 GPU compared to 64 cores. A discussion of the performance of these kernels compared to the theoretical limitations was also included, and some possible improvements were outlined.

We also compared the performance of our implementation on both AMD and NVIDIA GPUs. We found that they were very similar, except that the NVIDIA A100 card we used showed to be significantly faster for the solve phase than the AMD MI210 GPU, possibly due to the performance of underlying vendor libraries.

6.1 Future work

There are several ways to further improve the performance and usability of the work presented. The following sections list several of these possible avenues.

6.1.1 GPU-based assembly

The numerical results when running on GPU show that most of the time is spent in the CPU-bound assembly of the finite element matrices. Such assembly is currently not possible to do on GPUs in MFEM when the mesh has simplex elements. Future work may therefore involve either implementing GPU-based assembly for simplex element in MFEM, or bypassing MFEM entirely and directly assemble matrices on GPU that can be passed to *hypre* for solving.

6.1.2 Tuning of the preconditioner and solver

As discussed in Section 5.2, the PCG solver and BoomerAMG preconditioner we use to solve the linear systems use the default parameters that are set in MFEM's binding to *hypre*. We also compared performance with some other works that have investigated the tuning of these parameters. Future work could therefore include the application and evaluation of the presented parameters to find out if this can increase performance.

6.1.3 Profiling and optimization of GPU kernels

As discussed in Section 5.3 and Section 5.4, the offloading of the individual parts of the program has mostly been explored in the sense of making them work, and we also show that there are some clear areas of improvement when it comes to the GPU kernels. Future work might therefore involve proper profiling and optimizations of these kernels. This work could also include an investigation into whether the layer of convenience added by using the `MFEM_FORALL` macro has any negative effect on performance.

6.1.4 Algorithmic additions

The `ldrb` implementation adds some extra logic that makes it possible to define separate values of α_{endo} and β_{endo} for the left and right ventricle, which in turn makes it possible for the resulting fibers to better fit histological data. Future work might therefore involve adding this feature to `cardiac-fibers`. The heuristics used for locating which part of the myocardium a point is in, which we presented in Section 3.7, might help in this matter.

As mentioned in Section 1.2, recent works have investigated alterations to the original LDRB algorithm which makes it better fit the histological data when applied to other areas than the left ventricle, such as the septum, right ventricle,

outflow tracts, or even in the atrium. Future work might therefore include adding support for these adapted algorithms.

6.1.5 Further investigation of the region heuristic

The region heuristic presented in Section 3.7 makes it possible to more accurately identify where in the geometry an arbitrary point is, which makes the definition of the fiber orientations more robust. In the same section, we presented some threshold values for these heuristics which we found worked well for our specific mesh. Future work might therefore include analyzing other meshes based on patient-specific data, and investigating whether the values used can be generalized. Future work might also include using fibers generated using the region heuristics in some of the computational models that are dependent on the fiber orientations, and see if there are benefits.

Bibliography

[1] D. Le Bihan, J.-F. Mangin, C. Poupon, C. A. Clark, S. Pappata, N. Molko, and H. Chabriat, "Diffusion tensor imaging: Concepts and applications," *Journal of Magnetic Resonance Imaging: An Official Journal of the International Society for Magnetic Resonance in Medicine*, vol. 13, no. 4, pp. 534–546, 2001.

[2] D. F. Scollan, A. Holmes, R. Winslow, and J. Forder, "Histological validation of myocardial microstructure obtained from diffusion tensor magnetic resonance imaging," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 275, no. 6, H2308–H2318, 1998.

[3] S. A. Niederer, J. Lumens, and N. A. Trayanova, "Computational models in cardiology," *Nature Reviews Cardiology*, vol. 16, no. 2, pp. 100–111, 2019.

[4] P. Nielsen, I. Le Grice, B. Smaill, and P. Hunter, "Mathematical model of geometry and fibrous structure of the heart," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 260, no. 4, H1365–H1378, 1991.

[5] J. Wong and E. Kuhl, "Generating fibre orientation maps in human heart models using poisson interpolation," *Computer methods in biomechanics and biomedical engineering*, vol. 17, no. 11, pp. 1217–1226, 2014.

[6] J. D. Bayer, R. C. Blake, G. Plank, and N. A. Trayanova, "A novel rule-based algorithm for assigning myocardial fiber orientation to computational heart models," *Annals of biomedical engineering*, vol. 40, no. 10, pp. 2243–2254, 2012.

