Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Efficient Application of Accelerator Cards for the Coupling Library preCICE

Timo Pierre Schrader

Course of Study: Informatik

Examiner: Jun.-Prof. Dr. rer. nat. Benjamin Uekermann

Supervisor: David Schneider, M.Sc.

Commenced: October 13, 2022

Completed: April 13, 2023

Abstract

The usage of accelerator cards, mainly graphics processing units (GPU), in scientific and industrial research has been on the rise for years due to their highly data-parallel computational throughput capabilities. Common fields are, amongst other things, machine learning, computational physics, and cryptography.

This thesis investigates the efficient application of GPUs in the multi-physics coupling library preCICE. We look at data mapping methods, which are used to map values between two vertex clouds. More specifically, the focus lies on radial basis function (RBF) interpolation that acts on scattered data points. Solving an RBF interpolation problem requires the solution of mostly large and ill-conditioned systems of linear equations. High computational effort is needed in order to solve these systems, which increases the runtime of preCICE tremendously. We approach this problem by leveraging the high computing power of GPUs.

In order to integrate GPU support into preCICE, we make use of the Ginkgo linear algebra library, which supports multiple data-parallel backends, including Nvidia CUDA, AMD HIP, and OpenMP. It provides solvers and preconditioners for linear systems of equations such as conjugate gradient (CG) and GMRES. Using Ginkgo, we implement an assembly routine for RBF matrices that is up to $100 - 1,000$ times faster than already existing variants in preCICE. We discuss GPU-specific optimization approaches and the resulting efficiency of our implementation approach. The result is a nearly optimal assembly kernel that uses most of the 64-bit compute units on the GPU. Next, we evaluate CG and GMRES, combined with Jacobi and Cholesky preconditioners, on GPUs. The iterative solution approach works well on sparse system matrices, which are the result of RBF kernels with local support and are very competitive to using a very high number of CPU cores. To also provide an efficient way of solving dense systems on GPUs, we additionally implement a QR decomposition using the Nvidia cuSolver library. Our experiments show that using the CUDA QR decomposition on dense system matrices outperforms every other variant including iterative GPU and multi-core CPU solvers as well as single-core solvers by at least a factor of five for larger interpolation problems. As a last step, we investigate a matrix-free RBF solution approach that allows for solving problems of sizes that exceed GPU memory limitations in matrix-based methods.

To summarize our findings, preCICE can highly benefit from the efficient application of GPUs in RBF data mapping routines by being able to solve large interpolation problems much faster; enabling the users of preCICE to run their coupled simulation in less time.

Kurzfassung

Heutzutage sind Grafikprozessoren (GPU) aufgrund ihrer außerordentlichen Rechenleistung in der industriellen und akademischen Forschung nicht mehr wegzudenken. Rechenintensive Disziplinen wie maschinelles Lernen, Computersimulationen und Kryptographie profitieren hierbei von der sehr hohen Datenparallelität, die GPUs aufweisen.

Die vorliegende Thesis untersucht, wie Grafikprozessoren effizient in der preCICE Software genutzt werden können. Die preCICE Bibliothek ermöglicht gekoppelte Multiphysik-Simulationen, indem sie mehrere numerische Löser für verschiedene Aspekte der Simulation koppelt und somit deren Verhalten synchronisieren und Datenaustausch unterstützen kann. Unser Fokus liegt hierbei auf dem Datenmapping in preCICE, welches benötigt wird, wenn berechnete Werte zwischen zwei unterschiedlichen Gittern ausgetauscht werden müssen. Speziell fokussieren wir uns auf Interpolation mit radialen Basisfunktionen (RBF). Dieses Verfahren erfordert das Lösen eines linearen Gleichungssystems, welches meistens sehr groß und schlecht konditioniert ist. Da diese Eigenschaft einen großen Einfluss auf die Laufzeit von preCICE hat, wollen wir untersuchen, ob Interpolation mit RBFs durch GPUs beschleunigt werden kann.

Wir nutzen die lineare Algebra Bibliothek Ginkgo, um GPU Unterstützung in preCICE zu realisieren. Ginkgo bietet für verschiedene Löser und Vorkonditionierer Implementierungen an, die auf Nvidia und AMD GPUs lauffähig sind und ebenfalls mittels OpenMP parallelisiert werden können. Die Matrixassemblierung auf der GPU, welche wir mit der Hilfe von Ginkgo umsetzen, ist um Faktor $100 - 1.000$ schneller als bisherige Varianten in preCICE. Des Weiteren evaluieren wir, wie das Verfahren der konjugierten Gradienten (CG) und das GMRES-Verfahren inklusive Jacobi- und Cholesky-Vorkonditionierer auf GPUs das Datenmapping beschleunigen können. Unsere Experimente zeigen, dass diese Verfahren besonders auf dünn-besetzten Systemen gut funktionieren und mit hoch parallelisierten CPU Implementierungen mithalten können. Darüber hinaus implementieren wir eine QR Zerlegung mittels der cuSolver Bibliothek. Auf dicht-besetzten Matrizen ist diese Implementierung zeitlich um mindestens einen Faktor fünf den anderen Implementierungen überlegen. Abschließend evaluieren wir eine matrixfreie Implementierung, welche es erlaubt, besonders große Interpolationsprobleme auf der GPU zu lösen, die ansonsten nicht in den verfügbaren Speicher der GPUs passen würden.

Das Fazit dieser Arbeit ist, dass preCICE stark von Grafikprozessoren im Datenmapping in Bezug auf die Laufzeit profitieren kann. Speziell die großen Interpolationsprobleme können hierbei deutlich schneller auf GPUs gelöst werden als auf CPUs.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 The Coupling Library preCICE . . . 1
1.3 Goals of this Thesis . . . 3
1.4 Outline of this Thesis . . . 4

2 Theoretical Background **5**
2.1 Overview on Parallel Computing . . . 5
2.2 Accelerator Cards and GPGPU Programming . . . 7
2.3 RBF Interpolation . . . 10
2.4 Solver for Systems of Linear Equations . . . 13
2.5 Related Work . . . 18

3 Implementation Details **21**
3.1 Requirements . . . 21
3.2 Frameworks . . . 22
3.3 Integration into preCICE . . . 24
3.4 RBF Interpolation Components . . . 27

4 Experimental Evaluation **41**
4.1 Simulation Setup and Test Cases . . . 41
4.2 Turbine Blade Grid Mappings . . . 42
4.3 Continuous Coupling in 2D Heat Simulation . . . 56
4.4 Discussion . . . 62

5 Conclusion and Outlook **65**
5.1 Summary and Conclusion . . . 65
5.2 Outlook . . . 66

Bibliography **69**

A LU and Cholesky Decomposition on GPUs **73**

List of Figures

1.1 Overview of the functionality of preCICE (Chourdakis et al., 2022). preCICE couples different solvers by providing communication channels, data mapping routines, different coupling schemes, and time-stepping methods. 2
1.2 Conceptual overview of the mapping from mesh M_1 to M_2 using RBFs where each vertex of M_1 influences the value of the vertex of M_2 via a weight w_i and a function value φ_i. Arrows are drawn bi-directional to emphasize the influence of the distance. 3

2.1 Architecture overview of Nvidia GPUs. Every SM, which consists of multiple streaming processors, has its unique shared memory space. Global memory is used to share data across all SMs. 8
2.2 Thread hierarchy model of CUDA. Every thread (denoted as t) is contained within one block. A global grid contains all blocks. Every thread and block can be positioned in a 1D, 2D, or 3D coordinate system. 9

3.1 The unified kernel mechanism of Ginkgo takes one kernel and compiles it for different parallel backends. 24
3.2 The UML class diagram of the RBF mapping in preCICE. Our implementation is embedded into the `RadialBasisFctMapping` class. 25
3.3 Conceptual thread distribution for matrix assembly; each color represents a distinct thread that is responsible for a unique φ. 28
3.4 Mesh vertex coordinates are stored in a dense matrix. The thread (i, j) (depicted in green) accesses all three dimensions of v_i and v_j. 29
3.5 Optimized vertex layout for coalesced GPU usage. First, all red elements are accessed in parallel, then all green ones, and finally the blue ones. 30
3.6 A matrix-vector multiplication using a GPU as it is used in the matrix-free implementation. Having one thread per matrix row keeps write accesses to b as efficient as possible and allows for coalesced write transactions. 36
3.7 Reducing global memory transactions is possible by calculating every part of b locally and writing the result only once to global memory. 38

4.1 3D model geometry of the turbine blade provided by ASTE. It is used to generate the meshes for the turbine mapping test cases. 42
4.2 Roofline diagram of the matrix assembly kernel. The kernel is compute-bound and reaches almost the maximum performance of the FP64 compute units. 44
4.3 Memory analysis of the matrix assembly kernel. Whereas the L1 cache hit rate is low, the L2 cache hit rate is almost 100% and therefore optimal. 45
4.4 Assembly time of A_M for the meshes of *fine* series depending on mesh width h in ms (log-scale). The CUDA assembly kernel is the fastest out of all implementations. 46

4.5 Assembly time of Φ for the meshes of the *coarse* series depending on mesh width h in ms (log-scale). The CUDA kernel is the fastest version for all mesh sizes smaller than $h = 0.01$. 47
4.6 Assembly time of A_M for the meshes of the *fine* series depending on mesh width h in ms using CUDA unified memory (log-scale). The memory allocation of the CUDA unified memory model causes significant overhead. 48
4.7 Assembly time of matrices of the *coarse* series depending on mesh width h in ms (log-scale). The AMD Radeon Pro VII performs better than the Nvidia RTX 3090 by almost a factor of ten. 49
4.8 Results for compact polynomial C6 RBF. The left plot shows the initial setup phase duration in ms and the right side the recurring mapping time. 50
4.9 Results for compact polynomial C6 RBF. The left plot shows the overall runtime of preCICE in ms and the right plot the average discrete mapping error. 50
4.10 The left plot shows the number of iterations until the stopping criterion of 10^{-9} was reached for different iterative solver and preconditioner combinations. GMRES without a preconditioner did not hit the stopping criterion at all. The right plot displays the time it took preCICE to map the data. Preconditioners have a positive impact on the runtime for finer meshes. 51
4.11 Results for C-TPS C2. The left plot shows the initial setup phase duration in ms and the right side the recurring mapping time. 53
4.12 Results for C-TPS C2. The left plot shows the overall runtime of preCICE in ms and the right plot the average discrete mapping error. 53
4.13 The roofline diagram of our matrix-free kernel which combines matrix assembly and matrix-vector multiplication. This kernel achieves a higher computational intensity than the assembly kernel and is also compute-bound. 55
4.14 The initial setup phase time is shown on the left side and the recurring mapping time on the right. Whereas PETSc requires the least time for the three coarsest meshes of the *fine* series, only our matrix-free implementation is able to calculate a solution in a feasible amount of time. 55
4.15 The left plot shows the global runtime of preCICE, the right one the mapping error. The mapping error is almost equal between all three approaches. 56
4.16 The two coupled domains of our two heat solvers. The coupling is carried out along the common coupling interface, depicted by the red line. (Chourdakis et al., 2022) 57
4.17 Coupling setup in our coupled 2D heat equation simulation. 58

5.1 A decision strategy that helps to decide whether using a GPU for RBF interpolation is the best choice. The most important factors are problem size and sparsity of the system matrix Φ. 66

A.1 The time that it takes to decompose the system matrix and solve the resulting triangular system using LU and Cholesky on CUDA. It can be seen that Cholesky needs much more time than any other solver tested in this thesis. 73

List of Tables

2.1 The three different BLAS levels and their computational complexity. Each level corresponds to a different set of linear algebra operations. However, higher levels can be composed of operations of lower levels. 6

4.1 The different turbine blade meshes provided by artificial solver testing environment (ASTE). They differ in mesh size h. 43

4.2 Runtime results of the coupled heat solver using Eigen for calculating the RBF mapping. Especially the setup phase takes a very long time. 60

4.3 Runtime results of the coupled heat solver using the CUDA QR decomposition for calculating the RBF mapping. Initializing preCICE, i.e., factorizing the system matrix, is much faster than using Eigen. 61

4.4 Runtime results of the coupled heat solver using the CUDA QR decomposition in preCICE and running both solvers on the CPU. Especially freeing VRAM consumes a lot of time. However, most time is spent on solving PDEs on the CPU. 61

4.5 Runtime results of the coupled heat solver using GMRES on a GPU for calculating the RBF mapping. Since there is no matrix factorization involved, setup phase is faster than before, the recurring mapping computation takes more time. 62

List of Listings

3.1 Optimized assembly kernel that allows for coalesced memory accesses to the vertex arrays and uses FMA. 32
3.2 The QR decomposition that is implemented using cuSolver. Its function names are derived from LAPACK functions. We omit function arguments for better readability. 35
3.3 The matrix-free kernel that combines matrix assembly and matrix-vector multiplication. It is executed every time the `apply()` function is called on the object. . . 37

List of Algorithms

2.1 RBF Interpolation Procedure for Separate Polynomial 12
2.2 Householder QR Decomposition (taken from Meister (2015)) 15
2.3 CG Algorithm (taken from Schulte (2022)) 16
2.4 GMRES Algorithm (taken from Meister (2015)) 17

Acronyms

AMG algebraic multi-grid. 18

ASTE artificial solver testing environment. ix

BLAS basic linear algebra subprograms. 6

C-TPS compact thin plate spline. 52

CG conjugate gradient. 16

CPU central-processing unit. 1

CUDA Compute Unified Device Architecture. 8

FLOP floating-point operation. 44

FMA fused multiply-add. 29

FSI fluid-structure interaction. 2

G-TPS global thin plate spline. 46

GMRES generalized minimal residual method. 16

GPGPU general purpose computation on graphics processing unit. 4

GPU graphics processing unit. 1

HIP heterogeneous interface for portability. 9

HPC high-performance computing. 1

MPI Message Passing Interface. 10

PDE partial differential equation. 2

RAM random access memory. 7

RBF radial basis function. 3

s.p.d. symmetric positive-definite. 16

SM streaming multiprocessor. 7

VRAM video random access memory. 7

xSDK Extreme-scale Scientific Software Development Kit. 1

1 Introduction

1.1 Motivation

The importance of high-performance computing (HPC) and the demand for supercomputers have been increasing significantly in recent years in industry and academic research alike. For example, Nvidia leverages the power of up to 3.072 Nvidia A100 graphics processing units (GPUs) to train their Megatron language model[1]. GPUs are widely used for machine learning-related tasks nowadays, however, their application spans across a broad range of research areas. A very important area is the field of numerical simulation. Due to the very high computational workload, simulations heavily rely on HPC too. Common use cases include, amongst others, computational fluid dynamics and fluid-structure interaction simulations. The TOP500[2] list of supercomputers also reflects the continuously growing effort that goes into developing even more powerful HPC systems. According to it, the most powerful system is hosted at the Oak Ridge National Laboratory, Tennessee, USA, and achieves an HPL benchmark[3] performance of over one Exaflop/s, i.e., 10^{18} floating point operations per second as of June 2022. This is an increase in computational power of over 700% compared to the best system in June 2019 from IBM with 148.60 Petaflop/s and indicates that there is massive research and funding in the field of HPC.

This development is highly driven by the application of accelerator cards. Accelerator cards, mainly GPUs nowadays, provide a throughput-oriented architecture that focuses on processing as much data in parallel as possible by leveraging thousands of compute units. Typically, they are not used for the same tasks that central-processing units (CPUs) are used for, but rather complement them by focusing on highly data-parallel operations as they provide a much larger amount of cores with lower per-core performance. Being able to incorporate GPUs in scientific computations provides a huge potential for significant speedups.

1.2 The Coupling Library preCICE

In this thesis, we address the efficient application of such accelerator cards in the multi-physics coupling library preCICE (*Precise Code Interaction Coupling Environment*, Chourdakis et al. (2022))[4] which is part of the "Extreme-scale Scientific Software Development Kit (xSDK)"[5], an ecosystem for extreme-scale scientific software. It is an open-source library written in C++ . The

[1] https://github.com/NVIDIA/Megatron-LM
[2] https://www.top500.org/
[3] https://netlib.org/benchmark/hpl/
[4] https://precice.org/
[5] https://xsdk.info/

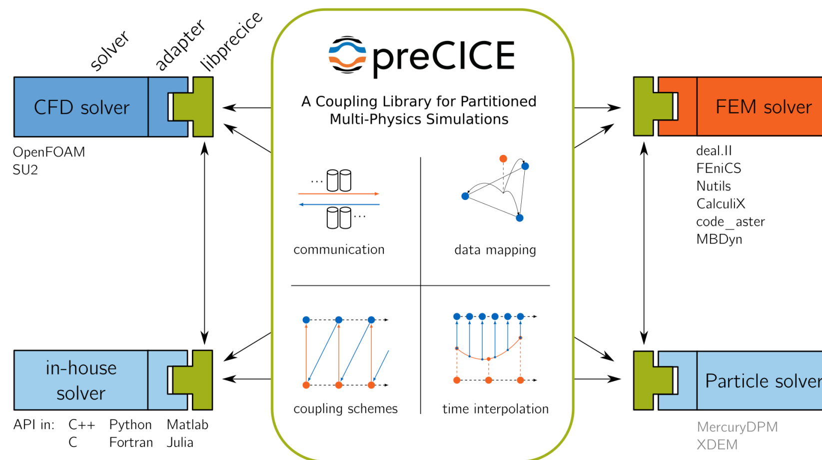

Figure 1.1: Overview of the functionality of preCICE (Chourdakis et al., 2022). preCICE couples different solvers by providing communication channels, data mapping routines, different coupling schemes, and time-stepping methods.

preCICE library couples multiple numerical solvers, which are used for simulation purposes, in a black-box fashion, i.e., it only receives data from each solver that is necessary for other solvers in order to perform their part of the simulation. Hence, its main purpose is to distribute calculated values to every required participant and steer the communication between solvers and their time-stepping behavior. This enables the solvers to exchange data with each other, be accelerated when implicitly coupled, and use data mapping methods to exchange calculated values more closely tied to the discretization mesh of each individual solver. A solver needs to work with the preCICE API in order to be coupled since preCICE works as a "man-in-the-middle" library (cf. Figure 1.1). Moreover, preCICE already offers out of the box adapters[6] for commonly used solvers such as OpenFOAM[7] and deal.II[8]. Uekermann et al. (2017) write about the concept of adapters in preCICE in detail. To summarize, adapters provide an easy way of coupling already existing solver code with widely used, well-known solver libraries. These adapters support the users of preCICE as they do not have to spend much time writing their own coupling code for common libraries themselves.

The preCICE library has been used in many research projects in the past. Kandekar et al. (2023) present a partitioned computational framework for simulating the interaction between corrosion and mechanical stresses that can cause failure of material. They use preCICE to couple two solvers, one for the corrosion process and one for the mechanical stresses. Another use case is presented by Naseri et al. (2020). They use preCICE for coupling of partitioned fluid-structure interaction (FSI) simulations.

The focus of this thesis is on the data mapping part of preCICE. Partial differential equations (PDEs), which naturally arise in many real-world simulations, are numerically solved by first discretizing the problem domain. Discretization points are modeled as vertices in preCICE. These vertices form a "mesh" and can be connected to each other s.t. they form triangles or other geometries. A mesh with

[6] https://precice.org/adapters-overview.html
[7] https://www.openfoam.com/
[8] https://www.dealii.org/

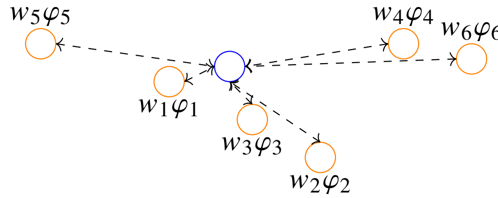

Figure 1.2: Conceptual overview of the mapping from mesh M_1 to M_2 using RBFs where each vertex of M_1 influences the value of the vertex of M_2 via a weight w_i and a function value φ_i. Arrows are drawn bi-directional to emphasize the influence of the distance.

a triangle topology is shown in the top right corner of Figure 1.1. Instead of calculating an analytic solution for these PDEs, which is mostly infeasible or even impossible, approximate solutions are calculated by only computing numerical approximations on discrete elements. Hence, the accuracy of the simulation highly depends on the number of unknowns. Accuracy, but also computational intensity increase with the number of unknowns. Since different solvers, which are coupled via preCICE, target different problems of the simulation, separate meshes are involved with unique mesh sizes h and geometries. When it now comes to exchanging data between different meshes, e.g., calculated forces and displacements in a FSI simulation, data mapping must be performed to compensate for non-matching meshes and vertices. Figure 1.2 visualizes a basic case where data is mapped from a mesh M_1 to M_2 using radial basis function (RBF) data mapping. The orange vertices are the input vertices provided by M_1. The value of the blue vertex of M_2 is calculated by taking a weighted sum of the values of all orange vertices since RBF interpolation takes all support points into account, which is described in detail in Sec. 2.3.1. Since no topological information is given in this mesh, RBF interpolation works on an unstructured "cloud" of vertices. Besides RBF interpolation, there are other mapping methods in preCICE, namely "nearest-neighbour" and "nearest-projection" mapping.

1.3 Goals of this Thesis

As already described above, one key component is data mapping which describes the process of mapping calculated scalars or vectors from one mesh, which consists of vertices and optionally edges, to an output mesh which most often is made up of another set of vertices. Due to the data-parallel nature of data mapping methods on discretized grids, there is a huge potential for massive speed-ups by using GPUs. One commonly used method is RBF interpolation, which is very expensive in terms of computational complexity and memory requirements. Mapping data using RBF interpolation includes solving a system of linear equations in order to obtain the interpolation coefficients. This thesis investigates how RBF interpolation can profit from the usage of GPUs in order to significantly speed up the interpolation process in preCICE. RBF interpolation itself is already available in preCICE in two different implementation approaches: a sequential single-core one and one that addresses multi-node distributed memory environments. The major research questions in this thesis are:

- How can different components of RBF interpolation utilize accelerator cards?
- Which solver algorithms work well? Which preconditioners can be applied?

- How well is the hardware utilized?
- How can portability across different parallel backends be achieved?
- How can the problem of memory restrictions on GPUs be handled?
- How efficient is a coupling approach in which both solvers and preCICE make use of GPUs?

In order to realize GPU support in preCICE, we make use of the linear algebra library Ginkgo. This allows preCICE to execute RBF data mapping on Nvidia and AMD GPUs as well as multi-threaded environments using OpenMP.

We show that assembling the RBF matrices on the GPU and solving these interpolation systems using both direct and iterative solvers can significantly improve the runtime of preCICE by factor of five and higher, depending on the properties of the interpolation problem, while keeping mapping errors in the same range as the already available implementations in preCICE. Starting from this implementation, which relies on traditional matrix-based methods, we further investigate the usage of so-called matrix-free methods. Our matrix-free implementation addresses problems that are too large to otherwise fit into the available memory of GPUs and therefore enables preCICE to solve even bigger interpolation problems on GPUs. We demonstrate that this implementation is able to obtain a mapping for very large problems in a feasible amount of time, as opposed to other approaches. Not only does our highly competitive GPU implementation outperform the already existing variants in various cases, but it also complements the already existing implementations by providing the opportunity to run on separate accelerator cards.

1.4 Outline of this Thesis

This thesis is structured as follows: Chapter 2 gives an introduction to parallel computing and general purpose computation on graphics processing unit (GPGPU). In addition, it provides the mathematical background relevant to this work. Chapter 3 starts with a detailed overview of different GPU programming frameworks and their suitability for application in the preCICE library. Furthermore, it depicts the state of RBF interpolation in preCICE. Afterward, implementation details and integration into preCICE are elaborated. The performance evaluation and analysis of these GPU-accelerated algorithms is done in Chapter 4. Chapter 5 concludes this thesis by summarizing the results of this work and providing an outlook for potential follow-up work.

2 Theoretical Background

This chapter deals with the theoretical background necessary for this thesis. First, we provide an introduction to different types of parallel computing architectures with a special focus on GPGPU programming. Second, we explain the necessary mathematical basics. Furthermore, we focus on how these algorithms can be parallelized on massively data-parallel hardware accelerators. This chapter is concluded with an overview of related work that deals with the acceleration of numerical algorithms on graphics cards.

Throughout this thesis, we denote every matrix that is specific to the data mapping by the subscript M if it could be confused with matrices that are standard notions in mathematics.

2.1 Overview on Parallel Computing

Parallel computing refers to the concurrent execution of multiple tasks by dispatching them onto multiple execution units such as CPU cores or GPUs. We introduce two of the most important concepts relevant to this thesis. They allow us to judge if an algorithm is suited to be executed on GPUs.

Flynn's Classification The term *Flynns' classification* refers to a descriptive overview of different parallel architectures that has been first published by Flynn (1972). Nowadays, there are mainly four categories with unique properties:

- **Single-Instruction Stream - Single-Data Stream/SISD**: This category refers to the single-threaded execution of a single instruction.
- **Single-Instruction Stream - Multiple-Data Stream/SIMD**: This describes a single instruction that is performed on multiple data points simultaneously. Vector processing units such as AVX on modern x86 chips are a popular example of this category.
- **Multiple-Instruction Stream - Single-Data Stream/MISD**: According to Flynn (1972), this type describes machines that include streaming organizations that execute a series of instructions on the same data point repeatedly.
- **Multiple-Instruction Stream - Multiple-Data Stream/MIMD**: This term classifies multi-core processors working on different tasks including different sets of data points concurrently.

Although this classification was formulated with respect to CPUs, it is also applied to the area of GPUs. Lindholm et al. (2008) have introduced the term *SIMT* (*Single Instruction - Multiple Threads*) after Nvidia presented the Tesla architecture in 2006. The remarkable similarity to *SIMD* is no coincidence: *SIMT* combines the concept of *SIMD*, i.e., performing a single instruction on

multiple data points, with the multi-threaded execution model, i.e., performing these instructions concurrently in multiple threads. This furthermore summarizes the main concept of GPUs, which we discuss in more detail in Sec. 2.2.1.

BLAS The term "basic linear algebra subprograms (BLAS)" refers to a set of linear algebra routines that were specified by Lawson et al. (1979) for the scientific programming language FORTRAN. These routines include mathematical operations such as dot product (**DOT**), vector additions (**AXPY**), and general matrix multiplication (**GEMM**). Furthermore, these routines can be classified into three categories as shown by Table 2.1.

Level	Operation	Comp. Complexity
1	Vector-Vector	$O(N)$
2	Matrix-Vector	$O(N^2)$
3	Matrix-Matrix	$O(N^3)$

Table 2.1: The three different BLAS levels and their computational complexity. Each level corresponds to a different set of linear algebra operations. However, higher levels can be composed of operations of lower levels.

These three BLAS levels are not disjoint from each other. For example, a matrix multiplication between two different matrices is composed of many multiplications of row and column vectors. Moreover, the multiplication of a row vector and a column vector represents a dot product. In conclusion, BLAS routines can be also a composition of lower-level BLAS routines. Hence, it is important to not only look at BLAS routines used in an algorithm but also how they leverage different levels.

BLAS routines are of special interest since they offer potential for parallelization. For instance, a dot product between two vectors $u, v \in \mathbb{R}^n$ is defined as $\langle u, v \rangle = \sum_{i=1}^{n} u_i \cdot v_i$. There are n independent products, one for each dimension. Having n calculation units, this results in a runtime complexity of $O(1)$. The final summation produces a so-called "fan-in" process in which groups of two elements are summed up by a different thread. However, after every run, the remaining number of summands is divided by two, i.e., the degree of parallelization decreases continuously. Still, **DOT** as well as other BLAS routines can make use of parallelization techniques such as multi-threading.

We mainly focus on the following BLAS routines in our theoretical analysis provided in Sec. 2.4:

- **DOT** (Level 1): $z = \langle x, y \rangle$: This is the dot product between two vectors x and y, which includes a "fan-in" summation process.
- **AXPY** (Level 1): $z = \alpha \cdot x + y$: This is the (scaled) sum of two vectors.
- **GEMV** (Level 2): $z = \alpha \cdot A \cdot x + \beta \cdot y$: This reflects a general matrix-vector multiplication with additional scaling factors.

2.2 Accelerator Cards and GPGPU Programming

2.2.1 High-Level GPU Architecture

Historically, GPUs have been developed for repeatedly rendering individual pixels on the screen. Since computer screens consist of millions of individual pixels nowadays, of which each one must pass through the rendering pipeline, GPUs were originally designed for highly data-parallel operations. Uelschen (2019) provides an overview on the high-level architecture of modern Nvidia GPUs.

Compute Units On the top level, there are multiple "graphical processing clusters (GPC)" on a GPU. These GPCs comprise the main compute units called "streaming multiprocessor (SM)". A SM contains a control unit for processing instructions on single cores called "streaming processor" or "CUDA cores". Every instruction on a SM is distributed to each SP contained in it. This architecture induces the high data-parallel processing power of GPUs. For reference, a modern Nvidia A100 GPU[1], released in 2020, has 6912 CUDA cores in total. This is more than 100 times as many cores compared to modern state-of-the-art server CPUs such as the AMD EPYC 7763[2], which is made up of 64 physical cores. Still, these cores are bound to the *SIMT* paradigm explained in Sec. 2.1. As a result, in order to take advantage of these accelerator cards, we need to think of data-parallel operations s.t. as many SMs and CUDA cores can work in parallel as possible.

Figure 2.1 depicts a GPC that contains multiple SMs and CUDA cores.

Memory The GPU compute units have access to a separate memory space called video random access memory (VRAM). VRAM is structured hierarchically. Figure 2.1 visualizes the different memory areas and how they are distributed on a GPU.

Each SM uses one distinct *shared memory* unit. Within a single SM, each CUDA core uses the same shared memory space. This implies that data can be easily shared among CUDA cores as long as they are part of the same SM. In general, shared memory has lower latency than *global memory*. Besides global and shared memory space, there are also registers, which are comparable to their CPU counterpart, and L1/L2 caches. However, every data point which ought to be processed on the GPU must be transferred from random access memory (RAM) to VRAM first which induces runtime overhead, i.e., algorithmic problems should focus on leveraging as much compute units as possible.

We use these technical details in Chapter 3 to discuss how we can efficiently make use of a large number of cores and the different memory hierarchies.

[1] https://www.nvidia.com/de-de/data-center/a100/

[2] https://www.amd.com/de/products/cpu/amd-epyc-7763

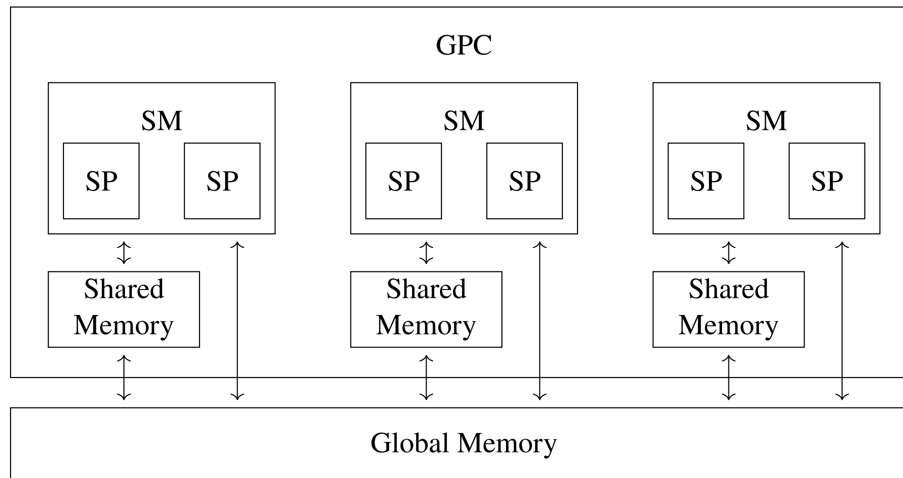

Figure 2.1: Architecture overview of Nvidia GPUs. Every SM, which consists of multiple streaming processors, has its unique shared memory space. Global memory is used to share data across all SMs.

2.2.2 Nvidia CUDA and AMD HIP

Whereas in the beginning it was only possible to leverage GPUs through shader programming, it is nowadays very common to make use of general purpose GPU programming frameworks. The probably most commonly known is the "Compute Unified Device Architecture (CUDA)" parallel computing platform introduced by Nvidia Corporation (2006). It aims towards the massively data-parallel execution paradigm of GPUs by allowing to write code that is close to the C++ language. Additionally, it supports some standard library features of C++ that can be embedded into CUDA code. This makes it possible to use GPU accelerator cards as coprocessors to the CPU.

The CUDA terminology differentiates between CPU and GPU by referring to them as "host" and "device". Likewise, memory spaces are divided into "host memory", which describes RAM, and "device memory", which covers the VRAM that is manufactured onto the GPU board.

Since GPUs address a multi-threaded execution paradigm, CUDA code must always be written with thread parallelism in mind. Referring to Figure 2.1, the CUDA execution model is tightly coupled to the hardware architecture of such accelerator cards. It builds upon a highly sophisticated thread model in which each thread is part of a hierarchy and responsible for the same kind of task as any other thread. This is a fundamental difference to POSIX threads for example, known from UNIX operating systems. Whereas a POSIX thread refers to a subprocess, likely to be responsible for an independent branch of the process, a CUDA thread refers to one part of an entire group in which each thread is responsible for executing almost the same operations, only on different data points. This is why it is called data-parallel and therefore related to the *SIMD* category explained above. CUDA threads are spawned whenever a *kernel* is launched. A kernel is a function that is asynchronously dispatched onto the GPU. Its code is executed by the number of threads configured beforehand. CUDA employs the thread system shown in Figure 2.2. Every thread is contained within a group called block. These blocks are distributed among different SMs during execution. Therefore, every thread within a block has access to the same shared memory space. All blocks together form a 1D, 2D, or 3D grid of blocks. Every block in the grid gets its unique identifier. On

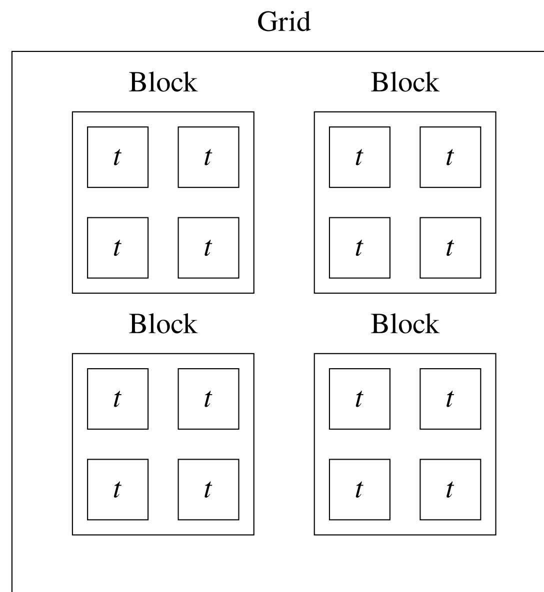

Figure 2.2: Thread hierarchy model of CUDA. Every thread (denoted as t) is contained within one block. A global grid contains all blocks. Every thread and block can be positioned in a 1D, 2D, or 3D coordinate system.

top, every thread gets an identifier within a block. This enables every thread of a kernel to calculate its own position in x and optionally y and z direction within the grid, depending on whether the grid was configured to be 1D, 2D, or 3D.

Cheng (2014) provide a very detailed introduction to high-performance CUDA programming that the interested reader may refer to.

A few years ago, AMD presented a counterpart called "heterogeneous interface for portability (HIP)" tailored for their own Radeon GPUs. It sticks to almost the same concepts as CUDA with the option to compile HIP code for both, Nvidia and AMD GPUs.

In this work, we aim for supporting the accelerator cards of both vendors.

CUDA Unified Memory "CUDA unified memory" is a memory management technique offered by newer CUDA version. "Unified memory" is used to create a pool of managed memory, i.e., a memory address in this pool is a valid pointer for both, host and device. As a result, these memory locations can be used from the host and device side simultaneously. The CUDA runtime environment automatically migrates data when requested from the host or device if it is not available there yet. On one hand, this simplifies memory management as no explicit memory transfers in the code are required. Moreover, it enables the allocation of more memory than provided by the GPU. On the other hand, repeatedly copying memory from and to the GPU creates overhead. Well-optimized memory management has the potential to outperform "unified memory".

2.2.3 CPU Alternatives

The concept of thread- and data-parallel programming is not restricted to GPUs only. One way of achieving thread-parallel execution on a large amount of data is by using OpenMP (OpenMP Architecture Review Board et al., 2021). OpenMP targets multi-core systems, i.e., the more cores a CPU has the better the parallelization achievable through OpenMP. Adding OpenMP to existing code is made possible through preprocessor directives by tagging routines such as `for`-loops with the corresponding `#pragma omp for` directive. This forces the compiler to generate a thread-parallel `for`-loop that is executed by N different threads in parallel, working on distinct items. Not only do we consider GPU frameworks in this thesis, but also OpenMP. Hence, we do compare CUDA and HIP against OpenMP in Chapter 4. This allows us to draw conclusions about the possible speedup by using accelerator cards over CPUs when running almost the same code.

There is also a widely used standard for parallelism that addresses distributed memory environments, namely "Message Passing Interface (MPI)". It allows network nodes to communicate data structures such as arrays over the network through different broadcasting mechanisms such as "scatter" and "gather". Every participant in a MPI environment does hold a rank that works as a unique identifier in the network. MPI is available in preCICE through the PETSc linear algebra library. Thus, we additionally compare the accelerator cards against node-level parallelism in the context of data mapping methods.

2.3 RBF Interpolation

Since this work deals with RBF interpolation problems and systems of linear equations, we now define relevant mathematical concepts that we use throughout this thesis.

2.3.1 RBF Interpolation

Interpolation problems are a widely studied topic in the field of numerical analysis. Dahmen and Reusken (2022) define the interpolation problem as follows:

Definition 2.3.1
Given a set of $n+1$ unique points $\{x_0, \dots, x_n\}, x_i \in \mathbb{R}$ and corresponding function values $\{f_0, \dots, f_n\} \in \mathbb{R}$, the goal is to find a function $g : \mathbb{R} \mapsto \mathbb{R}$ s.t. the following condition holds:

$$g(x_i) = f_i \quad \forall i = 0, \dots, n \tag{2.1}$$

Interpolation based on RBFs is usually applied when dealing with scattered data. For instance, preCICE uses a black-box approach. When it comes to mapping values between non-matching meshes, we deal with scattered data within a grid, often without knowing how nodes in the grid are connected to each other. Therefore, RBF interpolation comes in handy. RBFs are radially-symmetric functions centered around a point $x \in \mathbb{R}^d$ with their extremum being located at the center x:

$$\varphi_x : \mathbb{R}^d \to \mathbb{R}, y \mapsto \varphi_x(||y - x||) \tag{2.2}$$

$||\cdot||$ denotes a norm on a vector space, usually the Euclidean norm.

For a specific point y, interpolation is done by calculating a weighted sum of all RBF kernels with y being plugged in s.t. that the distance between y and the center point of the corresponding RBF kernel determines its function value. Given a point y_i, its interpolated function value g_i is calculated as follows:

$$g_i = \sum_{j=1}^{K} \lambda_j \cdot \varphi_{x_j}(||y_i - x_j||) \tag{2.3}$$

Each summand φ_j is weighted by λ_j. The weight vector λ is the central part in RBF interpolation. It is calculated by setting up an interpolation system using K center points and their corresponding function values. For a system with K center points, we get the following K interpolation conditions that result in a system of K linear equations:

$$f_i = \sum_{j=1}^{K} \lambda_j \cdot \varphi_{x_j}(||x_i - x_j||) \quad \forall i = 1...K \tag{2.4}$$

In matrix notation:

$$f = \Phi \cdot \lambda \tag{2.5}$$

RBF interpolation always takes all pair-wise distances into account.

We now denote the matrix A_M as the output interpolation matrix used to map the values. A_M is constructed similarly to Φ. However, instead of evaluating the RBF between vertices of one single vertex set, A_M uses radii between the target vertices, for which the function value g shall be calculated, and the vertices used during the calculation of λ. As the amount and coordinates of support vertices and target vertices usually differ, A_M is a non-square and non-symmetric matrix.

The interested reader may refer to Lewis et al. (2010) for further information and historical background on RBF interpolation and its application on scattered data as well as Fasshauer (2007) for a deeper introduction to RBF interpolation in so-called "meshfree" problems which arise when obtaining a mesh representation is too difficult. In Chapter 3, we discuss how the computationally intense assembly of Φ and A_M can be accelerated using GPUs.