[7] Y. Liu, S. Jiao, W. Wu, and S. De, "Gpu accelerated fast fem deformation simulation," in *APCCAS 2008-2008 IEEE Asia Pacific Conference on Circuits and Systems*, IEEE, 2008, pp. 606–609.

[8] *HPCG - June 2022*, `https://top500.org/lists/hpcg/2022/09/`, Accessed: 2023-02-01.

[9] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, no. 100, 2015.

[10] H. Finsberg, *Laplace-Dirichlet Rule-Based (LDRB) algorithm for assigning myocardial fiber orientations*, version 2022.5.0, Nov. 2022. [Online]. Available: `https://github.com/finsberg/ldrb`.

[11] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, *et al.*, "Mfem: A modular finite element methods library," *Computers & Mathematics with Applications*, vol. 81, pp. 42–74, 2021.

[12] A. Logg, K.-A. Mardal, and G. Wells, *Automated solution of differential equations by the finite element method: The FEniCS book*. Springer Science & Business Media, 2012, vol. 84.

[13] J. Cranford, D. Richards, X. Zhang, S. Laudenschlager, J. Glosli, T. O'Hara, A. Mirin, E. Draeger, J.-L. Fattebert, R. Blake, *et al.*, "Cardioid," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.

[14] P. C. Africa, ": A flexible, high performance library for the numerical solution of complex finite element problems," *SoftwareX*, vol. 20, p. 101 252, 2022.

[15] P. C. Africa, R. Piersanti, M. Fedele, L. Dede', and A. Quarteroni, *An open tool based on lifex for myofibers generation in cardiac computational models*, 2022. DOI: 10.48550/ARXIV.2201.03303. [Online]. Available: https://arxiv.org/abs/2201.03303.

[16] D. Arndt, W. Bangerth, M. Feder, M. Fehling, R. Gassmöller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Sticko, B. Turcksin, and D. Wells, "The deal.II library, version 9.4," *Journal of Numerical Mathematics*, vol. 30, no. 3, pp. 231–246, 2022. DOI: 10.1515/jnma-2022-0054. [Online]. Available: https://dealii.org/deal94-preprint.pdf.

[17] R. Doste, D. Soto-Iglesias, G. Bernardino, A. Alcaine, R. Sebastian, S. Giffard-Roisin, M. Sermesant, A. Berruezo, D. Sanchez-Quintana, and O. Camara, "A rule-based method to model myocardial fiber orientation in cardiac biventricular geometries with outflow tracts," *International journal for numerical methods in biomedical engineering*, vol. 35, no. 4, e3185, 2019.

[18] R. Piersanti, P. C. Africa, M. Fedele, C. Vergara, L. Dedè, A. F. Corno, and A. Quarteroni, "Modeling cardiac muscle fibers in ventricular and atrial electrophysiology simulations," *Computer Methods in Applied Mechanics and Engineering*, vol. 373, p. 113 468, 2021.

[19] H. Zou, C. Xi, X. Zhao, A. S. Koh, F. Gao, Y. Su, R.-S. Tan, J. Allen, L. C. Lee, M. Genet, *et al.*, "Quantification of biventricular strains in heart failure with preserved ejection fraction patient using hyperelastic warping method," *Frontiers in physiology*, vol. 9, p. 1295, 2018.

[20] W. Commons. "Diagram of the human heart." File: Diagram of the human heart.svg. (2003), [Online]. Available: https://commons.wikimedia.org/wiki/File:Diagram_of_the_human_heart.svg.

[21] I. J. LeGrice, B. H. Smaill, L. Chai, S. Edgar, J. Gavin, and P. J. Hunter, "Laminar structure of the heart: Ventricular myocyte arrangement and connective tissue architecture in the dog," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 269, no. 2, H571–H582, 1995.

[22] P. Hunter, A. D. McCulloch, and H. Ter Keurs, "Modelling the mechanical properties of cardiac muscle," *Progress in biophysics and molecular biology*, vol. 69, no. 2-3, pp. 289–331, 1998.

[23] G. A. Holzapfel and R. W. Ogden, "Constitutive modelling of passive myocardium: A structurally based framework for material characterization," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 367, no. 1902, pp. 3445–3475, 2009.

[24] S. Rossi, R. Ruiz-Baier, L. F. Pavarino, and A. Quarteroni, "Orthotropic active strain models for the numerical simulation of cardiac biomechanics," *International journal for numerical methods in biomedical engineering*, vol. 28, no. 6-7, pp. 761–788, 2012.