Lindner et al. (2017) elaborate an extension of this concept by adding a polynomial interpolant to the RBF system, which builds upon the work of Fasshauer (2007):

$$f(x) = \sum_{j=1}^{K} \lambda_j \cdot \varphi_{x_j}(||x - x_j||) + \beta_0 + \beta^T \cdot x; \quad \beta \in \mathbb{R}^d \tag{2.6}$$

The polynomial addition represents the linear components of the data exactly. Additionally, to regularize the resulting over-determined system, the following conditions are added to the problem:

$$\sum_{j=1}^{K} \lambda_j = 0; \qquad \langle \lambda, x_j \rangle = 0 \quad \forall j = 1, \dots, d \tag{2.7}$$

The authors present two ways of handling this additional polynomial contribution, namely **integrated** and **separate** polynomial, out of which we only consider the latter in this work due to better numerical properties.

The integrated variant constructs a larger system matrix Φ with the additional constraints as well as the polynomial contribution added to the original interpolant.

The separate polynomial method refers to treating the polynomial coefficients separately by solving an additional system of linear equations and subtracting the result from the original system. This is not mathematically equivalent to the integrated polynomial variant. Algorithm 2.1 shows the steps involved in this RBF interpolation variant.

Algorithm 2.1 RBF Interpolation Procedure for Separate Polynomial

1: Solve least squares problem for $\beta : Q_M \cdot \beta = f$
2: Solve modified system for $\lambda : \Phi \cdot \lambda = f - Q_M \cdot \beta$
3: Calculate interpolation values $g : g = A_M \cdot \lambda + V_M \cdot \beta$

$Q_M \in \mathbb{R}^{K \times (d_1+1)}$ and $V_M \in \mathbb{R}^{K \times (d_2+1)}$ are the polynomial matrices. Every row in these matrices consists of d values $x_1, \dots, x_d$, i.e., every entry is a copy of the coordinate value of the corresponding dimension, plus an additional 1 for the constant polynomial.

RBF Kernels in preCICE There are two categories of RBFs implemented in preCICE: one with local and one with global support. Local support means that an RBF evaluates to zero if the radius between two points is larger than a predefined threshold. This threshold can be freely configured in preCICE for every local RBF. There is no golden rule for getting the best support radius, hence it is up to the user to find one which works well. Small support radii produce sparse Φ. On one hand, this leads to better convergence behavior of iterative solvers. On the other hand, mapping accuracy might be decreased. RBFs with a global support radius do not use this threshold, but instead work with every input radius, even very large ones. They take a shape parameter upon initialization instead of a support radius. This produces dense system matrices that lead to a higher mapping accuracy but worse numerical conditioning of the system.

Gaussian RBF, which is a widely used function, e.g., in the field of machine learning, is treated as both local and global in preCICE by being able to use both support radius and shape parameter.

Throughout this thesis, we provide the formulas of each RBF that is used for experimental evaluation. A complete overview of all RBFs in preCICE is provided by Chourdakis et al. (2022).

2.4 Solver for Systems of Linear Equations

Since the focus of this thesis lies on parallelization using GPUs, it is also important to look at how linear systems of equations of the form $Ax = b$ can be solved in a data-parallel fashion.

Direct Solvers Direct solvers calculate a solution of a well-defined system of linear equations by factorizing the system matrix. A resulting triangular matrix is then used to sequentially compute each entry of the solution vector x. Direct methods are able to calculate an exact solution if the system of linear equations is well-defined. Throughout this thesis, the most important algorithms are the LU decomposition, which factorizes a matrix into a lower triangular matrix L and upper triangular matrix R, the Cholesky factorization, which computes $A = LL^T$ and the QR decomposition that factorizes A into an orthogonal matrix Q and upper triangular matrix R. However, Meister (2015) describes some properties of direct solvers which might render some of these algorithms rather unsuited for data-parallel acceleration cards:

- The runtime complexity of such algorithms lies in $O(N^3)$, which could be an infeasible scaling factor in our case since RBF systems can be very large in multi-physics simulations.
- Algorithms based on the LU decomposition have a decreasing degree of parallelism during matrix decomposition.
- Irregular sparsity patterns are hard to optimize when it comes to matrix decompositions.

Still, major research effort has been put into developing matrix factorization algorithms for GPUs. As a result, there are libraries that offer matrix factorization methods, which we discuss in Sec. 3.2.

QR Decomposition A QR decomposition decomposes a matrix $A \in \mathbb{R}^{m \times n}, m \geq n$ into an orthogonal matrix Q, i.e., $Q^T Q = \mathbb{I}$ and an upper triangular matrix R. Different algorithms have been developed to compute the QR decomposition, e.g., Gram-Schmidt orthogonalization and Householder reflections. In this thesis, we make use of Householder reflections. Householder reflections have been introduced by Householder (1958), hence the name. The main idea is to produce an upper triangular matrix R by mirroring the column vectors of A s.t. they are co-linear to unit vectors. The matrix Q is the product of matrices Q_k. Each Q_k modifies the k-th column vector x_k s.t. every entry below the k-th is zero. This is achieved through Householder reflections:

$$Q_k = \mathbb{I} - 2 \cdot v \cdot v^T \tag{2.8}$$

v is defined as follows:

$$v = \frac{u}{||u||}, \quad u = x_k - \alpha \cdot e_k \tag{2.9}$$

e_k is the k-th basis vector of the canonical basis. Q_k now transforms x_k s.t. that every entry below and above row k is zero:

$$Q_k \cdot x_k = \begin{bmatrix} 0 \\ \vdots \\ \alpha_k \\ \vdots \\ 0 \end{bmatrix} \tag{2.10}$$

This vector is co-linear to e_k and has the length α. Iteratively applying this scheme to every column of A transforms it into R. Q is the product of all transformation matrices Q_k.

Details of the proof are provided by Householder (1958).

The entire procedure is provided in Algorithm 2.2. Analysis of parallelization strategies is done by inspecting the different components of this algorithm separately.

Referring back to Sec. 2.1, we can decompose multiple steps of the QR decomposition algorithm into BLAS routines and therefore make a connection to the advantages of data-parallel GPUs. BLAS 1 routines can be found whenever a vector is multiplied with a scalar or added to/subtracted from another vector (**AXPY**) which results in N independent calculations for a vector in $\mathbb{R}^N$. Furthermore, dot products (**DOT**) are also covered by this level. However, they additionally require the addition of multiplied values which results in a so-called "fan-in" process which decreases parallelization opportunities by a logarithmic factor. Finally, **NORM** is part of BLAS 1 since it is composed of a **DOT** combined with a square root operation. BLAS 2 routines come into place when matrices are multiplied with vectors and are much more favorable in GPU settings since their workload ratio is much better as $\mathbb{R}^{N \times N}$ matrices imply N independent BLAS 1 routines (**AXPY** or **DOT**). Having more calculations enables the GPU to make better use of a large number of available cores.

We marked some BLAS routines in the algorithm. First and foremost, there are multiple `for`-loops in the Householder algorithm that reflect BLAS 1 routines. The loop in line three is a **DOT** between elements of a with themselves, including the summation process that adds up to α. In line 14, there is also a **DOT**, but between two different vectors. Line 18 and 22 an **AXPY** operation, including scaling factor γ. Finally, a BLAS 2 operation is listed in line 26ff, which is a **GEMV**. Depending on the implementation, some BLAS 1 routines might be grouped s.t. they form a BLAS 2 operation. The most outer loop, which begins in line one, cannot be parallelized as every iteration also depends on results from previous iterations, which modify Q and R.

We now look at the class of iterative solvers that have a constant degree of parallelism throughout the entire solving process.

Iterative Methods According to Meister (2015), direct solvers cannot always exploit properties of sparse matrices, i.e., reduce runtime and memory requirements. Since many simulation setups already suffer from discretization errors, it is often sufficient to calculate an approximate solution. It is often impossible to calculate an exact solution with residual $r = 0$. Iterative methods are well suited for calculating approximate solutions whose error is in the range of discretization errors. Meister (2015) defines iterative methods as follows:

Definition 2.4.1
An iterative method is a mapping

$$\phi : \mathbb{C}^n \times \mathbb{C}^n \to \mathbb{C}^n \tag{2.11}$$

and called linear if there exist matrices $M, N \in \mathbb{C}^{n \times n}$ such that

$$\phi(x, b) = Mx + Nb \tag{2.12}$$

Algorithm 2.2 Householder QR Decomposition (taken from Meister (2015))

Ensure: $A = QR$
1: **for** $k = 1, \ldots, n$ **do**
2: $\alpha = 0$
3: **for** $i = k, \ldots, n$ **do**
4: $\alpha = \alpha + a_{ik}^2$
5: **end for**
6: **if** $a_{kk} > 0$ **then**
7: $d_k = -\sqrt{\alpha}$
8: **else**
9: $d_k = \sqrt{\alpha}$
10: **end if**
11: $\beta = \frac{1}{d_k \cdot a_{kk} - \alpha}, a_{kk} = a_{kk} - d_k$
12: **for** $j = k + 1, \ldots, n + 1$ **do**
13: $\alpha = 0$
14: **for** $i = k, \ldots, n$ **do**
15: $\alpha = \alpha + a_{ik} \cdot a_{ij}$
16: **end for**
17: $\gamma = \beta \cdot \alpha$
18: **for** $i = k, \ldots, n$ **do**
19: $a_{ij} = a_{ij} + \gamma a_{ik}$
20: **end for**
21: **end for**
22: **for** $i = k, \ldots, n$ **do**
23: $a_{ik} = \sqrt{\frac{-\beta}{2}} \cdot a_{ik}$
24: **end for**
25: **end for**
26: **for** $k = n, \ldots, 1$ **do**
27: **for** $j = k + 1, \ldots, n$ **do**
28: $a_{k,n+1} = a_{k,n+1} - a_{kj} \cdot x_j$
29: **end for**
30: $x_k = \frac{a_{k,n+1}}{a_{kk}}$
31: **end for**

The main aspect of every iterative method is the *convergence* to a close or even the true solution. Convergence is defined as shown below:

Definition 2.4.2
An iterative method is said to converge if for every right-hand side $b \in \mathbb{C}^n$ and every initial guess $x_0 \in \mathbb{C}^n$ there exists a limit independent from the initial guess:

$$\hat{x} = \lim_{m \to \infty} x_m = lim_{m \to \infty} \phi(x_{m-1}, b) \tag{2.13}$$

The two algorithms that we discuss now are so-called *Krylov subspace methods*. Meister (2015) elaborates Krylov subspaces and corresponding methods in more detail.

Conjugate Gradients The conjugate gradient (CG) algorithm is a widely used algorithm, which has proven to work well on large and sparse systems of linear equations. It requires the system matrix $A \in \mathbb{R}^{n \times n}$ to be symmetric positive-definite (s.p.d.), i.e., $A^T = A$ and $x^T Ax > 0$. If there was exact floating-point arithmetic, it would be able to calculate the exact solution in n steps. A pseudocode is listed in Algorithm 2.3.

Algorithm 2.3 CG Algorithm (taken from Schulte (2022))

Require: A s.p.d.
Ensure: $x = A^{-1}b$

1: $x = 0^n$
2: $d = r = b - A \cdot x$
3: $r_0 = r$
4: **while** $\frac{||r||}{||r_0||} \geq \epsilon$ **do** // Convergence Criterion
5: $\quad \alpha = \frac{r^T \cdot d}{d^T \cdot A \cdot d}$
6: $\quad x = x + \alpha \cdot d$
7: $\quad r = r - \alpha \cdot A \cdot d$
8: $\quad \beta = \frac{r^T \cdot A \cdot d}{d^T \cdot A \cdot d}$
9: $\quad d = r + \beta \cdot d$
10: **end while**

The while-loop starting in line 4 is strictly sequential, i.e., performing multiple iterations at the same time is not possible since every iteration first checks for convergence of the residual. By looking at the algorithm, it becomes clear that almost all operations are either BLAS 1 or BLAS 2. For instance, the convergence criterion in line 4 relies on a **NORM**, which is a combination of a **DOT** and a square root.

It can be seen that CG heavily relies on BLAS routines that can be parallelized using accelerator cards. However, there is no possibility of distributing every outer loop iteration to different threads simultaneously.

GMRES Another well-known method for iteratively solving systems of linear equations is generalized minimal residual method (GMRES), which has been brought up by Saad and Schultz (1986). Its algorithmic description is shown in Algorithm 2.4. We again marked some of the BLAS 1 and BLAS 2 routines. On one hand, one GMRES iteration contains more operations than one CG iteration. On the other hand, GMRES can also be applied on non-s.p.d. system matrices.

2.4.1 Preconditioning

The convergence behavior of CG and GMRES can be improved by reducing the condition number κ of the system matrix. Meister (2015) provides the following definitions for the condition number:

Definition 2.4.3
Let $A \in \mathbb{C}^{n \times n}$ be regular. The condition number κ follows as:

$$\kappa(A) = ||A||_a ||A^{-1}||_a \tag{2.14}$$

Algorithm 2.4 GMRES Algorithm (taken from Meister (2015))

Ensure: $x = A^{-1}b$
1: $r_0 = b - Ax_0$
2: **if** $r_0 \neq 0$ **then**
3: $v_1 = \frac{r_0}{||r_0||} \quad \gamma_1 = ||r_0||$
4: **for** $j = 1, \ldots, n$ **do**
5: **for** $i = 1, \ldots, j$ **do**
6: $h_{ij} = (v_i, Av_j)_2$
7: **end for**
8: $w_j = Av_j - \sum_{i=1}^{j} h_{ij}v_i \quad h_{j+1,j} = ||w_j||$
9: **for** $i = 1, \ldots, j-1$ **do**
10: $\begin{bmatrix} h_{ij} \\ h_{i+1,j} \end{bmatrix} = \begin{bmatrix} c_i & s_i \\ -s_i & c_i \end{bmatrix} \begin{bmatrix} h_{ij} \\ h_{i+1,j} \end{bmatrix}$
11: **end for**
12: $\beta = \sqrt{h_{jj}^2 + h_{j+1,j}^2}; \quad s_j = \frac{h_{j+1,j}}{\beta}$
13: $c_j = \frac{h_{jj}}{\beta} \quad h_{jj} = \beta$
14: $\gamma_{j+1} = -s_j\gamma_j; \quad \gamma_j = c_j\gamma_j$
15: **if** $\gamma_{j+1} = 0$ **then**
16: **for** $i = j, \ldots, 1$ **do**
17: $\alpha_i = \frac{1}{h_{ii}}(\gamma_i - \sum_{k=i+1}^{j} h_{ik}\alpha_k)$
18: **end for**
19: $x = x_0 + \sum_{i=1}^{j} \alpha_i v_i$
20: **BREAK**
21: **else**
22: $v_{j+1} = \frac{w_j}{h_{j+1,j}}$
23: **end if**
24: **end for**
25: **end if**

$||.||_a$ is a matrix norm. The matrix with the best possible condition number $\kappa = 1$ is the identity matrix $\mathbb{I}$. Hence, the aim of preconditioning is to reduce the condition number of the system matrix by left- or right-multiplying the system with a preconditioner M s.t. the system matrix becomes more similar to the identity matrix. Since both sides are multiplied with the same M, the almost same solution in fewer iteration steps is expected. In this thesis, we work with preconditioning techniques that modify the system by multiplying A on the left:

$$M \cdot \Phi \cdot \lambda = M \cdot f \tag{2.15}$$

There are many preconditioning techniques in order to obtain M. In this thesis, we mainly focus on two: (Block-)Jacobi and Cholesky. We use the definitions provided by Meister (2015).

(Block-)Jacobi The Jacobi preconditioner is a rescaling of the system matrix by dividing every element using the diagonal entries of A, i.e., $M = D^{-1}$ with $D = \text{diag}(A)$. This preconditioner has proven to be effective for diagonally dominant matrices. However, in the case of RBF interpolation,

most matrices have diagonals consisting entirely of ones or zeros. The latter case also leads to an undefined operation. An alternative is the blocked version called Block-Jacobi. Instead of only using diagonal elements of A, D consists of diagonal blocks that span across the main diagonal and one or multiple neighboring diagonals. D^{-1} can be obtained by solving multiple small systems that arise from the blocking scheme.

Incomplete Cholesky The Cholesky decomposition of a s.p.d. matrix is $A = L \cdot L^T$ with L being a lower triangular matrix. There is also a preconditioning method that makes use of the Cholesky decomposition. It does not compute the true L, but a matrix K which is close to L and has the same sparsity pattern as A by copying all zero entries of A into K. This results in the following decomposition where F denotes the difference between the incomplete and complete Cholesky factorization:

$$A = K \cdot K^T = L \cdot L^T + F \tag{2.16}$$

Further Preconditioners There are further preconditioning methods. We want to mention two of them here, but they only play a minor role in this thesis.

The incomplete LU (ILU) decomposition computes a factorization of A s.t. the following holds:

$$A = L \cdot U + F \tag{2.17}$$

L is a lower triangular matrix, U an upper triangular matrix and F the difference between incomplete and complete LU decomposition. The main motivation of this technique is to avoid increasing memory requirements through a so-called “fill-in” process, which describes the fact that during a factorization process, zero entries in A can be non-zero in L and U. Therefore, L and U stick to the sparsity pattern defined by A.

Another widely applied method is the algebraic multi-grid (AMG) which can be used as a standalone solver and preconditioner. Its concept is to recursively calculate a solution of a coarser version of the system and then propagate the error estimate onto the finer levels. Naumov et al. (2015) present a GPU-accelerated library called “AmgX”. It contains many different AMG methods as well as Krylov subspace methods and preconditioners. Furthermore, it is capable of distributing problems among multiple GPUs using graph-based heuristics, which increases the computational resources.

2.5 Related Work

High effort has been put into the research of solving large and sparse linear systems efficiently and lately also on GPUs. RBF interpolation is also of high interest in many different applications that deal with scattered data points. We discuss some related research work and explain how we can use the observations and results presented there in our work.

Bolz et al. (2003) implement and investigate sparse matrix solvers, namely CG and multigrid, on GPUs in the context of computer graphics applications. Since back then frameworks such as CUDA were not available yet, they had to rely on fragment shader programming, which however is not

necessary anymore nowadays. First, they describe what is required in order to use solvers on GPUs. They draw the conclusion that the following ingredients are necessary: memory-efficient data structures, data-parallel algorithms for BLAS routines, reduction operators, and matrix assembly kernels. These observations are very similar to ours: Whereas nowadays we can rely upon highly optimized libraries such as *cuBLAS*, we still have to investigate efficient matrix assembly routines and data structures. Only being able to use shader programming and pixel buffers, they were still able to achieve a significant speed up when comparing the number of unstructured matrix multiplications per second (120 on the GPU vs. 75 on the CPU) and structured multiplications ($1,370$ vs. 750). This clearly shows that even back then, before GPGPU programming was made possible through frameworks such as CUDA, GPUs have shown great potential in speeding up very large mathematical operations.

Siegel et al. (2011) discuss a more technical aspect of GPGPU optimization which is efficient memory layouts. They mainly focus on nested data structures such as "Array of Structures" (AoS) and "Structure of Arrays (SoA)". Their main finding is that memory accesses by different threads in a warp shall always be coalesced, i.e., neighbored threads should access neighbored memory addresses. In this thesis, we also investigate how we can possibly optimize our code w.r.t. memory accesses and try to measure the benefit that we can get out of it. It is of high relevance to check for well-executed memory accesses since these can heavily impact the performance of GPUs.

A very similar work is presented by Ding et al. (2018). They also investigate how RBF interpolation can potentially benefit from the usage of accelerator cards. The focus lies on local RBF interpolation, i.e., only vertices near the target vertex are considered. They implement a "k nearest neighbor" search to find the nearest neighbors of each vertex on the GPU. The authors draw the conclusion that RBF interpolation does not gain significant speedups when executed on accelerator cards because of two reasons: the limited size of VRAM and high computational workload when it comes to solving a system of linear equations. Contrary to their findings, we present different cases in Chapter 4 where the usage of accelerator cards definitely improves the runtime of the simulation. We also discuss the matrix-free approach that we implement to circumvent these VRAM limitations, which we also encounter throughout our experiments.

In addition to the work of Ding et al. (2018), Cuomo et al. (2013) discuss GPU-accelerated RBF interpolation for surface reconstruction in computer graphics. As already mentioned in Sec. 2.3.1, RBF interpolation is a widely used technique for dealing with scattered data points. In order to solve the interpolation problem, they use a preconditioned GMRES provided by PETSc. They especially focus on the choice of a sparse matrix format as they conclude that matrix-vector multiplication is the most expensive part in terms of runtime. We discuss in Sec. 3.4.4 why we decide to not focus on sparse matrix formats, although many GPU solver algorithms can be optimized for sparse matrix formats. The authors observe speedup factors of four and higher in their experimental evaluation compared to a single-core CPU. In this thesis, we explicitly look at multi-core CPUs to ensure a more realistic comparison between them and many-core GPUs.

3 Implementation Details

The last chapter listed the necessary ingredients for RBF interpolation. There are two computationally intense components that are of high importance:

1. The assembly of the RBF matrices Φ and A_M, which require the evaluation of $O(K^2)$ RBF entries.
2. The calculation of the coefficient weights λ of the system $\Phi \cdot \lambda = f$.

Throughout this chapter, we always focus on solving these two problems.

We start with discussing requirements that build the foundation of all decisions made throughout the implementation phase. Afterward, we describe implementation strategies using the HPC linear algebra library of our choice in C++ in order to address the two main components.

3.1 Requirements

Being able to accelerate preCICE using GPUs of different vendors requires a well-considered implementation strategy. Our task is to find a good strategy that fulfills multiple requirements. This involves finding a library that offers cross-platform GPU support. In the following, we impose several requirements on a linear algebra library in order to be suited for our use case.

One, if not the most important requirement is that a candidate supports mathematical routines on GPUs natively. We define this as an absolute must since writing all calculation kernels from scratch ourselves is infeasible and too error-prone and time-consuming compared to what can be achieved when using a robust and proven framework. However, this does not mean that we do not want to be able to implement some GPU kernels ourselves. Therefore, a library that offers custom kernel dispatch mechanisms is a big advantage. More precisely, since we want to be able to accelerate data-parallel operations, we need to be able to implement routines that reflect a `parallel_for` as easily as possible.

However, there are computing environments that do not have any GPU available. We still want to support them by either running our algorithms in parallel on the CPU or without any parallelism.

Furthermore, another important aspect is the functionality and availability of different solvers and other numerical routines offered by a library. Since preCICE relies heavily on mathematical computations and therefore well-known libraries, e.g., Eigen and PETSc, we need to make sure that our choice is able to compete with them.

When it comes to maintaining the software, we want the library to be well-documented and maintained s.t. we can rely on future support and state-of-the-art backends.

As a last point, we want a library to be as user-friendly as possible to make debugging and performance measurements as easy as possible since the goal of this thesis is to evaluate which performance gains are possible when using GPUs.

3.2 Frameworks

We discuss different frameworks and libraries in this section that should meet the different requirements that we listed above.

Ginkgo Ginkgo (Anzt et al., 2022) describes itself as a "linear *operator* algebra library" and is part of the xSDK, which indicates that it is developed w.r.t. large-scale HPC systems. The term "linear *operator* algebra library" refers to the basis which Ginkgo builds upon: linear operators. The main approach is to treat linear operators (e.g., matrix factorization routines) as basic building blocks that can be modified and extended via an object-oriented approach. This implies that different operations (e.g., factorization, preconditioning, multiplication) can be stacked upon each other and applied to a matrix on which all operations are generated. Ginkgo also allows for the creation of custom matrix data structures. This enables the users of the library to implement matrix-free methods by providing a definition for the `apply()` function, which is the function that defines what an operator does.

Being part of the xSDK means that the library must provide (extreme-)scalability features for different state-of-the-art HPC backends. Ginkgo uses the concept of so-called "executors". Every mathematical operation is implemented for every available executor redundantly, which makes it very easy to write code within one environment (e.g., one that provides Nvidia GPUs) and run it in another environment (e.g., one without any GPUs). As of the time of writing this thesis, Ginkgo supports the following parallel backends: CUDA, AMD HIP, OpenMP, and Intel DPCPP. On top, it provides sequential reference implementations that allow for correctness checks.

Referring to our requirements in Sec. 3.1, we can rely upon being able to support many heterogeneous HPC systems consisting of GPUs by Nvidia and/or AMD and multi-core CPUs.

Since Ginkgo offers logging mechanisms that yield information about solver algorithms such as required iterations or norm reduction, we can also use it for debugging and performance measurement purposes.

Kokkos Kokkos (Edwards et al., 2014; C. Trott et al., 2021; C. R. Trott et al., 2022) is another, well-known C++ library for high-performance programming. Just as Ginkgo, it does support multiple HPC backends, also including GPUs via CUDA and SYCL[1]. However, its approach is slightly different. Instead of providing "building blocks" for mathematical operations, it offers machine-independent abstractions for (data-parallel) logical constructs such as `parallel_for` and `reduce`. Furthermore, Kokkos represents memory as a new data structure called *View*. These views allow for memory copy mechanisms between RAM and remote memory spaces such as

[1] https://www.khronos.org/sycl/

VRAM. They also take into account different memory layouts, which enables more efficient memory transactions, by providing default memory spaces for backends such as CUDA. Both parts, code execution and memory access, are defined as *execution space* and *memory space* respectively.

Further Libraries and Frameworks Ginkgo and Kokkos are two out of many libraries that do offer support for GPU acceleration. There are a few more that we want to discuss in order to draw the best conclusion.

The *cuSolver* library[2] is developed by Nvidia and offers matrix factorization methods such as QR and Cholesky decomposition. It is tailored for Nvidia GPUs specifically. AMD also provides a corresponding library called *rocSOLVER*[3]. Both libraries stick to the naming scheme and functionality of the *LAPACK* numerical linear algebra library (Anderson et al., 1999), which is developed for the scientific programming language Fortran. Since cuSolver and rocSOLVER are released by the GPU vendors themselves, we can expect high performance optimizations.

OpenCL[4], initiated by Apple and currently developed by the Khronos Group, is a C-like programming interface that supports cross-platform parallel computing. Its language and logical structure are very similar to CUDA. Although it does support multiple accelerator cards, it cannot make use of vendor-specific optimizations such as CUDA.

The last framework that we want to discuss is *SYCL*[5]. Like OpenCL, it is developed and maintained by the Khronos Group and offers abstraction layers similar to Kokkos. It builds upon OpenCL by leveraging its low-level concepts through a higher-level programming interface. The SYCL compiler is able to compile this higher-level code using low-level implementations as provided by OpenCL for example.

Comparison and Conclusion Given the different frameworks and libraries, we can now evaluate the suitability of each one w.r.t. to RBF mapping in preCICE.

First, we start by comparing Kokkos to Ginkgo as they are the main candidates. The main focus of Kokkos is HPC programming, which fits our requirement of being able to program GPUs very well. However, it does not serve as a pure linear algebra library. Since we deal with mathematical problems in this work, namely RBF interpolation, we need the library to offer as much mathematical functionality as possible. Even though Kokkos maintains a math library called “Kokkos Kernels”[6], it is not as self-contained as Ginkgo from a developer’s perspective. For instance, whereas Ginkgo allows to directly use its data structures within its algorithms and combines different preconditioners and solvers in just a few lines of code, Kokkos algorithms and data structures are standalone which implies longer configurations and therefore code length. This also creates more dependencies for preCICE since this setup would not only require Kokkos but also its math library and its own dependencies. Another advantage of Ginkgo over Kokkos is given by the developers of Ginkgo. They argue that their approach of writing optimized, native kernels for each backend separately,

[2] https://docs.nvidia.com/cuda/cusolver/
[3] https://github.com/ROCmSoftwarePlatform/rocSOLVER
[4] https://www.khronos.org/opencl/
[5] https://www.khronos.org/sycl/
[6] https://github.com/kokkos/kokkos-kernels

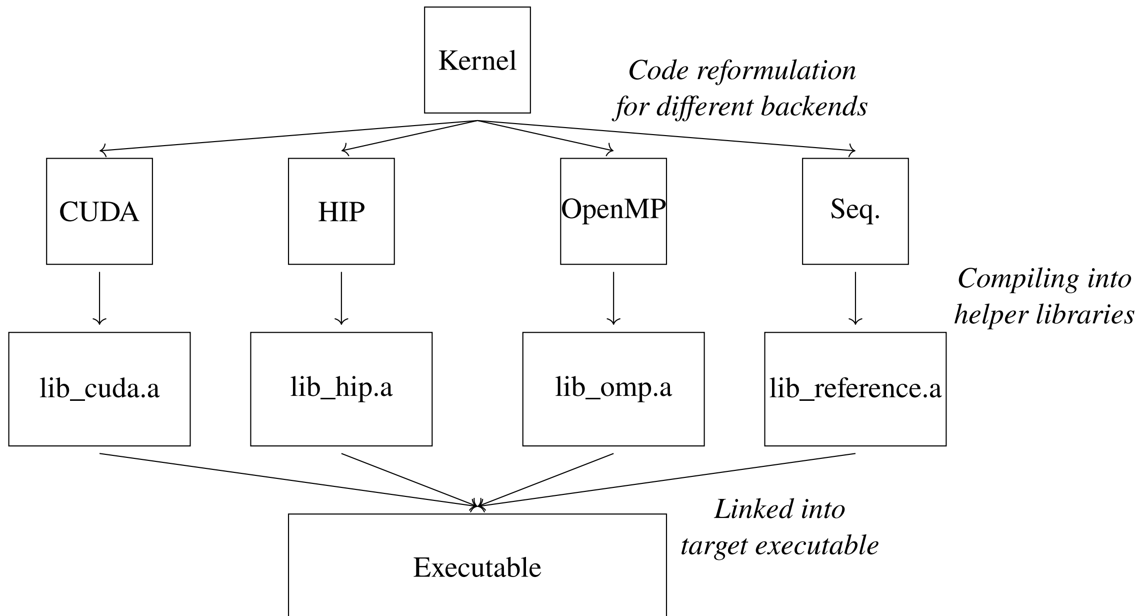

Figure 3.1: The unified kernel mechanism of Ginkgo takes one kernel and compiles it for different parallel backends.

instead of relying on one portability layer as with Kokkos, allows for much better optimized kernels w.r.t. hardware constraints. However, one could also argue that Kokkos is much more known and used than Ginkgo (yet), i.e., Kokkos is used, tested and verified by a much wider audience than Ginkgo.

Our goals could very likely be achieved with both frameworks. We decide to go with Ginkgo because of its natural linear algebra approach which makes it more similar to the libraries already used in preCICE, namely Eigen and PETSc. Yet there is still the issue that Ginkgo mainly focuses on iterative solvers for sparse systems and therefore not on direct solvers. As a result, we further take a look at cuSolver to additionally implement and evaluate its QR decomposition on GPUs, which can be translated to AMD HIP through “hipify”. One downside of this approach is that cuSolver and rocSOLVER are restricted to GPUs, however, they both make it easy to implement a fast QR decomposition. Therefore, we decide to use Ginkgo for all parts of our implementation except for the QR decomposition, which is only a small portion of the entire implementation. If Ginkgo is going to support QR factorizations in the future, our cuSolver implementation could easily be replaced.

3.3 Integration into preCICE

This section depicts the workflow that our GPU-based solver introduces in preCICE and which solvers and preconditioners are available in our implementation. It also describes the configuration possibilities of our implementation, which gives the user as much freedom as possible.

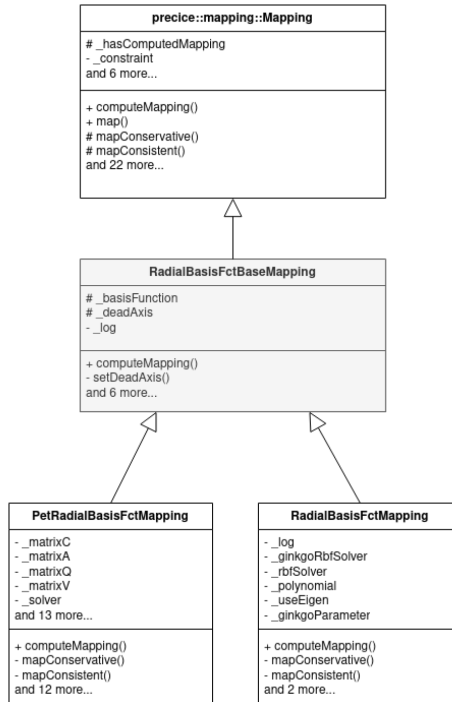

Figure 3.2: The UML class diagram of the RBF mapping in preCICE. Our implementation is embedded into the `RadialBasisFctMapping` class.

First and foremost, we need to integrate Ginkgo, cuSolver and our matrix assembly kernel in preCICE. The matrix assembly kernel is responsible for setting up Φ and A_M (cf. Sec. 2.3.1 and Sec. 3.4.1). Ginkgo and cuSolver can be added to preCICE via the CMake[7] toolchain. Our assembly kernel is compiled using the Ginkgo unified kernel mechanism depicted in Figure 3.1. As of writing this thesis, the interface is exposed to the user in an experimental branch of Ginkgo[8]. This functionality takes the kernel code and compiles it for different parallel backends. We decide to compile for CUDA, HIP, and OpenMP as well as to provide a single-core version in preCICE.

Figure 3.2 shows the class diagram of RBF mapping in preCICE. At the heart of the implementation, there is an interface class called `Mapping` that defines the methods that each mapping method has to provide. There are two methods that we put special focus on: `computeMapping()` and `map()`. In the context of RBF mapping, the first method is responsible for initializing the RBF solver and executing one-time computations such as matrix decompositions in the case of direct solvers as they are independent of changing right-hand side values. This is also the method where the matrix assembly described in Sec. 3.4.1 takes place. It is possible to do it only once because we only deal with non-changing, i.e., constant meshes. If the meshes changed over time, it would be necessary to repeatedly assemble the matrices. If preCICE introduced such a feature in the future, the usage of matrix assembly on GPUs would play an even bigger role. Furthermore, in the case of direct solvers, the decomposition of the system matrix can be computed once and reused afterward for the entire

[7] `https://cmake.org/`

[8] `https://github.com/ginkgo-project/ginkgo/tree/a195f856ee946b1b9387d30513dd992df4501df3` (last accessed on February 9, 2023)

duration of the simulation. In contrast, the mapping functions are called repeatedly since they have to recompute the interpolation coefficients every time preCICE is provided with new right-hand side values. The `map()` function differentiates between two mapping constraints: *consistent* and *conservative*. Consistent mapping means that constant values are exactly reproduced. Conservative mapping methods preserve the sum of all values. In this thesis, we focus on the `mapConsistent()` function, which is called by `map()`, as our test cases use a consistent mapping approach. The goal is to put as much work as possible into `computeMapping()` in order to keep the amount of recurring workload as small as possible.

There are two specialization classes for RBF mapping, `PetRadialBasisFctMapping` and `RadialBasisFctMapping`. The first one corresponds to the PETSc implementation and the second one to the Eigen implementation. We decide to add our Ginkgo implementation as a class member to the second class (as depicted in Figure 3.2) and decide conditionally whether to use the Eigen solver `_rbfSolver` or `_ginkgoRbfSolver` based on the configuration. As a result, our Ginkgo implementation integrates well into preCICE as it retrieves values from Eigen data structures, does its work on the accelerator card using Ginkgo and CUDA/HIP, and finally writes it back into Eigen variables. This architecture might be subject to change in the future.

To realize GPU support in preCICE by using Ginkgo, we create a separate class called `GinkgoRadialBasisFctSolver`. It stores solver configurations, RBF-related matrices, and solver as well as preconditioner objects. Furthermore, it is responsible for calling the cuSolver QR decomposition. Having it this way allows us to exchange the QR decomposition for another implementation in the future if necessary.

Every coupling process in preCICE can be configured via an XML file. Therefore, we introduce the following XML attributes for the already existing RBF mapping XML tag that steer the behavior of our GPU solvers:

- `executor`: This is the (parallel) backend that the Ginkgo solver runs on (CUDA, HIP, OpenMP, Sequential).
- `solver`: This attribute configures the solver to use, which is either CG, GMRES, or QR.
- `gpu-device-id`: This number determines the GPU device ID which is unique for every card in a multi-GPU setting.
- `max-iterations`: This is the maximum number of iterations that a solver can execute before being stopped.
- `use-preconditioner`: This Boolean flag enables or disables preconditioning.
- `preconditioner`: If the flag is set to `true`, this setting is taken as preconditioner, which must be either Jacobi or Cholesky.
- `jacobi-block-size`: As described in Sec. 2.4.1, there are also blocked variants of the Jacobi preconditioner. The block size can be set using this attribute.
- `enable-unified-memory`: This enables the CUDA unified memory mechanism that allows for overallocation by automatically transferring memory from and to the GPU.

We reuse the already existing `solver-rtol` to set the residual norm reduction criterion that is used to end iterating before the maximum count of iterations is reached. This criterion measures the relative norm reduction of the current approximation compared to the initial guess and is only relevant for iterative solvers.

Testing There are already existing unit tests in preCICE for almost every functionality, including RBF mappings. We can make use of them to verify that our GPU-based implementation works correctly.

3.4 RBF Interpolation Components

We now describe our approaches for dealing with RBF interpolation as a mathematical problem that should be solved as accurately and fast as possible. This means that we do not always want to achieve the *same* accuracy as direct solvers (e.g., LU decomposition using Eigen), but rather achieve an interpolation accuracy close to the one of the direct solvers but with significant speedup.

First, we start with the initial setup phase of every RBF interpolation, the matrix assembly. As already stated in Sec. 2.3.1, the workload of matrix assembly is squared w.r.t. the problem size. Afterward, we provide implementation details of our different solvers. Moreover, we can divide the optimization approaches of both matrix assembly and solvers into two categories: technical and mathematical. Whereas the matrix assembly can be optimized in a technical way, i.e., w.r.t. hardware constraints, the iterative solvers might be optimized mathematically s.t. we get higher accuracy in less time.

3.4.1 Matrix Assembly

As stated in Sec. 2.3.1, RBFs are defined as follows:

$$\varphi_x(||y - x||) \tag{3.1}$$

Every function evaluation requires the norm between an input point and the support point, which usually is the Euclidean norm. Afterward, the function value itself is calculated w.r.t. to the radius using the aforementioned norm. Our interpolation problem consists of solving a linear system of equations in order to obtain a coefficient vector λ:

$$f = \Phi \cdot \lambda \tag{3.2}$$

We can then retrieve the interpolated values g as

$$g = A_M \cdot \lambda \tag{3.3}$$

When using a separate polynomial contribution, we need the two matrices Q_M and V_M on top.

$$\Phi = \begin{bmatrix} \varphi_{1,1} & \varphi_{1,2} & \cdots & \varphi_{1,K} \\ \varphi_{2,1} & \ddots & & \varphi_{2,K} \\ \vdots & & \ddots & \vdots \\ \varphi_{K,1} & \varphi_{K,2} & \cdots & \varphi_{K,K} \end{bmatrix} \tag{3.5}$$

Figure 3.3: Conceptual thread distribution for matrix assembly; each color represents a distinct thread that is responsible for a unique φ.

Data Parallelism Exploiting data parallelism for RBF interpolation is done by reformulating the linear system of equations as matrix-vector product. Let $\varphi_{i,j}$ denote the RBF kernel which takes the distance between x_i and x_j as argument:

$$\begin{bmatrix} \varphi_{1,1} & \varphi_{1,2} & \cdots & \varphi_{1,K} \\ \varphi_{2,1} & \ddots & & \varphi_{2,K} \\ \vdots & & \ddots & \vdots \\ \varphi_{K,1} & \varphi_{K,2} & \cdots & \varphi_{K,K} \end{bmatrix} \cdot \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_K \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_K \end{bmatrix} \tag{3.4}$$

Each entry of the system matrix contains an evaluation of a RBF kernel given two points x_i and x_j. Furthermore, each calculation is independent of every other kernel. As a result, we get $O(K^2)$ independent function evaluations which are entirely data-parallel by definition. Speaking in programming terms, this corresponds to a `parallel_for` loop.

Assembling the interpolation matrix Φ as well as the mapping matrix A_M and optionally the polynomial matrices Q_M and V_M on the GPU brings two advantages:

1. The matrix assembly process is embarrassingly data-parallel. Therefore, we can make full use of the number of calculation units on an accelerator card.

2. Since both matrix assembly and solvers run on the GPU, no additional memory transfers are required; the matrices can just reside in the VRAM.

Figure 3.3 shows the simple concept of distributing the workload amongst the threads on the GPU. There is one thread per matrix element (depicted by a different color for each thread). This leads to $O(K^2)$ threads working independently on these matrix entries, one per thread. Given that a very fine discretization of the mesh result in a lot of RBF support points, the workload that must be handled by the executing device grows rapidly.