[25] J. Sundnes, G. T. Lines, X. Cai, B. F. Nielsen, K.-A. Mardal, and A. Tveito, *Computing the electrical activity in the heart*. Springer Science & Business Media, 2007, vol. 1.

[26] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large data visualization," *The visualization handbook*, vol. 717, no. 8, 2005.

[27] K. Shoemake, "Animating rotation with quaternion curves," in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, 1985, pp. 245–254.

[28] P. G. Ciarlet, *The finite element method for elliptic problems*. SIAM, 2002.

[29] A. Ern and J.-L. Guermond, *Theory and practice of finite elements*. Springer, 2004, vol. 159.

[30] *Hypre: High performance preconditioners*, `https://llnl.gov/casc/hypre`, `https://github.com/hypre-space/hypre`.

[31] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[32] L. S. Avila, U. Ayachit, S. Barré, J. Baumes, F. Bertel, R. Blue, D. Cole, D. DeMarle, B. Geveci, W. A. Hoffman, B. King, K. Krishnan, C. C. Law, K. M. Martin, W. McLendon, P. Pebay, N. Russell, W. J. Schroeder, T. Shead, J. Shepherd, A. Wilson, and B. Wylie, *The VTK User's Guide 11th Editition*. Kitware, Inc., 2011.

[33] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities," *International journal for numerical methods in engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[34] Khronos ©OpenCL Working Group, *CUDA C++ Programming Guide*, 2022. [Online]. Available: `https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf`.

[35] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2023. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[36] Advanced Micro Devices, Inc., *HIP Programming Guide v5.3*, 2022. [Online]. Available: `https://docs.amd.com/bundle/HIP-Programming-Guide-v5.3/page/Introduction_to_HIP_Programming_Guide.html`.

[37] MFEM, *GPU support in MFEM*, `https://mfem.org/gpu-support/`, Accessed: 2022-12-12.

[38] I. Håkonsen, *A GPU-enabled implementation of the LDRB algorithm for computation of cardiac fiber orientations*, Nov. 2023. [Online]. Available: `https://github.com/ivhak/cardiac-fibers`.

[39] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[40] J. R. Shewchuk *et al.*, *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[41] X. Li, J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki, "SuperLU Users' Guide," Lawrence Berkeley National Laboratory, Tech. Rep., Sep. 1999, `https://portal.nersc.gov/project/sparse/superlu/ug.pdf`.

[42] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Mathematical Software*, vol. 29, no. 2, pp. 110–140, Jun. 2003.

[43] K. Stüben, "Algebraic multigrid: An introduction with applications," *Multigrid, Academic Press Inc., San Diego, CA*, 2001.

[44] R. D. Falgout, "An introduction to algebraic multigrid," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2006.

[45] R. D. Falgout, R. Li, B. Sjögreen, L. Wang, and U. M. Yang, "Porting hypre to heterogeneous computer architectures: Strategies and experiences," *Parallel Computing*, vol. 108, p. 102 840, 2021.

[46] P. Bauman, R. Li, and U. Yang, "Report on hypre performance on amd gpus," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2021.

[47] C. Mancinelli, M. Livesu, and E. Puppo, "A comparison of methods for gradient field estimation on simplicial meshes," *Computers & Graphics*, vol. 80, pp. 37–50, 2019.

[48] C. Mancinelli, M. Livesu, and E. Puppo, "Errata corrige," *Computers & Graphics*, vol. 80, 2022. [Online]. Available: `http://pers.ge.imati.cnr.it/livesu/papers/MLP18/MLP18_extended_errata.pdf`.

[49] H. Martinez-Navarro, A. Mincholé, A. Bueno-Orovio, and B. Rodriguez, "High arrhythmic risk in antero-septal acute myocardial ischemia is explained by increased transmural reentry occurrence," *Scientific reports*, vol. 9, no. 1, pp. 1–12, 2019.

[50] Advanced Micro Devices, Inc., *AMD Epyc™ 7003 Series Processors*, 2022.

[51] Advanced Micro Devices, Inc., *AMD Instinct™ MI210 Accelerator*, 2022.

[52] T. Deakin and S. McIntosh-Smith, *BabelStream*, version 3.4, Apr. 2019.

[53] NVIDIA Corporation, *NVIDIA A100 Tensor Core GPU*, 2021.

[54] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*, Springer, 2003, pp. 44–60.