To implement this kernel in preCICE, we make use of the unified kernel mechanism provided by Ginkgo, which is described in Sec. 3.3. The kernel dispatch on the GPU, however, is not fully optimal in the sense that Ginkgo always launches a one-dimensional grid of blocks and threads, independent of whether the problem is one-dimensional or two-dimensional. As a result, to also allow for artificial 2D grids, every thread in this long 1D grid receives its 2D position (i, j) as function parameters. As a result, each thread gets its unique matrix element to work on.

$$M_I = \begin{bmatrix} v_{1_x} & v_{1_y} & v_{1_z} \\ & \vdots & \\ v_{i_x} & v_{i_y} & v_{i_z} \\ & \vdots & \\ v_{j_x} & v_{j_y} & v_{j_z} \\ & \vdots & \\ v_{K_x} & v_{K_y} & v_{K_z} \end{bmatrix}$$

Figure 3.4: Mesh vertex coordinates are stored in a dense matrix. The thread (i, j) (depicted in green) accesses all three dimensions of v_i and v_j.

In order to assemble the matrices Φ and A_M on an accelerator card, we need the mesh vertices with their coordinates on them too. We realize this by putting all vertex coordinates into a $K \times D$ matrix where D refers to the spatial dimension of the mesh; either 2D or 3D. As an example, we look at the thread at matrix position (i, j). Figure 3.4 displays that this thread (marked in green) will access six elements in total from the input vertex matrix M_I. To better distinguish between dimension x and vertices, we will denote vertices as v in the following theoretical discussion. The thread will iteratively access the x, y, and z dimension of both v_i and v_j, calculate the Euclidean distance and therefore obtain the distance between vertex v_i and v_j, which is the radius that is passed to the RBF. For the assembly of A_M, we need an additional matrix containing the vertex coordinates of the output mesh too.

Optimization Approaches for the Matrix Assembly The implementation described above gives rise to further optimization in that manner that it does not take optimal memory layouts and accesses into account. We use the Nvidia Nsight Compute profiler[9] to analyze the performance and bottlenecks of our GPU kernel. It measures how well a GPU kernel handles L1/L2 caching, global memory accesses, workload balancing, the behavior of warps (= groups of 32 identically timed threads), and further statistics. With the help of Nsight Compute and looking at the memory usage, we can identify the following optimization approaches:

1. Access to the input vertex matrix is not coalesced. We need to modify its layout to have a better memory access scheme.
2. Multiply operations followed immediately by addition can be combined into a *fused multiply-add (FMA)* operation, which only needs one instead of two rounding steps.

Thus, besides our reference kernel implementation, we also add an additional CUDA and HIP optimized variant that improves on these observations. The goal is to gain additional speedup.

[9] https://developer.nvidia.com/nsight-compute

$$M_I = \begin{bmatrix} v_{1_x} & \cdots & v_{j_x} & v_{j+1_x} & \cdots & v_{K_x} \\ v_{1_y} & \cdots & v_{j_y} & v_{j+1_y} & \cdots & v_{K_y} \\ v_{1_z} & \cdots & v_{j_z} & v_{j+1_z} & \cdots & v_{K_z} \end{bmatrix}$$

Figure 3.5: Optimized vertex layout for coalesced GPU usage. First, all red elements are accessed in parallel, then all green ones, and finally the blue ones.

Our first optimization strategy targets memory layout. It is important to emphasize that continuous threads within a block perform best if and only if memory addresses coalesce, i.e., two consecutive threads access two consecutive memory slots of the underlying data type (in our case *double*). The Euclidean distance is defined as follows:

$$||x - y|| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{3.6}$$

Speaking in programming terms, this is a simple `for`-loop iterating sequentially over each dimension of the vertices. Therefore, we have to rearrange the data in the vertex matrices. We exemplary look at two threads, $t_1 = (i, j)$ and $t_2 = (i, j + 1)$ that are grouped in the same block. To calculate the Euclidean distance between v_i and v_j and v_{j+1} respectively, the threads need to sequentially process x, y and finally z dimension of these vertices. However, Figure 3.4 shows that the memory access is not coalesced in this case. When t_1 and t_2 read v_{j_x} and v_{j+1_x} respectively to compute the distance in x direction in parallel, there is a padding between these two values, induced by v_{j_y} and v_{j_z}. As a result, memory access is not aligned and the GPU will encounter significant performance drops as it cannot properly schedule and handle memory accesses. The solution is to transpose the input vertex matrix and slightly change the access logic of the kernel. Instead of iterating over the columns of M_I and therefore getting x, y, and z, the kernel now iterates over the rows of M_I. As a result, v_{j_x} and v_{j+1_x} are now neighbored in the memory (as C and C++ are row-major languages), and therefore, memory accesses are now fully coalesced (cf. Figure 3.5).

The second optimization that we want to address here is the usage of FMA operations. They can play an important role whenever two floating-point numbers are first multiplied and then added to a third number. Since floating-point arithmetic requires the result to be rounded to the most proximate number, a multiplication followed by an addition induces two separate rounding operations:

$$(x, y, z) \mapsto \text{rnd}(\text{rnd}(x * y) + z)$$

However, FMA omits the first, inner rounding step:

$$\text{FMA} : (x, y, z) \mapsto \text{rnd}(x * y + z)$$

Therefore, both calculations are carried out with full precision before rounding. Hence, the goal here is to increase the precision of the calculated results and decrease the computational intensity.

These optimizations can be exploited on Nvidia and AMD GPUs. There are still minor bottlenecks. One is the fact that GPUs have much more 32bit floating-point compute units (FP32) than double precision units (FP64). The ratio for Nvidia GPUs is usually 32:1 or 64:1. As a result, Nsight Compute reports that our kernel overutilizes the FP64 pipeline but does not use a single FP32 unit. We still decide to have it like that because we do not want to introduce additional mathematical errors in the simulation by only using single-precision floating-point numbers. However, preconditioners could also be calculated using FP32, which is called “mixed-precision”. This is out of the scope of this thesis but might be investigated in follow-up work.

Listing 3.1 shows the final source code of our assembly kernel, which is declared using the Ginkgo macro `GKO_KERNEL` and dispatched using internal Ginkgo functions. It takes the desired RBF kernel as a lambda function, which is implemented to run on CPU and GPU. Besides the GPU-optimized variant, we also offer a CPU counterpart that does not need the transpose of the input vertices. We omit its listing here.

```
// Preparation for the kernel call
// Input vertices are stored in a transposed order on the GPU to allow for
// iterating over the different dimensions.
// The same happens for the output vertices.
for (std::size_t i = 0; i < inputMeshSize; ++i) {
    for (std::size_t j = 0; j < meshDim; ++j) {
        inputVertices->at(j, i) = inputMesh.vertices().at(i).rawCoords()[j];
    }
}

// Snippet of the assembly kernel code that is declared
// using the Ginkgo macro GKO_KERNEL
GKO_KERNEL(auto i, // Thread ID in x direction
           auto j, // Thread ID in y direction
           auto N, // Number of Vertices, i.e., matrix size into one direction
           auto dataDimensionality, // Whether vertices are 2D or 3D
           auto mtx, // Pointer to matrix Phi (or A_M)
           auto supportPoints, // First vertex set
           auto targetPoints, // Second vertex set
           auto f, // Lambda RBF function
           auto rbfParams, // Call parameters for f
           auto supportPointRowLength, // Number of support vertices
           auto targetPointRowLength) // Number of target vertices
{
    double dist = 0.0; // Euclidean distance between two vertices
    double y; // Stores distance between two coordinates of the same dimension

    // Distance in each dimension between two vertices
    for (size_t k = 0; k < dataDimensionality; ++k) {
        // All threads access the x dimension simultaneously, than y, than z
        // Thread coordinates (i, j) are used to identify the correct two vertices
        y = supportPoints[k * supportPointRowLength + j]
            - targetPoints[k * targetPointRowLength + i];
        dist = fma(y, y, dist); // Call to FMA instead of adding squared y
    }

    dist = sqrt(dist);
    // Every thread works on exactly one matrix entry; identified by (i, j)
    mtx[i * N + j] = f(dist, rbfParams);
}
```

Listing 3.1: Optimized assembly kernel that allows for coalesced memory accesses to the vertex arrays and uses FMA.

3.4.2 Solver Integration

This section describes the implementation strategies for the different solvers.

Polynomial Solver First of all, there is the problem of solving the system of polynomial contributions (cf. Algorithm 2.1):

$$f = Q_M \cdot \beta \tag{3.7}$$

Since $Q_M \in \mathbb{R}^{K\times(d+1)}$, where d denotes the dimension of the input vertices plus an additional constant contribution, the resulting system is usually over-determined unless $K = d$ which should nearly never happen. This implies that iterative solvers such as CG and GMRES cannot be applied because Q_M is not invertible.

There are two options to tackle this problem:

1. Use a direct matrix decomposition such as QR which can handle over-determined systems.
2. Solve the normal equation $Q_M^T \cdot Q_M \cdot \beta = Q_M^T \cdot f$, which always results in a square matrix.

We decide to use a plain CG solver in order to solve the normal equation for the separate polynomial system due to the fact that Ginkgo does not offer a least-squares solver as of writing this thesis, and our QR implementation is currently restricted to GPUs and therefore not available in our CPU implementation.

Iterative Solvers The workflow of iterative solvers, which are provided by Ginkgo, in preCICE is straightforward and involves the following steps:

1. A one-time initialization is done by generating the solver object using the already assembled, constant Φ, including the configuration such as the number of iterations. This takes place in `computeMapping()` .
2. In the recurring mapping process, i.e., in `map()` , the latest right-hand side f is transferred to the device.
3. The solver, generated on Φ, is called using f as right-hand side and calculates λ.
4. λ is transferred back to the host and put into a vector data structure provided by the Eigen library, which is then used by preCICE for post-processing.

QR Decomposition We take the cuSolver library to implement a QR decomposition for GPUs. AMD provides a tool called "hipify" that can convert CUDA code to HIP code. It can therefore convert this implementation s.t. it also works on AMD GPUs. The solver itself is implemented as a function that can be called from `GinkgoRadialBasisFctSolver`. Having it this way allows us to combine every GPU solver in one class that can be adapted in the future.

Listing 3.2 shows the conceptual code of the QR decomposition, omitting function arguments for better readability. The Ginkgo data structures can be reused as they act as a wrapper around CUDA memory allocations and transfers. The cuSolver library can access the underlying memory pointers as they reside in the VRAM.

The workflow of the QR decomposition is as follows:

1. The input matrix (called `A_Q` in the code) is transposed since cuSolver assumes column-major storage.
2. In contrast to self-written CUDA kernels, the execution policy is hidden behind library functions that ought to calculate the best execution policy, i.e., the grid and thread structure. The `cusolverDnXgeqrf_bufferSize()` and `cusolverDnDorgqr_bufferSize()` functions calculate buffer sizes s.t. the necessary memory can be allocated for the calls to the actual decomposition functions.
3. `cusolverDnXgeqrf()` calculates an implicit QR decomposition by storing R in the upper triangular part of `A_T` and Q as a sequence of Householder vector in the lower triangular part of `A_T`.
4. To get the row-major R, R is explicitly transposed again and stored in a corresponding variable.
5. `cusolverDnDorgqr()` calculates Q explicitly as it is needed multiple times for recurring mappings in preCICE. Its row-major variant overrides the input matrix `A_Q`.

The output of this procedure is then given to the Ginkgo upper triangular solver, which repeatedly calculates the solution for the following system:

$$R \cdot \lambda = Q^T \cdot f \tag{3.8}$$

```
// Since cuSolver assumes column-major storage, we need to transpose matrix A_Q, which
// stores initially the input matrix and will store matrix Q in the end
auto A_T = gko::share(GinkgoMatrix::create(exec, gko::dim<2>(A_Q->get_size()[1], A_Q->
get_size()[0])));
A_Q->transpose(gko::lend(A_T));

// Query working space of geqrf and orgqr
cusolverDnXgeqrf_bufferSize(...);
cusolverDnDorgqr_bufferSize(...);

// Compute QR factorization
cusolverDnXgeqrf(...);
cudaDeviceSynchronize();

// Copy transposed A_T to R s.t. the upper triangle corresponds to R
A_T->transpose(gko::lend(R));

// Compute Q
cusolverDnDorgqr(...);
cudaDeviceSynchronize();

// Copy row-major Q to A_Q
A_T->transpose(gko::lend(A_Q));
```

Listing 3.2: The QR decomposition that is implemented using cuSolver. Its function names are derived from LAPACK functions. We omit function arguments for better readability.

3.4.3 Matrix-Free Approach

One limiting factor is the size of VRAM that modern GPUs offer. Our implementation has a quadratic memory requirement when it comes to storing the matrices. We have to find a remedy that allows us to deal with memory limitations. We already discussed “CUDA unified memory” in Sec. 2.2.2, which, however, also has quadratic memory requirements.

One very common approach is to use so-called “matrix-free” methods. Matrix-free methods do not store the coefficient matrix explicitly, but they express the system matrix as operators. Entries of the matrix are computed on-the-fly; the assembly and matrix-vector multiplication are merged into one operation. This especially reduces memory requirements and could potentially leverage optimal caching behavior implemented on modern hardware. Hence, the concept is very different to “CUDA unified memory” and makes it interesting for comparison.

Matrix-free methods have already been tested on accelerator cards before. Ljungkvist (2014) presents a matrix-free finite-element operator implementation specifically tailored for GPUs. The author argues that it is favorable to rely on the throughput-oriented architecture of GPUs to handle the increasing computational complexity of matrix-free methods.

$$\begin{bmatrix} \varphi_{1,1} & \varphi_{1,2} & \cdots & \varphi_{1,K} \\ \varphi_{2,1} & \ddots & & \varphi_{2,K} \\ \vdots & & \ddots & \vdots \\ \varphi_{K,1} & \varphi_{K,2} & \cdots & \varphi_{K,K} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix}$$

Figure 3.6: A matrix-vector multiplication using a GPU as it is used in the matrix-free implementation. Having one thread per matrix row keeps write accesses to b as efficient as possible and allows for coalesced write transactions.

Our matrix-free implementation integrates very well into our existing Ginkgo workflow. It relies on Ginkgo's ability of creating custom matrix data structures that can be used as linear operators in all iterative solvers.

Recall from Sec. 2.4 that solving the system $f = \Phi \cdot \lambda$ in an iterative fashion reuses the system matrix Φ in every single iteration in matrix-vector multiplications. Instead of having Φ precomputed in the memory, it is now recomputed every iteration. Our `RBFMatrix` needs two components: its assembly routine and a multiplication method for the general product $A \cdot x = b$. The already implemented assembly routine described above can be used again. The matrix-vector product must be optimized w.r.t. to CUDA and HIP. Instead of having one thread for each matrix entry, we now use one thread for an entire row of the matrix. Looking at Figure 3.6, every entry in b is handled by one thread only. Having it this way avoids memory access conflicts since no two threads have to write to the same memory address. Even more, b can be kept local for the entire summation process and needs to be flushed to global memory only once upon finalization (cf. Figure 3.7). Global memory access is then also coalesced as neighboring threads access neighboring memory addresses. Finally, vertex x_i, which is constant for the entire matrix row i, can be loaded once into local memory and reused throughout the multiplication process. We implement this matrix-free procedure as traditional CUDA kernel by tagging it with the keyword `__global__`. This is inspired by example implementations provided by Ginkgo[10].

Listing 3.3 shows the matrix-free kernel than computes every $\varphi_{i,j}$ upon execution and multiplies it with the corresponding vector element x_j.
The expression `blockIdx.x * blockDim.x + threadIdx.x` is standard CUDA code and automatically available in every kernel. The variables provide block ID, block size, and ID of the executing thread, as suggested by their names, and identify every thread uniquely.

[10] https://github.com/ginkgo-project/ginkgo/tree/develop/examples/custom-matrix-format

```
template <typename ValueType, typename EvalFunctionType>
__global__ void multiply_kernel_impl(std::size_t M, // Number of rows
                                     std::size_t N, // Number of columns
                                     ValueType *v1, // First set of vertices
                                     ValueType *v2, // Second set of vertices
                                     ValueType* x,  // Vector x of the product
                                     ValueType *b, // Result of the product
                                     EvalFunctionType f, // Lambda to RBF function
                                     RadialBasisParameters params,  // RBF parameters
                                     size_t v1RowLength, // Columns of set v1
                                     size_t v2RowLength) // Columns of set v2
{
    // Every thread in the 1D grid calculates its part of output vector b
    auto i = blockIdx.x * blockDim.x + threadIdx.x;
    // The summation of a row is kept local
    ValueType localB = 0;

    ValueType dist = 0;
    ValueType y;

    // Prefetched support point which is same for the entire row
    ValueType prefetchedEvalPoint[3];

    if(i < M){
        prefetchedEvalPoint[0] = v1[i];
        prefetchedEvalPoint[1] = v1[v1RowLength + i];
        prefetchedEvalPoint[2] = v1[2 * v1RowLength + i];

        for(size_t j = 0; j < v2RowLength; ++j){

            dist = 0;
            for (size_t k = 0; k < 3; ++k) {
                y = prefetchedEvalPoint[k] - v2[k * v2RowLength + j];
                dist = fma(y, y, dist);
            }

            // The product of Phi(radius, params) times x_j is added to b
            // It does not use a precomputed value of Phi
            localB =  fma(f(sqrt(dist), params), x[j], localB);
        }

        // On kernel exit, the locally computed entries of b are
        // written into global memory
        b[i] = localB;
    }
}
```

Listing 3.3: The matrix-free kernel that combines matrix assembly and matrix-vector multiplication. It is executed every time the `apply()` function is called on the object.

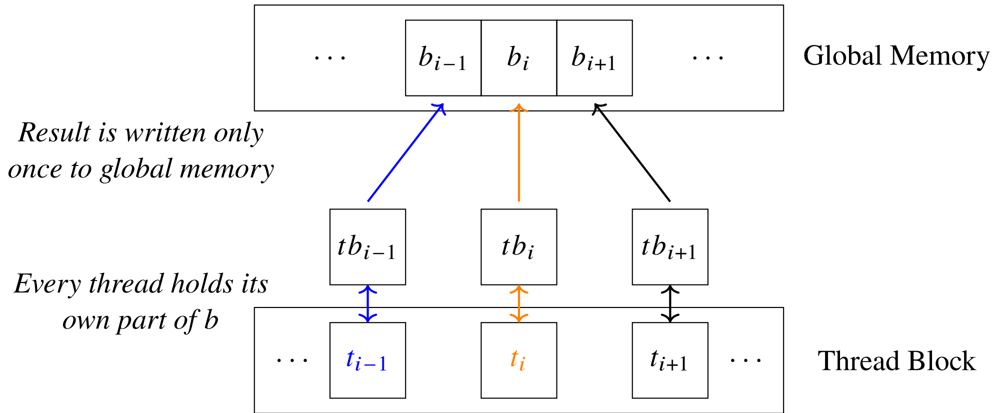

Figure 3.7: Reducing global memory transactions is possible by calculating every part of b locally and writing the result only once to global memory.

3.4.4 Further Considerations

Optimization of GPU Usage Besides the implemented concepts described above, we want to shortly discuss some further possible concepts and why we do not investigate them further. Similar to previous optimizations, these shortcomings are mostly motivated by memory bottlenecks.

Our first consideration is to use single-precision instead of double-precision floating-point numbers in our GPU implementation. Not only are there 32 to 64 times as many FP32 units on Nvidia GPUs as FP64 units, but they also just consume half of the memory. However, first and foremost, preCICE uses double precision almost everywhere because simulation accuracy highly depends on floating-point arithmetic. Using FP32 instead of FP64 would cause additional mathematical errors which we want to avoid.

The next consideration includes the usage of so-called sparse matrix formats such as “compressed sparse row (CSR)”. Sparse matrix formats do not store every value explicitly, including zero entries. Instead, they only store non-zero entries by making use of additional arrays in which non-zero values and their corresponding positions in the matrix are stored. However, it usually is not clear beforehand whether the RBF matrix turns out to be sparse or dense. If it is dense, the additional index pointers will use more memory than plain dense matrix formats. As a result, we do not consider sparse matrix formats further since we do not know the sparsity pattern until every RBF value has been evaluated. Moreover, the sparsity pattern is highly irregular and depends on the vertex distribution of the mesh. Additionally, the conversion from dense to sparse matrix formats produces runtime overhead and makes efficient memory management even more difficult.

Another strategy is to assemble both matrices on different GPUs if available. However, there are only rare cases in which GPUs can directly exchange data without having to transfer the data to the host first. This topic is beyond the scope of this thesis.

A last solution strategy to tackle memory problems might be to move the matrices Φ and A_M from RAM to VRAM and back in an alternating fashion. However, this does not help if either matrix is already too large to fit into VRAM.

Other Data Mapping Methods As already mentioned in Sec. 1.2, there are further algorithms for data mapping implemented in preCICE, namely "nearest-neighbor" and "nearest-projection mapping". In contrast to RBF mapping, these techniques do not rely on the solution of a system of linear equations. Instead, nearest-neighbor simply copies the value of the spatially nearest vertex of the input mesh onto the target vertex of the output mesh. Nearest-projection makes use of topological information on meshes, e.g., triangles formed by mesh vertices. It calculates a projection point for the target vertex, followed by an interpolation method such as barycentric interpolation. Both algorithms are highly data-parallel since they act on all target vertices independently. This makes them suited very well for GPUs. For instance, Heidrich (2005) proposes an efficient algorithm for barycentric interpolation that works well on accelerator cards. However, we decide to not further look at the parallelization of these algorithms in preCICE as both mapping methods are already fast enough. As a result, focusing on RBF data mapping has much more potential for significant speedups in preCICE.

4 Experimental Evaluation

This chapter evaluates our implementation w.r.t. to performance and efficiency. Since this work deals with “efficient” application of accelerator cards, we look at qualitative and quantitative results. The first one ensures that resources are efficiently used, whereas the second one shows the superiority in terms of runtime, which highly benefits the users of preCICE.

In order to measure these different aspects, we make use of two test cases: one plain data mapping test case and one coupled simulation. We publish all test case setup and measurement result files in a dedicated data repository (Schrader et al., 2023).

4.1 Simulation Setup and Test Cases

We collect different parameters to evaluate how good or bad different algorithms and their configuration perform within preCICE data mapping procedures. These parameters contain among others but not exclusively the time which is required to set up system matrices and solve the interpolation system, the time that it takes to calculate the output values and memory transfer times between RAM and VRAM, which is of high relevance in the context of GPU programming.

We make use of the compute servers provided by the IPVS at the University of Stuttgart. Our performance benchmarking hardware setup is made up of the following components:

- CPU: AMD EPYC 7763 64-Core Processor
- GPU 1: Nvidia A100 with 40GB VRAM
- GPU 2: Nvidia RTX 3090 with 24GB VRAM
- GPU 3: AMD Radeon Pro VII with 16GB VRAM

We use the Nvidia A100 card as a state-of-the-art accelerator, whereas the RTX 3090 is used for direct comparison against the AMD Radeon Pro VII in order to have a similar setup between Nvidia and AMD. For qualitative analysis, for which no runtime measurements are required, an Nvidia RTX 2070 Super with 8GB VRAM is used.

In the following, we denote the first coupling participant as $\mathcal{S}_1$ and the second as $\mathcal{S}_2$.

Figure 4.1: 3D model geometry of the turbine blade provided by ASTE. It is used to generate the meshes for the turbine mapping test cases.

ASTE We use the ASTE[1] framework (Chourdakis et al., 2022) to measure the performance of our mapping implementations. ASTE is part of the preCICE tooling that allows the setup of simulation cases and monitoring them. This can be achieved by either creating a new simulation or by replaying an already existing one. Furthermore, ASTE helps to debug coupled code and develop new adapters for preCICE. Our setup is similar to Chourdakis et al. (2022) which makes our results comparable to the current preCICE performance.

4.2 Turbine Blade Grid Mappings

ASTE also supports the creation of different grids with mesh size h. On top, it provides already usable meshes in the shape of a turbine blade[2] (cf. Figure 4.1). We use these meshes, each with a different mesh width h, to measure the matrix assembly performance as well as the performance on solving the interpolation problem, i.e., calculating the data mapping between a coarser mesh M_1 and finer mesh M_2. The problem size and memory requirements increase with decreasing mesh width h. Sticking to the categorization of Chourdakis et al. (2022), the meshes are divided into *coarse* and *fine* ones. Table 4.1 lists all turbine mesh configurations and their corresponding category. Our goal is to use an experimental setup which close to the reference one. However, some changes are necessary due to the fact that GPUs usually have much less memory available than CPUs. It is mentioned whenever we do not stick to the setup.

The mapping error itself is calculated using a discrete L_2-norm:

$$E = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(f_i - g(v_i))^2} \tag{4.1}$$

The overall error is the square root of the mean squared error which takes every output vertex v_i into consideration and calculates the L_2 norm between the real values f and interpolated values $g(v_i)$.

[1] https://github.com/precice/aste

[2] https://grabcad.com/library/wind-turbine-blade--4

h	Vertices	Category
0.03	438	*coarse*
0.02	924	*coarse*
0.01	3,458	*coarse*
0.009	4,302	*coarse*
0.008	5,310	*coarse*
0.006	9,588	*coarse*
0.004	21,283	*coarse/fine*
0.003	38,112	*fine*
0.002	84,882	*fine*
0.0014	172,803	*fine*
0.001	338,992	*fine*
0.0007	691,426	*fine*
0.0005	1,354,274	*fine*

Table 4.1: The different turbine blade meshes provided by ASTE. They differ in mesh size h.

Finally, a reference function is required which provides the values for each vertex. The so-called “Franke” function is used throughout all turbine mapping experiments, which is a widely used test function for scattered data interpolation (e.g., cf. Idais et al. (2019)):

$$\begin{aligned} F(v) = 0.75 * e^{-((9*v_x-2)^2+(9*v_y-2)^2+(9*v_z-2)^2)/4} + 0.75 * e^{-((9*v_x+1)^2/49+(9*v_y+1)/10+(9*v_z+1)/10)} \\ +0.5 * e^{-((9*v_x-7)^2+(9*v_y-3)^2+(9*v_y-5)^2)/4} - 0.2 * e^{-((9*v_x-4)^2+(9*v_y-7)^2+(9*v_z-5)^2)} \end{aligned} \tag{4.2}$$

4.2.1 Matrix Assembly

This section deals with the performance analysis and measurements of our matrix assembly kernel, which is executed once for every matrix in the `computeMapping()` method. Firstly, we look at technical details. Secondly, we compare the optimized GPU variant against the OpenMP implementation, the sequential Eigen version, and the PETSc assembly routine which uses a preallocation heuristic to optimize matrix assembly. We use the ASTE turbine mapping use case throughout our assembly measurements since it allows us to easily increase or decrease mesh width and hence problem size. There a no measurements for the case of separate polynomials because the matrices Q_M and V_M are much easier to assemble as they are basically only memory copies from vertices to matrix elements.

Qualitative Analysis We start by analyzing the technical performance of our fully optimized kernel using Nsight Compute. The mesh width of the input turbine is set to $h = 0.006$, whereas the output turbine uses a mesh width of $h = 0.004$.

The roofline diagram of the GPU kernel for the assembly of Φ is shown in Figure 4.2. A roofline diagram displays the performance of a compute kernel relative to bottlenecks induced by a specific architecture. It has been first introduced by Williams et al. (2009). The arithmetic intensity is

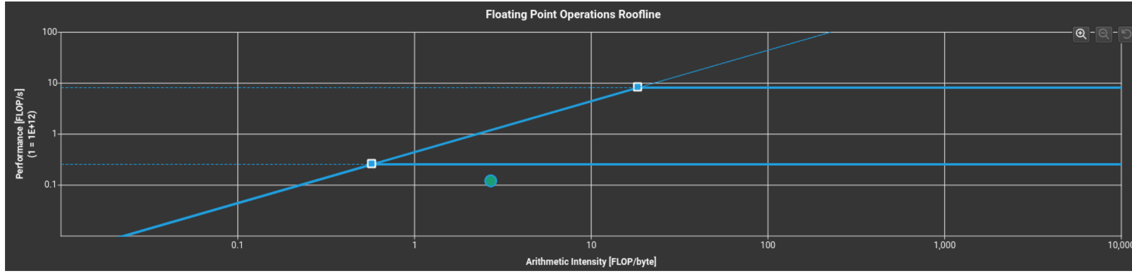

Figure 4.2: Roofline diagram of the matrix assembly kernel. The kernel is compute-bound and reaches almost the maximum performance of the FP64 compute units.

shown on the x-axis, which is the computational workload in floating-point operation (FLOP) per transferred byte, i.e., the ratio between "work" and "memory traffic". The performance in FLOP/s is represented on the y-axis using a logarithmic scale. There are two parts of the diagram, a diagonal upper bound which is the upper bound induced by memory bottlenecks, and the horizontal bound which is the computational maximum. The so-called "ridge point" is the point at which a compute kernel becomes compute-bound, i.e., the point in the diagram at which the slope is no longer diagonal but horizontal instead. Therefore, it points to the minimum arithmetic intensity that a kernel should achieve in order to achieve the peak computational performance of the hardware. Furthermore, the upper horizontal bound is the FP32 peak performance, and the lower one is the FP64 performance, which is the theoretical peak performance that can be achieved on this hardware.

The performance of our assembly kernel is visualized by the green dot. It has been executed on the Nvidia RTX 2070 Super. Since we only deal with double-precision floating-point numbers, there is only a report for FP64 performance, leaving out FP32. The arithmetic intensity is 2.73 FLOP/Byte, and the achieved performance is $0.116 * 10^{12}$ FLOP/s. First and foremost, the diagram shows that our assembly kernel exceeds the minimum arithmetic intensity that is required in order to be able to reach the peak performance of the GPU. This means that it is almost fully bound by the compute capabilities of the GPU and not by time-expensive memory operations. According to the diagram, there is still room for improving the peak performance of our kernel as there is still some space between the green dot and the upper bound. However, we cannot execute more operations than actually needed in our kernel. Since we already do the entire calculation of Φ on the GPU, there is no further work that can be done in the assembly process. We conclude that we achieve good performance with our matrix assembly as it provides enough computational workload s.t. the kernel is not limited to memory transfers. In general, reaching the upper bound with a compute kernel is difficult and would require more complex computations per byte.

Figure 4.3 displays the detailed memory analysis produced by Nsight Compute. It monitors every single memory access executed by our kernel, distinguishing different memory spaces on the card. Our kernel only accesses global memory since broadcasting values to an entire group of threads is more efficient than putting them into shared memory first. It becomes clear that the L1 hit rate is far from optimal, i.e., most of the requested values are not found in L1 cache. However, the slower L2 cache reaches a hit rate of over 98% which is almost optimal. Having a high cache hit rate reduces the time it takes the threads to wait for data significantly. Hence, we conclude that we not only have a well-behaving kernel in terms of computational intensity but also in terms of L2 cache management. Improvement of L1 cache usage might leave minor room for improvement.

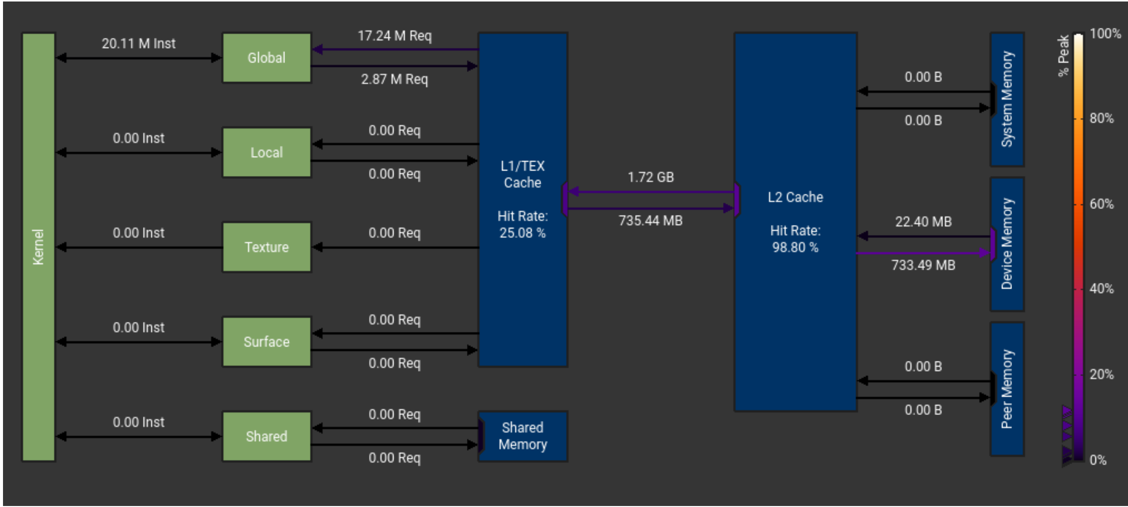

Figure 4.3: Memory analysis of the matrix assembly kernel. Whereas the L1 cache hit rate is low, the L2 cache hit rate is almost 100% and therefore optimal.

This detailed analysis has shown that our implementation of the matrix assembly routine is nearly optimal and therefore reflects an efficient way of setting up the RBF interpolation systems in preCICE. As a result, the matrix assembly routine can be used as a highly efficient building block for our matrix-free approach that we discussed in Sec. 3.4.3 and which requires all RBF values to be computed every time it is used.

Next, we look at the quantitative performance in the context of preCICE and compare it against the alternatives. Every configuration is executed five times and the numerical average across all runs is taken as the final measurement value.

Benchmarking: Assembly of Output Matrix (*fine* Series) We start by comparing the runtime of the matrix assembly kernel using high-end hardware. In this test case, we look at matrix A_M that is used for calculating the interpolated values:

$$g = A_M \cdot \lambda \tag{4.3}$$

To save memory, $\mathcal{S}_1$ uses $h = 0.01$. Note that this is a rather artificial setting in which one solver uses a pretty coarse mesh. However, this allows us to test the edge case of $h = 0.0005$ on the GPU. In the future, available memory on accelerator cards will increase further, i.e., it will be possible to assemble even larger matrices on the GPU.

We create four settings to compare against each other:

- One with the Nvidia A100 GPU that executes our self-written kernel.
- Another one using OpenMP with 64 threads and also our own kernel.
- A distributed participant using PETSc that uses 48 MPI ranks for $\mathcal{S}_1$ and 48 MPI for $\mathcal{S}_2$, i.e., the output mesh is divided heuristically into 48 parts and distributed among the ranks, which lowers the overall runtime compared to what it would be using only a single rank.

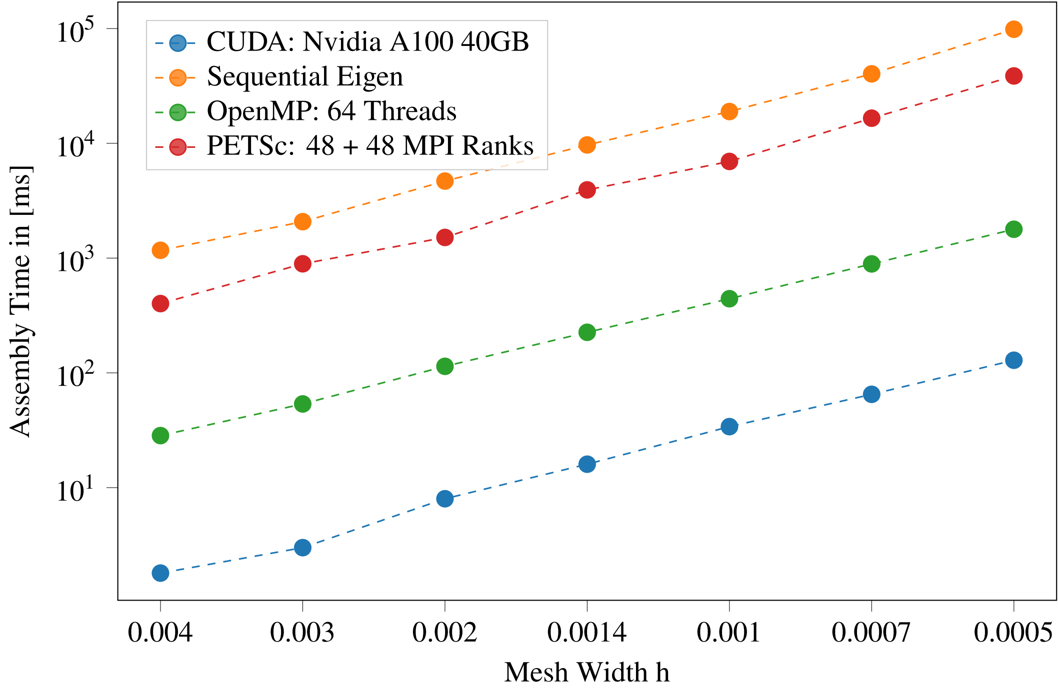

Figure 4.4: Assembly time of A_M for the meshes of *fine* series depending on mesh width h in ms (log-scale). The CUDA assembly kernel is the fastest out of all implementations.

- The sequential Eigen implementation, which runs on one core.

To enforce as many non-zero entries as possible, we use global thin plate spline (G-TPS) ($||\vec{x}^2|| \log(||\vec{x}||)$) as underlying RBF, which has global support. This enables us to see how each executing device behaves under maximum workload, both in terms of arithmetic intensity and memory usage.

Figure 4.4 shows the runtime results of each of the four cases. h is displayed on the x-axis, the y-axis shows the time it took in ms using a logarithmic scale.

It can be seen that our own kernel implementation, including the highly optimized CUDA kernel, clearly outperforms both existing variants. Whereas the GPU does not even need 100 ms to set up the matrix that belongs to mesh sizes $h_1 = 0.01$ and $h_2 = 0.0005$, i.e., a matrix with $3,458 \cdot 1,354,274$ independent entries, the sequential Eigen implementation almost requires 100 s to just fill all matrix entries. The distributed, multi-core PETSc routine also works for more than 10 s until the matrix is done. In contrast, the multi-core OpenMP executor provided by Ginkgo only needs around 1 s using 64 Threads. Even for smaller values of h, the advantage of accelerator cards over sequential and parallelized CPU variants can be observed. The CUDA-accelerated assembly is faster by an order of magnitude 10 compared to the OpenMP variant and $100 - 1,000$ times faster than the assembly procedures from the Eigen and PETSc implementations. Another important point is that the runtime difference stays constant w.r.t a logarithmic scale. This implies the difference grows exponentially on a linear scale. In conclusion, when it comes to large mathematical problems, the (high-end) GPU provides a huge speedup already during the setup phase of the interpolation.

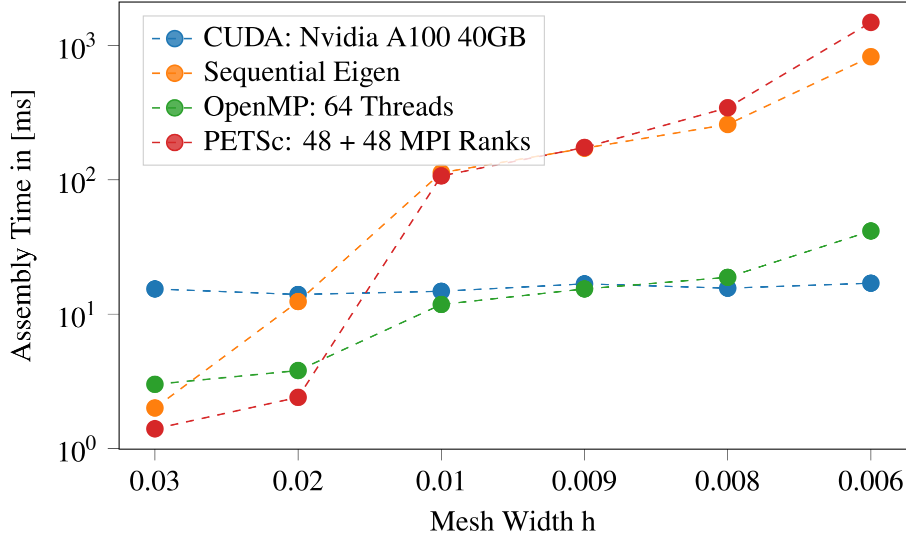

Figure 4.5: Assembly time of Φ for the meshes of the *coarse* series depending on mesh width h in ms (log-scale). The CUDA kernel is the fastest version for all mesh sizes smaller than $h = 0.01$.