[55] MFEM, *HowTo: Use partial assembly and matrix-free assembly*, `https://mfem.org/howto/assembly_levels/`, Accessed: 2023-02-01.

[56] J. S. Mueller-Roemer and A. Stork, "Gpu-based polynomial finite element matrix assembly for simplex meshes," in *Computer Graphics Forum*, Wiley Online Library, vol. 37, 2018, pp. 443–454.

[57] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, "Pumi: Parallel unstructured mesh infrastructure," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 3, pp. 1–28, 2016.

Appendix A

Additional code listings

This appendix contains the extra listings that have been referenced in the main parts of the text. Note that the listings that exemplify how some part of the program is implemented often do not contain the actual implementation used, but are rather meant to show the general approach. For the actual implementation, see the source code [38].

Code listing A.1: Example of setting up the linear system for ϕ_{epi} in MFEM.

```
// Setup the finite element space.
ParFiniteElementColleciton *fec;
fec = new H1_FECollection(/* order */ 1, parmesh->Dimension());
ParFiniteElementSpace fespace(parmesh, fec);

int nattr = parmesh->bdr_attributes.Max();
Array<int> essential_boundaries(nattr);          // Essential boundaries
Array<int> nonzero_essential_boundaries(nattr);  // Boundaries with value 1.0
Array<int> zero_essential_boundaries(nattr);     // Boundaries with value 0.0
Array<int> ess_tdof_list;

// Get a list of the essential (known value) DoFs.
essential_boundaries = 0;
essential_boundaries[EPI_ID-1] = 1;
essential_boundaries[LV_ID -1] = 1;
essential_boundaries[RV_ID -1] = 1;
fespace.GetEssentialTrueDofs(essential_boundaries, ess_tdof_list);

// Define the linear from (RHS) of the Laplace-Dirichlet equation
ParLinearForm *b = new ParLinearForm(&fespace);
ConstantCoefficient zero(0.0);
b->AddDomainIntegrator(new DomainLFIntegrator(zero));
b->Assemble();

// Define the bilinear form (LHS) of the Laplace-Dirichlet equation
ParBilinearForm *a = new ParBilinearForm(&fespace);
ConstantCoefficient one(1.0);
a->AddDomainIntegrator(new DiffusionIntegrator(one));
a->Assemble();

// Initialize x with initial guess of zero.
ParGridFunction *x_phi_epi = new ParGridFunction(&fespace);
*x_phi_epi = 0.0;

// The value is one on the epicardium.
nonzero_essential_boundaries = 0;
nonzero_essential_boundaries[EPI_ID-1] = 1;

// The value is zero on the left and right ventricle endocardiums.
zero_essential_boundaries = 0;
zero_essential_boundaries[LV_ID-1] = 1;
zero_essential_boundaries[RV_ID-1] = 1;

// Project the constant value 1.0 to all the essential boundaries marked as nonzero.
ConstantCoefficient nonzero_bdr(1.0);
x_phi_epi->ProjectBdrCoefficient(nonzero_bdr, nonzero_essential_boundaries);

// Project the constant value 0.0 to all the essential boundaries marked as zero.
ConstantCoefficient zero_bdr(0.0);
x_phi_epi->ProjectBdrCoefficient(zero_bdr, zero_essential_boundaries);

// Assemble the linear system.
OperatorPtr A;
Vector B, X;
a->FormLinearSystem(ess_tdof_list, *x_phi_epi, *b, A, X, B);
```