Benchmarking: Assembly of Input Matrix (*coarse* Series) This paragraph investigates if the significant speedup of GPUs can be observed for matrix assembly processes of the *coarse* series as well. Here, we look at the assembly of Φ. $\mathcal{S}_1$ uses all meshes from the *coarse* series. As opposed to the output mapping matrices, the input matrices are quadratic and symmetric. Sequential implementations can benefit from this fact as they only need to evaluate one triangular half which results in $O(\frac{1}{2}K^2)$ memory accesses. Our Ginkgo implementation does not take that into account, it always uses a full 2D thread grid with $O(K^2)$ memory requirements. This is because GPGPU does not allow for triangular thread grids or similar topologies; it only supports rectangular thread grids. Figure 4.5 visualizes the time measurement results in milliseconds. In contrast to the case before, the GPU implementation is not faster for every mesh size; it is the slowest for the two coarsest mesh sizes $h = 0.03$ and $h = 0.02$. This is due to the overhead that is induced by allocating memory space in the VRAM and copying data to it. However, looking at the absolute assembly times for these two meshes, no implementation needs more than 15 ms to assemble them, which makes the GPU overhead seem negligible. Starting with $h = 0.01$, the CUDA and OpenMP kernels become the fastest variants. Moreover, for $h = 0.006$, the CUDA kernel is about 100 times faster than the Eigen and PETSc implementation by only requiring slightly more than 10 ms instead of about 1 s. It is also about twice as fast as the OpenMP kernel.

Benchmarking: Assembly with CUDA Unified Memory As already discussed in the last paragraphs, the setting of having one solver use a very coarse mesh is a rather unusual one, especially for $\mathcal{S}_2$. Therefore, we look at a more common setting in which $\mathcal{S}_1$ holds the turbine mesh with $h = 0.006$ and $\mathcal{S}_2$ iterates over every mesh of the *fine* series. This, however, requires CUDA to enable the unified memory mechanism in order to allocate more memory than provided by the VRAM. It imposes a bottleneck that leads to a significant increase in runtime. Since the Ginkgo OpenMP executor has shown to be the strongest alternative, it is also used here to compare it against CUDA unified memory. Results are shown in Figure 4.6. The OpenMP implementation

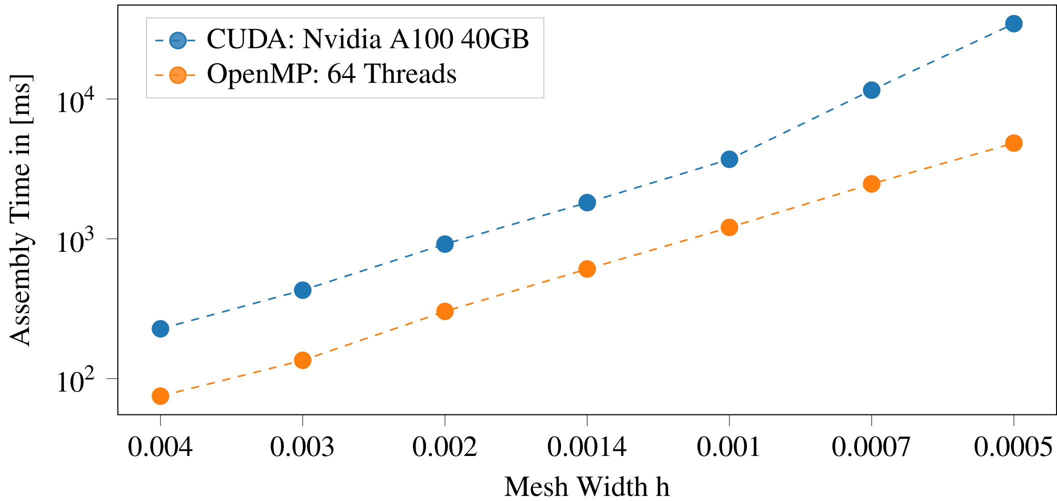

Figure 4.6: Assembly time of A_M for the meshes of the *fine* series depending on mesh width h in ms using CUDA unified memory (log-scale). The memory allocation of the CUDA unified memory model causes significant overhead.

using 64 threads requires less time than the counterpart using CUDA unified memory. Whereas for $h = 0.004$ the difference is rather low (~ 150 ms), the difference for $h = 0.0005$ is about 30 s, which is significant and non-negligible. We draw the conclusion that using CUDA unified memory allows for larger sizes of Φ, however, the additional allocation and data transfer time are very high and therefore a clear drawback of this approach.

Benchmarking: Nvidia vs. AMD GPU Not only does our work support Nvidia but also AMD GPUs. We investigate how well our GPU kernel, which has been optimized using hints from Nvidia Nsight Compute, performs on AMD devices.

Figure 4.7 visualizes the results of using an AMD Radeon Pro VII and comparing it against the Nvidia RTX 3090. Although the kernel has been written with CUDA in mind, the AMD counterpart is even faster when it comes to matrix assembly. It takes less than 1 ms to assemble the turbine matrix corresponding to $h = 0.006$. Due to memory constraints on both cards, we do not test the *fine* series here. We can still draw the conclusion that the AMD GPU clearly outperforms the Nvidia card. Still, both cards are very fast compared to the non-GPU cases. Therefore, both GPUs are suited very well for data-parallel matrix assembly.

4.2.2 Mapping Computations

The last section only considered matrix assembly time. However, we also need to look at the runtime of solving these (recurring) systems as well as the mathematical error. In conclusion, we get a better understanding of when it is feasible to use GPUs in favor of (sequential) CPU implementations.

We start by measuring the performance of `computeMapping()` and `mapConsistent()`. However, they are not equivalent between the different solver variants. The sequential, direct Eigen solver computes a Cholesky decomposition only once, the QR decomposition factorizes Φ into an

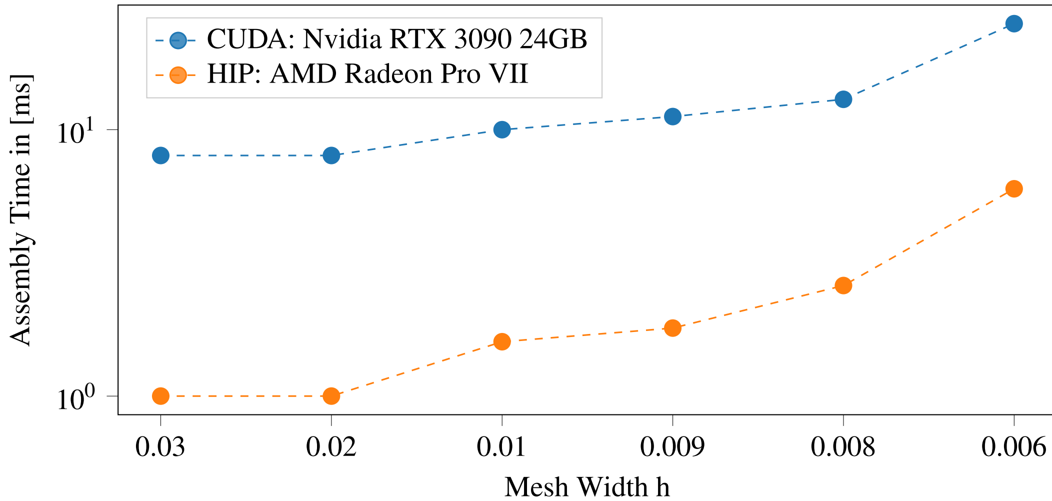

Figure 4.7: Assembly time of matrices of the *coarse* series depending on mesh width h in ms (log-scale). The AMD Radeon Pro VII performs better than the Nvidia RTX 3090 by almost a factor of ten.

orthogonal matrix and an upper triangular matrix, whereas the iterative Ginkgo implementation only needs to assemble Φ and A_M as subsequent computations depend on the right-hand side vector which is available when one participant sent their mesh data. In contrast, `mapConsistent()` requires the iterative Ginkgo solver to always start with an initial guess λ_0 and iteratively approach a solution, whereas the direct solvers, i.e., Cholesky and QR, can make use of the already computed decomposition and therefore only need to solve a triangular system via backward substitution. As a result, the global runtime of the entire simulation is additionally required in order to get a clear picture.

For all experiments of the *coarse* series, we define the relative residual reduction to be 10^{-9} in the case of iterative solvers and a maximum number of $10,000$ iterations. Hence, the iterative solvers are stopped if either the norm of the residual vector r has been decreased enough or $10,000$ iterations have been executed. For measuring the *coarse* meshes, $\mathcal{S}_1$ iterates over each coarse mesh from $h = 0.03$ to $h = 0.004$ and $\mathcal{S}_2$ holds the output mesh $h = 0.003$. This is different from the setup of Chourdakis et al. (2022), who map onto $h = 0.009$, as it is more realistic to have a finer mesh onto which data should be mapped. The *fine* series uses a residual reduction of 10^{-4} and varying maximum number of iterations because these systems are so large and ill-conditioned conditioned that executing $10,000$ CG or GMRES iterations becomes infeasible really fast. On top of that, it can be seen that a few iterations might already be sufficient to get a decently small error. The detailed setup is described for every experiment individually. We do not use every solver in every experiment. Instead, we focus on specific solvers that might be better suited than others for a specific case.

We use the Nvidia A100 GPU throughout all experiments.

Coarse Series: Sparse RBF This test case uses the compact polynomial C6 RBF which is defined as follows where p denotes the product between the input, which is a radius, and the inverse of the support radius:

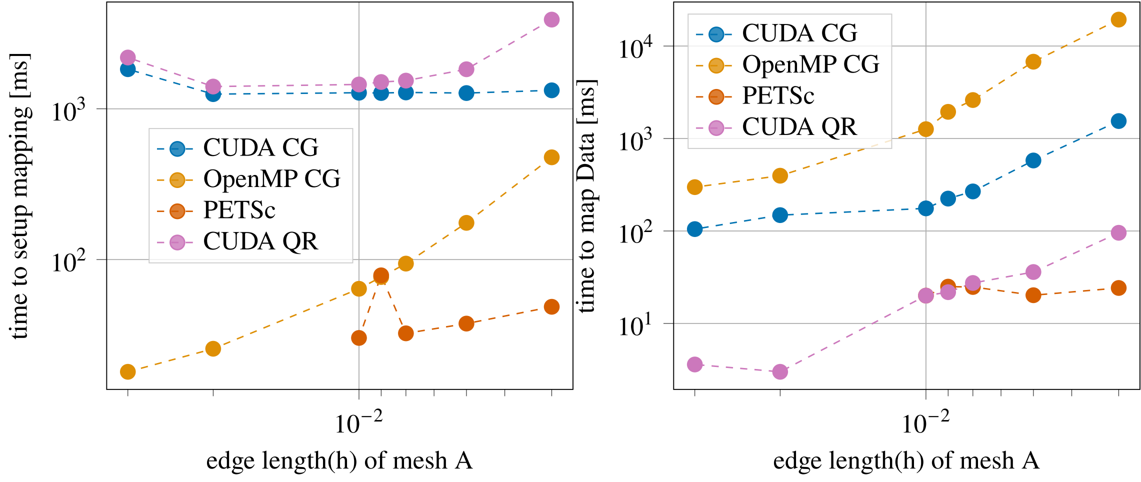

Figure 4.8: Results for compact polynomial C6 RBF. The left plot shows the initial setup phase duration in ms and the right side the recurring mapping time.

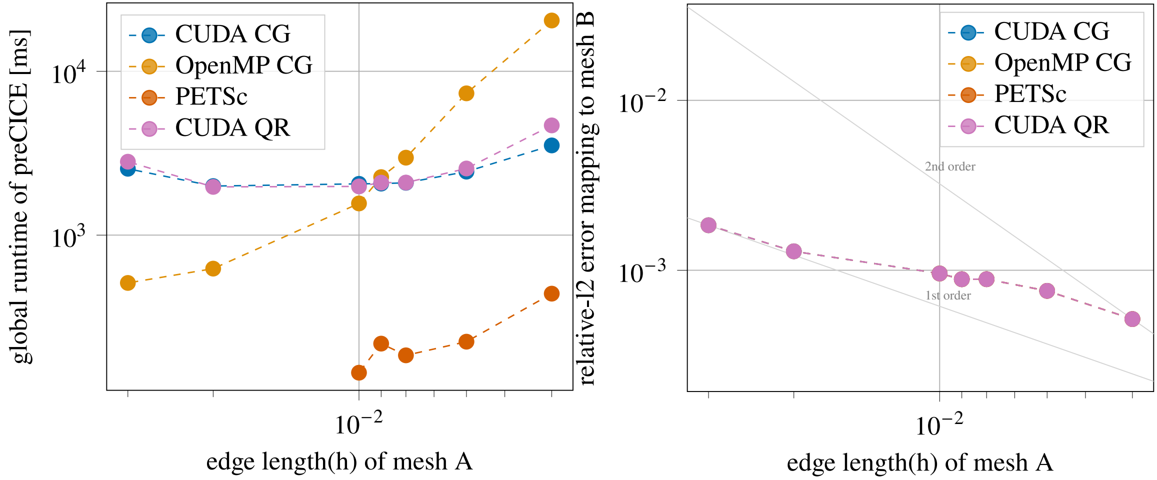

Figure 4.9: Results for compact polynomial C6 RBF. The left plot shows the overall runtime of preCICE in ms and the right plot the average discrete mapping error.

$$f(p) = \begin{cases} (1-p)^8 \cdot (32 \cdot p^3 + 25 \cdot p^2 + 8 \cdot p + 1) & \text{if } p < 1 \\ 0 & \text{else} \end{cases} \tag{4.4}$$

We set $3h$ as the support radius, i.e., the support radius decreases proportionally to the mesh width h. This produces sparse matrices in general with most entries being zero. Therefore, we mostly consider iterative solvers in this case since they are known to work best with sparse matrices. We still include the CUDA QR implementation to get a better understanding of whether it should be used for sparse matrices as well. In summary, we compare:

- CUDA CG

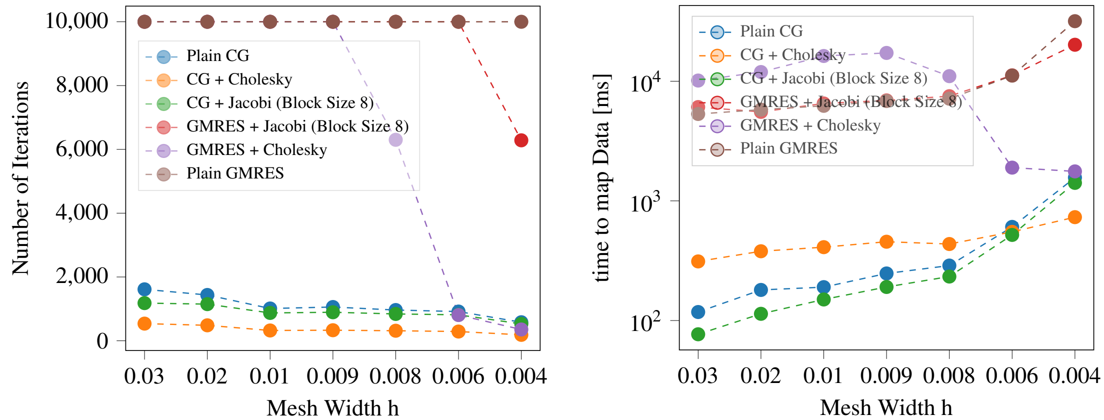

Figure 4.10: The left plot shows the number of iterations until the stopping criterion of 10^{-9} was reached for different iterative solver and preconditioner combinations. GMRES without a preconditioner did not hit the stopping criterion at all. The right plot displays the time it took preCICE to map the data. Preconditioners have a positive impact on the runtime for finer meshes.

- OpenMP CG with 64 Threads
- CUDA QR
- PETSc GMRES with 48 + 48 Ranks

Figure 4.8 and Figure 4.9 show four plots: The runtime of `computeMapping()` and `mapConsistent()` separately, the global runtime of preCICE and the mapping error. The mapping error is the same for all four variants, which proves that our GPU-based and OpenMP-based implementations produce the correct results. The PETSc implementation using 48 ranks for distributed GMRES solving works very well and produces the lowest global runtime. However, it fails for $h = 0.03$ and $h = 0.02$ as it cannot calculate a solution for the system due to divergence, hence its measurement points are missing in the plots. The OpenMP implementation using CG on 64 threads takes the most time for the largest problem sizes of the *coarse* series. Whereas it is faster in setting up the system matrices, it loses most time during iterating the system. This leads to the conclusion that a huge amount of matrix products are computed faster on GPUs. The CUDA QR implementation takes a bit longer to set up since it not only assembles the matrices but also decomposes them into Q and R. Still, the difference between the iterative CG and CUDA QR is very small s.t. we can conclude that the QR solver spends most of its time on matrix assembly and factorization. The recurring mapping is faster by an approximate magnitude of order 10 when using QR. Nevertheless, it has to be noted that the residual criterion of 10^{-9} for the iterative solvers is very strict s.t. with a lower residual reduction, the almost same accuracy could possibly be achieved. This guess is supported by the fact that the error produced by CG is the same as for the direct solvers. Increasing the residual norm criterion would very likely reduce the number of iterations significantly. This would make the Ginkgo CG implementation faster. Still, both QR and CG are very fast on GPUs and very competitive against the PETSc implementation which highly benefits from the huge number of ranks.

***Coarse* Series: Sparse RBF + Preconditioner** Since iterative solvers work well in this case, we now compare CG against GMRES and additionally enable the Jacobi and Cholesky preconditioning algorithms. Figure 4.10 plots the number of iterations needed for each combination for all meshes of the *coarse* series and the runtime of `mapConsistent()`. It can be seen that CG requires fewer iterations than GMRES. None of the three GMRES test cases are able to hit the residual norm criterion before reaching the maximum amount of 10,000 iterations for all mesh sizes up to $h = 0.009$. The preconditioners only begin to have an impact for the finest one respectively two meshes of the *coarse* series. GMRES combined with the Cholesky preconditioner requires much fewer iterations for $h = 0.006$ and $h = 0.004$ with the latter being in the same range as CG. Still, plain GMRES and Block-Jacobi preconditioned GMRES need 10,000 respectively about 6,000 iterations for calculating a solution for $h = 0.004$, whereas CG never requires more than 528 iterations for the finest mesh. CG in combination with Cholesky performs best since it iterates fewer times than every other solver in every test case. Interestingly, it takes more iterations for coarser systems to converge. From the perspective of runtime, the solvers mainly benefit from preconditioning if the mesh is rather fine. The Jacobi preconditioner has a positive impact mainly on CG. Due to its mathematical simplicity, the Jacobi preconditioner only creates a minor overhead. The Cholesky preconditioner has an even higher positive impact on the runtime, but only for $h = 0.004$ in the case of CG and $h = 0.006$ and $h = 0.004$ in the case of GMRES, i.e., for the finest mesh of the *coarse* series. Still, the difference of only a few hundred milliseconds between preconditioned and non-preconditioned CG is rather low. To summarize, CG in combination with Cholesky seems to be a good choice in the case of sparse matrices produced by local RBFs with a small support radius as there could be cases for which this combination provides larger speedups.

***Coarse* Series: Dense RBF** Here we compare the different solution strategies by using the compact thin plate spline (C-TPS) C2 function. Its definition is similar to the C6 RBF above with p again being the product between the input radius and the inverse of the support radius:

$$f(p) = \begin{cases} 1 - 30 \cdot p^2 - 10 \cdot p^3 + 45 \cdot p^4 - 6 \cdot p^5 - p^3 \cdot 60 \cdot \log(p); & \text{if } p < 1 \\ 0 & \text{else} \end{cases} \tag{4.5}$$

We look at the following solvers:

- CUDA CG
- CUDA GMRES
- CUDA QR
- Sequential Eigen Cholesky Decomposition
- PETSc GMRES with 48 + 48 Ranks

The support radius is set to $20h$ and therefore much larger than in the experiment before. This results in the system matrix being denser and worse conditioned. Therefore, it can already be expected that iterative solvers perform worse compared to direct solvers.