Code listing A.2: The alternative method used when setting up the boundary conditions and linear system for ψ_{ab}, compared to the approach for ϕ_{epi} in Code listing A.1. Read as a continuation of Code listing A.1.

```
// For apex to base, the base is the only essential boundary
essential_boundaries = 0;
essential_boundaries[BASE_ID-1] = 1;
fespace.GetEssentialTrueDofs(essential_boundaries, ess_tdof_list);

// If this rank has the apex in its submesh, add the DoF to the list of
// essential DoFs
int apex = find_apex();
int apex_dof = -1;
if (apex >= 0) {
    // Initialize the internal data needed in the finite element space
    fespace->BuildDofToArrays();
    // Find the dof at the local apex vertex
    apex_dof = fespace->GetLocalTDofNumber(apex);
    // Make sure the apex is in the list of essential true Dofs
    ess_tdof_list.Append(apex_dof);
}

// The DoFs have changed, reassemble the LHS and RHS.
b->Update();
b->Assemble();
a->Update();
a->Assemble();

// Initialize x with initial guess of zero.
ParGridFunction *x_psi_ab = new ParGridFunction(&fespace);
*x_psi_ab = 0.0;

// Set boundary condition 1.0 on the base surface
nonzero_essential_boundaries = 0;
nonzero_essential_boundaries[BASE_ID-1] = 1;
x_psi_ab->ProjectBdrCoefficient(nonzero_bdr, nonzero_essential_boundaries);

// Set the boundary condition 0.0 on the apex DoF
if (apex_dof >= 0) {
    Array<int> node_disp(1);
    node_disp[0] = apex_dof;
    Vector node_disp_value(1);
    node_disp_value[0] = 0.0;

    VectorConstantCoefficient node_disp_value_coeff(node_disp_value);
    x_psi_ab->ProjectCoefficient(node_disp_value_coeff, node_disp);
}

a->FormLinearSystem(ess_tdof_list, *x_psi_ab, *b, A, X, B);
```

Code listing A.3: Computing a gradient of a scalar field using MFEM, exemplified with the calculation of $\nabla\phi_{\text{epi}}$.

```
H1_FECollection h1_fec(1, mesh->Dimension());
FiniteElementSpace fespace_scalar_h1(&mesh, &h1_fec);
GridFunction x_phi_epi(&fespace_scalar_h1);

// Solve the Laplace-Dirichlet equation, solution in x_phi_epi
laplace(&x_phi_epi, ...)

L2_FECollection l2_fec(0, mesh->Dimension());
FiniteElementSpace fespace_vector_l2(&mesh, &l2_fec, 3, Ordering:byVDIM);
GridFunction grad_phi_epi(&fespace_scalar_l2);

// Calculate gradients
GradientGridFunctionCoefficient gfc(&x_phi_epi);
grad_phi_epi.ProjectCoefficient(gfc);
```

Code listing A.4: Projecting a scalar field from H^1 to L^2 using MFEM, exemplified with the projection of ϕ_{epi}.

```
H1_FECollection h1_fec(1, mesh.Dimension());
FiniteElementSpace fespace_scalar_h1(&mesh, &h1_fec);
GridFunction x_phi_epi(&fespace_scalar_h1);

// Solve the Laplace-Dirichlet equation, solution in x_phi_epi
laplace(&x_phi_epi, ...)

L2_FECollection l2_fec(0, mesh.Dimension());
FiniteElementSpace fespace_scalar_l2 (&mesh, &l2_fec);
GridFunction x_phi_epi_l2(&fespace_scalar_l2);

// Project the solution from H1 to L2
GridFunctionCoefficient gfc(&x_phi_epi);
x_phi_epi_l2.ProjectCoefficient(gfc);
```

Code listing A.5: Implementation of the `axis` function, as defined in Function 2 (page 12).

```
MFEM_DEVICE
void axis(mat3x3& Q, vec3& u, vec3& v)
{
    vec3 e0, e1, e2;

    e1 = u;
    vec3_normalize(e1);

    e2 = v;
    vec3 e1_dot_e2_e1;
    {
        const double e1_dot_v = vec3_dot(e1, v);
        e1_dot_v_e1 = e1;
        e1_dot_v_e1 *= e1_dot_v;
    }
    e2 -= e1_dot_v_e1;
    vec3_normalize(e2);

    // e0 = e1 x e2
    vec3_cross(e0, e1, e2);

    vec3_normalize(e0);

    Q[0][0] = e0[0]; Q[0][1] = e1[0]; Q[0][2] = e2[0];
    Q[1][0] = e0[1]; Q[1][1] = e1[1]; Q[1][2] = e2[1];
    Q[2][0] = e0[2]; Q[2][1] = e1[2]; Q[2][2] = e2[2];
}
```

Code listing A.6: Implementation of the `orient` function, as defined in Function 3 (page 13).

```
MFEM_DEVICE
void orient(mat3x3& Q_out, mat3x3& Q, double a, double b)
{
    const double sina = sin(a*PI/180.0);
    const double sinb = sin(b*PI/180.0);
    const double cosa = cos(a*PI/180.0);
    const double cosb = cos(b*PI/180.0);

    mat3x3 A;

    A[0][0] = cosa; A[0][1] = -sina*cosb; A[0][2] = -sina*sinb;
    A[1][0] = sina; A[1][1] =  cosa*cosb; A[1][2] = cosa*sinb;
    A[2][0] = 0;    A[2][1] = -sinb;      A[2][2] = cosb;

    mat3x3_mul(Q_out, Q, A);
}
```

Code listing A.7: Implementation of the `quat2rot` function, used to convert a quaternion to its corresponding rotation matrix. Based on the formula in Section I.1 in Appendix 1 of [27].

```
MFEM_DEVICE
void quat2rot(mat3x3& Q, quat& q)
{
    const double w = q[0], x = q[1], y = q[2], z = q[3];

    const double x2 = x*x;
    const double y2 = y*y;
    const double z2 = z*z;

    const double wx = w*x;
    const double wy = w*y;
    const double wz = w*z;

    const double xy = x*y;
    const double xz = x*z;

    const double yz = y*z;

    Q[0][0] = 1.0 - 2.0*y2 - 2.0*z2;
    Q[1][0] =       2.0*xy + 2.0*wz;
    Q[2][0] =       2.0*xz - 2.0*wy;

    Q[0][1] =       2.0*xy - 2.0*wz;
    Q[1][1] = 1.0 - 2.0*x2 - 2.0*z2;
    Q[2][1] =       2.0*yz + 2.0*wx;

    Q[0][2] =       2.0*xz + 2.0*wy;
    Q[1][2] =       2.0*yz - 2.0*wx;
    Q[2][2] = 1.0 - 2.0*x2 - 2.0*y2;

}
```

Code listing A.8: Implementation of the `rot2quat` function, used to convert a rotation matrix to its corresponding quaternion. Based on the pseducode in Appendix I.2 of [27].

```
MFEM_DEVICE
void rot2quat(quat& q, mat3x3& M)
{
    const double M11=M[0][0], M12=M[1][0], M13=M[2][0];
    const double M21=M[0][1], M22=M[1][1], M23=M[2][1];
    const double M31=M[0][2], M32=M[1][2], M33=M[2][2];

    const double w2 = 0.25 * (1 + M11 + M22 + M33);
    const double err = 1e-15;

    double w, x, y, z;

    if (w2 > err) {
        w = sqrt(w2);
        x = (M23 - M32) / (4.0*w);
        y = (M31 - M13) / (4.0*w);
        z = (M12 - M21) / (4.0*w);
    } else {
        w = 0.0;
        const double x2 = -0.5*(M22 + M33);
        if (x2 > err) {
            x = sqrt(x2);
            y = M12 / (2.0*x);
            z = M13 / (2.0*x);
        } else {
            x = 0.0;
            const double y2 = 0.5*(1-M33);
            if (y2 > err) {
                y = sqrt(y2);
                z = M23 / (2.0*y);
            } else {
                y = 0.0;
                z = 1.0;
            }
        }
    }
    q[0] = w; q[1] = x; q[2] = y; q[3] = z;
    quat_normalize(q);
}
```

Code listing A.9: Source code used for baseline benchmark of `ldrb`.

```
import dolfin
import ldrb
import sys
import os
import time

input_vol = sys.argv[1]
input_surf = sys.argv[2]

mesh = dolfin.Mesh()
with dolfin.XDMFFile(input_vol) as f:
    f.read(mesh)

surface_marker_collection = dolfin.MeshValueCollection("size_t", mesh, 2)
with dolfin.XDMFFile(input_surf) as f:
    f.read(surface_marker_collection, "markers")

ffun = dolfin.MeshFunction("size_t", mesh, surface_marker_collection)
ldrb_markers = { "base": 1, "epi": 2, "lv": 3, "rv": 4 }
fiber_space = "CG_1"

print(f"Calculating fibers for '{input_vol}'.")
t0 = time.time()
fiber, sheet, sheet_normal = ldrb.dolfin_ldrb(
    mesh=mesh,
    fiber_space=fiber_space,
    ffun=ffun,
    markers=ldrb_markers,
    alpha_endo_lv=60,  # Fiber angle on the endocardium
    alpha_epi_lv=-60,  # Fiber angle on the epicardium
    beta_endo_lv=0,    # Sheet angle on the endocardium
    beta_epi_lv=0,     # Sheet angle on the epicardium
)
t1 = time.time()
print('Time [ldrb.dolfin_ldrb]: {0}s'.format(t1-t0))
```

Appendix B

MFEM build scripts

This appendix contains listings with the scripts that were used to build MFEM for the hardware and software configurations presented in Subsection 4.1.2.

Code listing B.1: MFEM build script for the `mi210q` partition on eX^3

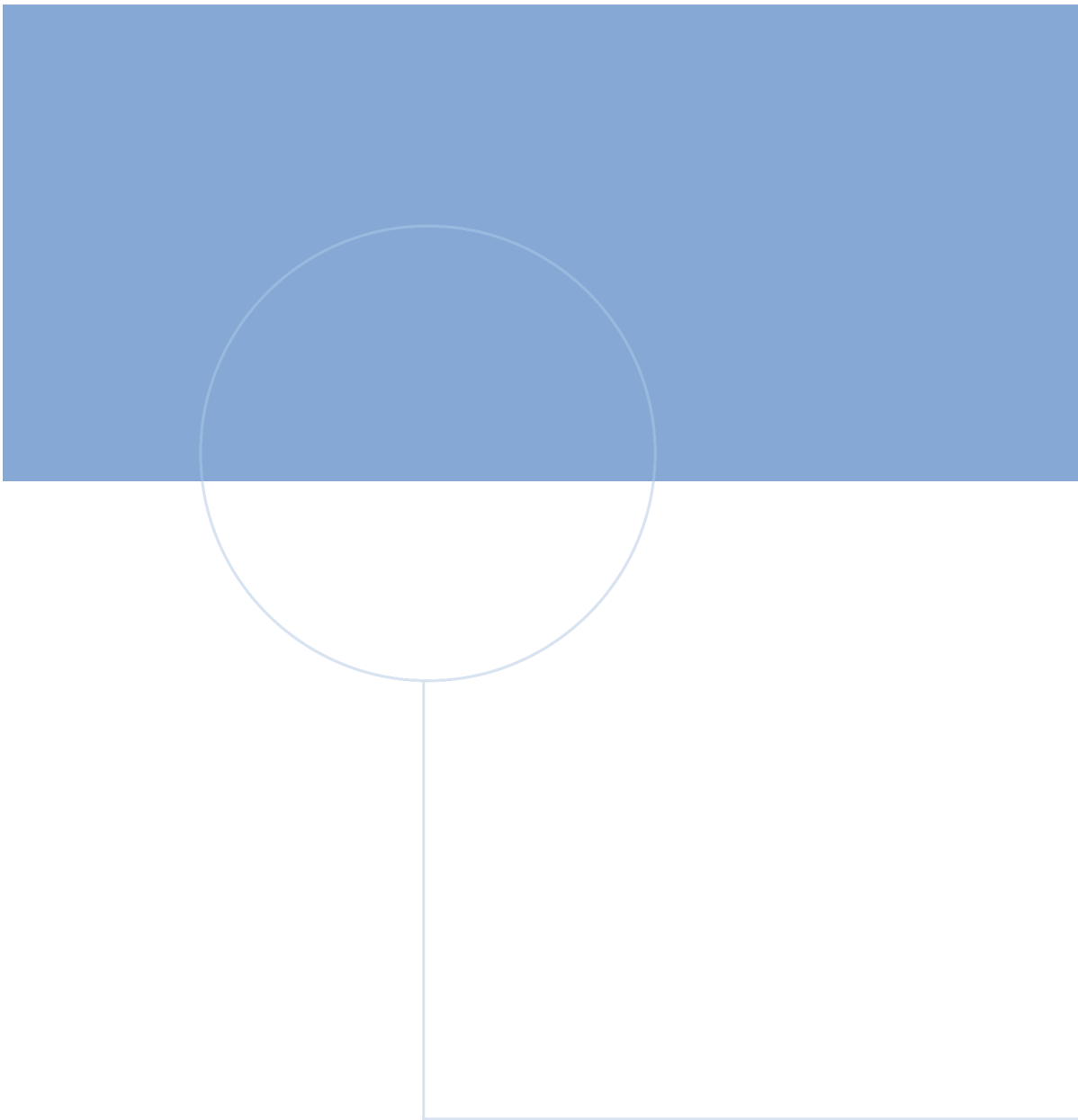
```
#!/usr/bin/env bash
#SBATCH --job-name=build-mfem-mi210q
#SBATCH --partition=mi210q
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --time=03:00:00

MFEM_BUILD_DIR=./build-mi210q
MFEM_INSTALL_DIR=/global/D1/homes/iverh/packages/mi210q/mfem-4.5

set -e
module purge
module use /global/D1/homes/james/ex3modules/mi210q/20221107/modulefiles

module load amd/rocm/5.1.3
module load hypre-32-2.25.0
module load openmpi-4.1.4
module load metis-32-5.1.0

make BUILD_DIR=${MFEM_BUILD_DIR} config \
    MFEM_USE_MPI=YES \
    MPICXX=mpic++ \
    MFEM_USE_HIP=YES \
    MFEM_USE_METIS=YES \
    MFEM_USE_METIS_5=YES \
    HYPRE_LIB="-L${HYPRE_LIBDIR} -lHYPRE" \
    HYPRE_OPT="-I${HYPRE_INCDIR}" \
    METIS_LIB="-L${METIS_LIBDIR} -lmetis" \
    METIS_OPT="-I${METIS_INCDIR}" \
    HIP_ARCH=gfx90a

make BUILD_DIR=${MFEM_BUILD_DIR} -j 4
make BUILD_DIR=${MFEM_BUILD_DIR} check
make BUILD_DIR=${MFEM_BUILD_DIR} install PREFIX=${MFEM_INSTALL_DIR}
```

Code listing B.2: MFEM build script for the `hgx2q` partition on eX^3

```
#!/usr/bin/env bash
#SBATCH --job-name=build-mfem-hgx2q
#SBATCH --partition=hgx2q
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --time=03:00:00
#SBATCH --gres=gpu:1

# Script for building mfem-4.5 on the hgx2q partition of eX3.
# Assumes that it is run inside the root of the mfem source directory.
#
# $ tar -xzvf mfem-4.5.tgz
# $ cd mfem-4.5
# $ # Copy this script in here
# $ sbatch build_mfem_hgx2q.sbatch

MFEM_BUILD_DIR=./build-hgx2q
MFEM_INSTALL_DIR=/global/D1/homes/iverh/packages/hgx2q/mfem-4.5

set -e
module purge
module use /global/D1/homes/james/ex3modules/hgx2q/2022-08-17/modulefiles

module load cuda11.8/toolkit/11.8.0
module load hypre-32-2.25.0
module load openmpi-4.1.4
module load metis-32-5.1.0

make BUILD_DIR=${MFEM_BUILD_DIR} config \
    MFEM_USE_MPI=YES \
    MPICXX=mpic++ \
    MFEM_USE_CUDA=YES \
    MFEM_USE_METIS=YES \
    MFEM_USE_METIS_5=YES \
    HYPRE_LIB="-L${HYPRE_LIBDIR} -lHYPRE" \
    HYPRE_OPT="-I${HYPRE_INCDIR}" \
    METIS_LIB="-L${METIS_LIBDIR} -lmetis" \
    METIS_OPT="-I${METIS_INCDIR}"\
    CUDA_ARCH=sm_80

make BUILD_DIR=${MFEM_BUILD_DIR} -j 4
make BUILD_DIR=${MFEM_BUILD_DIR} check
make BUILD_DIR=${MFEM_BUILD_DIR} install PREFIX=${MFEM_INSTALL_DIR}
```

Code listing B.3: MFEM build script for the `defq` and `milanq` partitions on eX^3

```
#!/usr/bin/env bash
#SBATCH --job-name=build-mfem-defq
#SBATCH --partition=defq
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --time=03:00:00

MFEM_BUILD_DIR=./build-defq
MFEM_INSTALL_DIR=/global/D1/homes/iverh/packages/defq/mfem-4.5

set -e
module purge
module use /global/D1/homes/james/ex3modules/defq/1.0.0/modulefiles

module load hypre-32-2.25.0
module load openmpi-4.1.4
module load metis-32-5.1.0

make BUILD_DIR=${MFEM_BUILD_DIR} config \
    MFEM_USE_MPI=YES \
    MFEM_USE_OPENMP=YES \
    MPICXX=mpic++ \
    MFEM_USE_METIS=YES \
    MFEM_USE_METIS_5=YES \
    HYPRE_LIB="-L${HYPRE_LIBDIR} -lHYPRE" \
    HYPRE_OPT="-I${HYPRE_INCDIR}" \
    METIS_LIB="-L${METIS_LIBDIR} -lmetis" \
    METIS_OPT="-I${METIS_INCDIR}"

make BUILD_DIR=${MFEM_BUILD_DIR} -j 4
make BUILD_DIR=${MFEM_BUILD_DIR} check
make BUILD_DIR=${MFEM_BUILD_DIR} install PREFIX=${MFEM_INSTALL_DIR}
```



NTNU
Norwegian University of
Science and Technology

simula