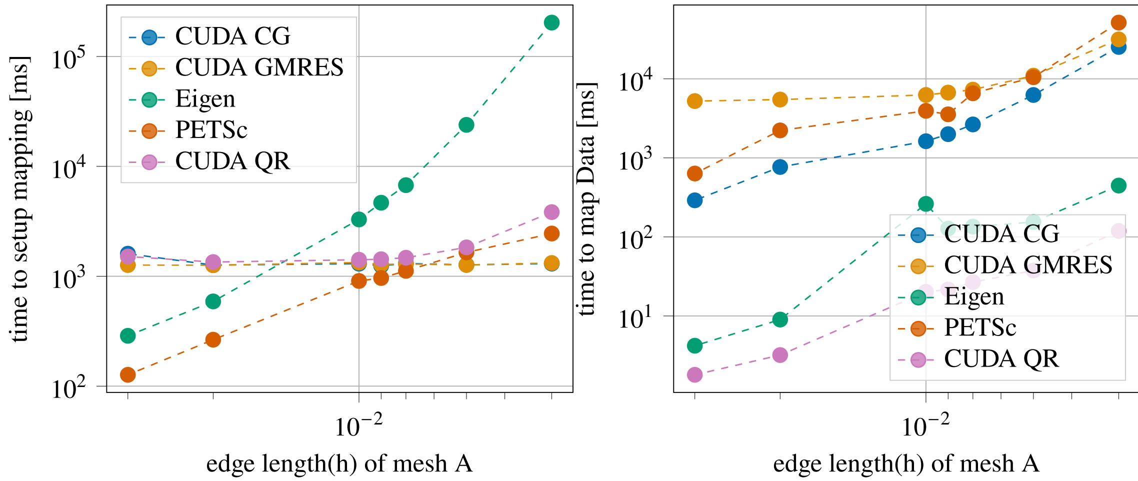

Figure 4.11: Results for C-TPS C2. The left plot shows the initial setup phase duration in ms and the right side the recurring mapping time.

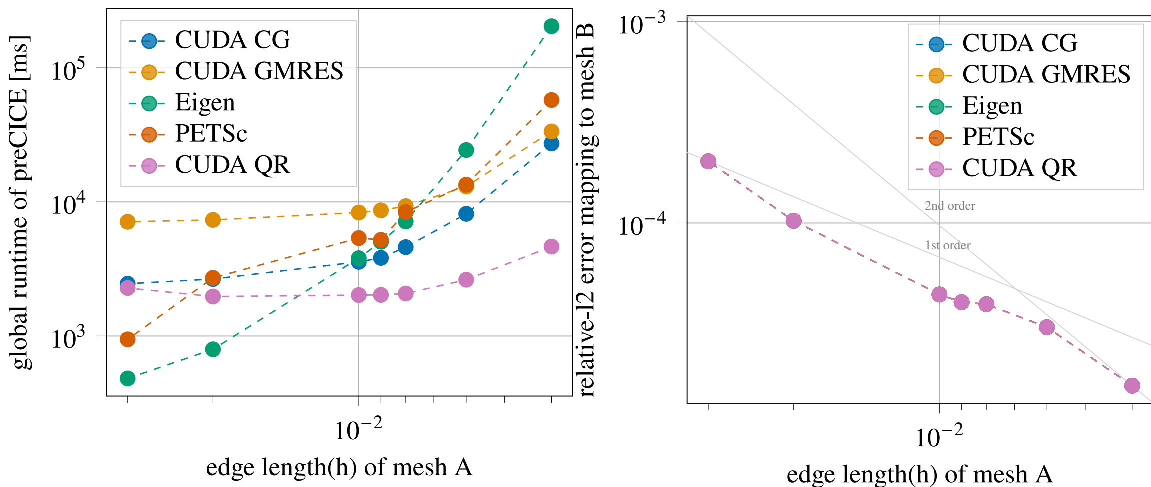

Figure 4.12: Results for C-TPS C2. The left plot shows the overall runtime of preCICE in ms and the right plot the average discrete mapping error.

Looking at the plots of Figure 4.11 and Figure 4.12, it can be seen that all mapping errors are almost the same and our CUDA QR implementation is the fastest solver out of all tested solvers. Compared to the iterative CUDA solver, it is more than five times faster for $h = 0.006$. Not only is the initial setup phase almost as fast as for the iterative GPU solvers, its recurring mapping time is the lowest among all. This is explained by the ill-conditioned system matrix which hurts the iterative solvers a lot. They need more iterations for convergence than in the sparse C6 RBF case. PETSc is more than ten times slower than the CUDA implementation and the Eigen solver requires more than 100 s. The overhead produced by CUDA/PETSc MPI distribution does only play a major role for the two coarsest meshes. This is where Eigen outperforms the other solvers.

CG and GMRES on the Nvidia A100 need more time than the QR solver, but they are still the second fastest variant outperforming PETSc and Eigen in the case of finer mesh sizes. Both behave similarly; however, CG is a bit faster.

The initial setup phase of the PETSc takes much more time, almost 50 s for $h = 0.004$. Solving the system is also one of the slowest processes. It can be concluded that PETSc loses its advantage seen above in the case of more dense matrices and worse condition numbers.

The sequential Eigen implementation takes the longest for all mesh sizes smaller than or equal to $h = 0.006$. The CUDA QR decomposition is faster already for $h = 0.01$ and produces the same error. Hence, it is almost always preferable to use the QR decomposition on accelerator cards over the Eigen Cholesky factorization.

We make use of this test case again in Appendix A in order to measure the performance of the LU and Cholesky decomposition methods using Ginkgo and CUDA.

Fine Series This test case examines our two approaches for treating VRAM limitations: our matrix-free implementation that assembles the matrix in every multiplication step from scratch and the CUDA unified memory model that automatically manages memory transfers and allows for over-allocation. We take the Gaussian RBF and set the support radius to $7h$ s.t. preCICE itself calculates a shape parameter that fits the selected support radius. The Gaussian RBF with shape parameter ζ reads as follows:

$$f(||\overrightarrow{x}||) = e^{-(\zeta \cdot ||\overrightarrow{x}||)^2} \tag{4.6}$$

This creates a rather sparse Φ. The GPU uses the Ginkgo CG iterative solver. We compare it against the PETSc distributed GMRES. We reduce the maximum number of iterations for these cases as the runtime increases significantly. These are the maximum number of iterations for each mesh of the *fine* series:

- $h = 0.004$: 500 iterations
- $h = 0.003/0.002/0.0014$: 200 iterations
- $h = 0.001$: 100 iterations
- $h = 0.0007/0.0005$: 50 iterations

Moreover, we reduce the residual criterion to 10^{-4}.

We start with a theoretical analysis of the matrix-free kernel which combines the assembly of Φ and matrix-vector multiplication. Figure 4.13 depicts the roofline diagram (generated by Nsight Compute using the Nvidia RTX 2070 Super) of the matrix-free kernel using input mesh $h = 0.002$. The arithmetic intensity is 33.82 FLOP/Byte, and the achieved performance is $0.78 * 10^{11}$ FLOP/s. Its computational intensity is about ten times higher than that of the assembly kernel. Looking at the green dot, we can see that it almost touches the horizontal bound for FP64 calculations, hence it again is compute-bound. The Nsight Compute profiler reports that the FP64 pipeline is overutilized, which, according to the profiler, might have an impact on the compute performance. Treating the issue of an overutilized compute pipeline is almost impossible since we repeatedly have to calculate $\mathcal{O}(K^2)$ matrix entries and perform matrix-vector multiplications.

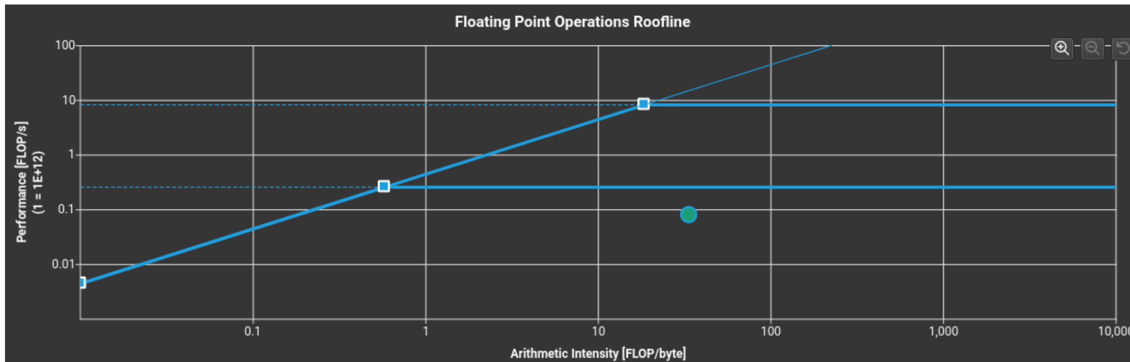

Figure 4.13: The roofline diagram of our matrix-free kernel which combines matrix assembly and matrix-vector multiplication. This kernel achieves a higher computational intensity than the assembly kernel and is also compute-bound.

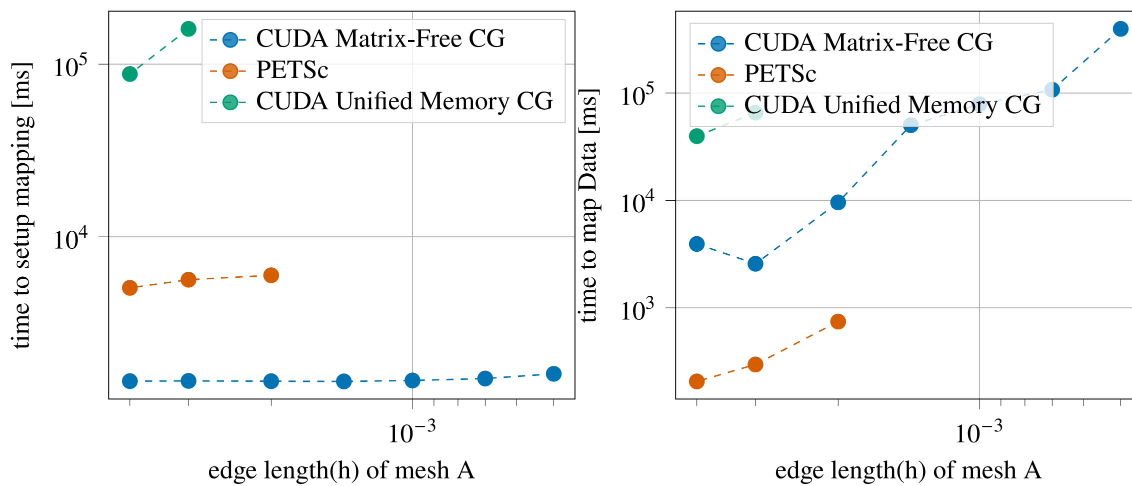

Figure 4.14: The initial setup phase time is shown on the left side and the recurring mapping time on the right. Whereas PETSc requires the least time for the three coarsest meshes of the *fine* series, only our matrix-free implementation is able to calculate a solution in a feasible amount of time.

Next, we look at the runtime measurement results produced by the three different implementations, which are shown in Figure 4.14 and Figure 4.15. All three approaches yield mapping errors in the same range. The CUDA unified memory approach can calculate the mapping for mesh sizes up to $h = 0.003$, whereas PETSc can do it up to $h = 0.002$. Afterward, both implementations do not finish in an acceptable amount of time. Moreover, using CUDA unified memory, matrix assembly, and system solving add up to over 100 s runtime, which is approximately ten times more than the other two approaches require. In contrast to CUDA unified memory and PETSc, our matrix-free approach handles all mesh sizes well. Even for the finest mesh size $h = 0.0005$, which contains more than one million vertices, the matrix-free implementation does calculate the mapping in about 400 s, which is a long time on one hand. On the other hand, it is the only approach in our evaluation that is capable of handling these large mapping problems. Furthermore, due to the fact that Φ is

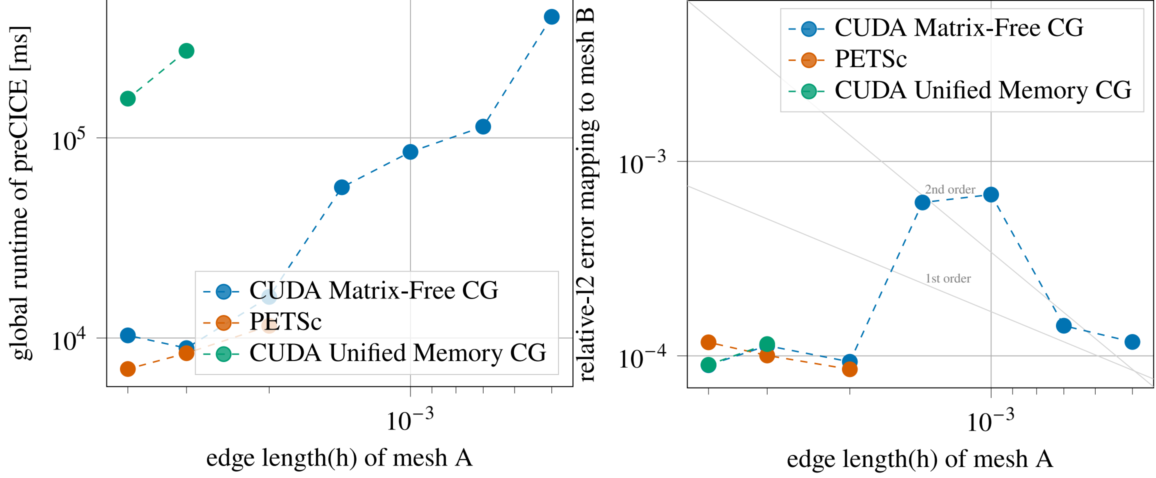

Figure 4.15: The left plot shows the global runtime of preCICE, the right one the mapping error. The mapping error is almost equal between all three approaches.

assembled whenever it is used in multiplication, there are almost no VRAM limitations anymore that must be considered. The matrix-free method only requires up to a few Gigabytes of VRAM to store intermediate results.

These observations lead to the conclusion that it is almost always favorable to use the matrix-free GPU approach, except if there is no accelerator card available or the mesh width h is still in a rather intermediate range. CUDA unified memory did not outperform OpenMP during matrix assembly (cf. Sec. 4.2.1) and performs worse in the entire RBF mapping process compared to the PETSc and matrix-free implementations. It therefore seems to be no viable option at all for most cases.

4.3 Continuous Coupling in 2D Heat Simulation

On top of the test cases described above, we use a time-dependent 2D heat solver where both solvers $\mathcal{S}_1$ and $\mathcal{S}_2$, which act on a separate domain each ($\Omega_1 = [0, 1] \times [0, 1]$ and $\Omega_2 = [1, 2] \times [0, 1]$), run on a GPU. The mathematical problem is a time-dependent 2D heat equation, which is explained in detail by Chourdakis et al. (2023). Its PDE reads as follows:

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \quad \text{in } \Omega \times (0, T] \tag{4.7}$$

$$u = u_D \quad \text{on } \Gamma_D \tag{4.8}$$

$$u = u_0 \quad \text{at } t = 0 \tag{4.9}$$

u is the function of unknowns that depends on the spatial directions x and y and time t. T is the end point in time of the simulation, Ω refers to the domain, Γ_D to the boundary of the domain and f is the source term.

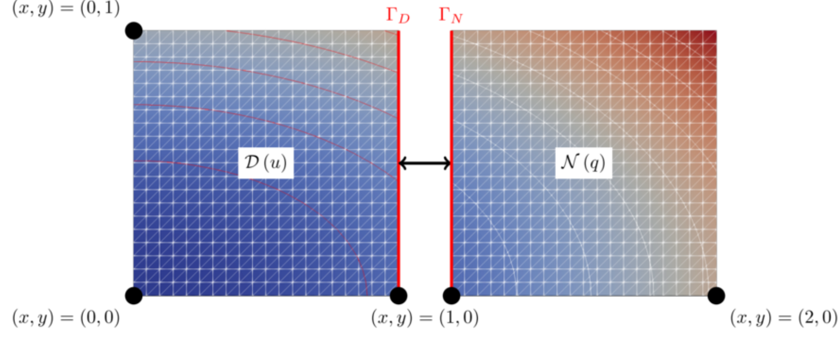

Figure 4.16: The two coupled domains of our two heat solvers. The coupling is carried out along the common coupling interface, depicted by the red line. (Chourdakis et al., 2022)

An analytical solution of the PDE is provided by Langtangen and Logg (2016, Sec. 3.1):

$$u(x, y, t) = 1 + x^2 + \alpha \cdot y^2 + \beta \cdot t \quad (4.10)$$

Given this function u, we can derive multiple analytical solutions of the PDE. The initial value function u_0 can be obtained by setting $t = 0$:

$$u_0 = u(x, y, t = 0) = 1 + x^2 + \alpha \cdot y^2 \quad (4.11)$$

The right-hand side function f is the result of plugging u into (4.7):

$$f(x, y, t) = \beta - 2 - 2 \cdot \alpha \quad (4.12)$$

We set $\alpha = 3$ and $\beta = 1.3$, which are the same values that are used by Chourdakis et al. (2023).

Both solvers ($\mathcal{S}_1$and $\mathcal{S}_2$) are coupled at the coupling interface $\Gamma_C = \Omega_1 \cap \Omega_2$ using preCICE. The left domain treats Γ_C as Dirichlet boundary conditions, i.e., it reads temperature values provided by $\mathcal{S}_2$. For the right domain, Γ_C acts as Neumann boundary, i.e., it retrieves derivative values provided by $\mathcal{S}_1$, which is the current heat flux at the boundary. The coupling of both domains is shown in Figure 4.16.

Both solvers use the CUDA support in the deal.II library (Arndt et al., 2022), i.e., the PDE problem is solved using CG on Nvidia GPUs. However, there is still an issue with this setup. Since our goal is stress testing of the simulation up to a point where preCICE on a CPU is the bottleneck, the setup provided by Chourdakis et al. (2022) is not suited as the computational intensity of the mapping of interface nodes is too low. As a result, we create a larger interface that emulates a volumetric coupling process. We therefore make use of a modified version where the overlapping percentage of interface nodes can be set programmatically. The more nodes are part of the interface, the larger our RBF interpolation system becomes.

Recurring, time-dependent simulations are possible via the `advance()` API function in preCICE. The term "advance" refers to advancing the coupling, i.e., communicating all necessary data and steering the time-stepping behavior. Hence, `mapConsistent()`, which happens repeatedly as described above, is called whenever one solver makes the call to `advance()`. Interpolated values are then distributed as configured.

This simulation test case allows us to test and measure the coupling setup shown in Figure 4.17. Both solvers run their simulation part on GPUs, whereas preCICE is configured to run on CPU and GPU depending on the test case we want to investigate. We do not evaluate our matrix-free approach here since meaningful results are provided in the previous section. Additionally, we set up a third variant in which both solvers run on the CPU and preCICE on the GPU. Having it this way allows us to measure all GPU memory operations that take place within preCICE during a coupled simulation.

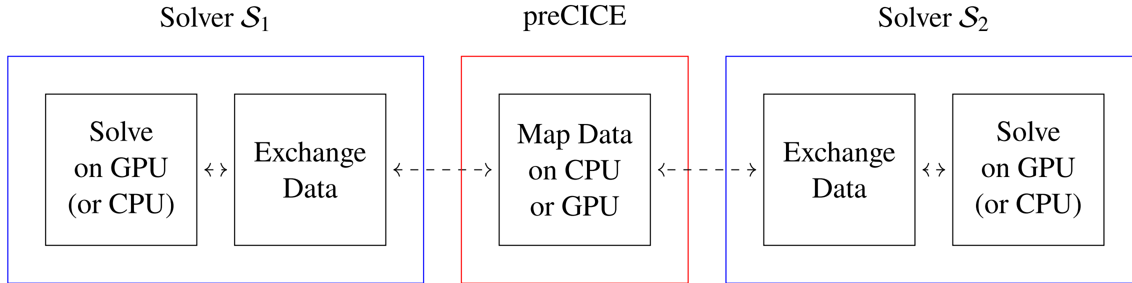

Figure 4.17: Coupling setup in our coupled 2D heat equation simulation.

There are multiple relevant questions that can be answered with this setup:

- How efficient is a coupling approach where both solvers run on GPU and preCICE is restricted to execute the mapping on the CPU (**GPU-CPU-GPU**), i.e., how is the performance portability between CPU and GPU?
- How expensive in terms of runtime are memory transfers from device to host and vice versa?
- Can we benefit from preCICE performing the RBF mapping on a GPU when it becomes the bottleneck on the CPU (**GPU-GPU-GPU**) in terms of runtime?
- How expensive are memory transfers between host and device and do they impose a major bottleneck?

To achieve a suitable stress-testing setup, we make the following configurations for two mapping cases, one that produces a dense RBF system matrix Φ and one that yields a sparse Φ:

- We set a time stepping size of 0.1 for $T = 1$, i.e., ten time steps are simulated.
- The percentage of overlapping interface nodes is set to 65%. The Neumann solver provides 24,000 interface nodes and the Dirichlet side 5,952.
- The coupling scheme is set to *serial*, i.e., only one of the two solvers is allowed to run at any time. This results in an alternating scheme with $\mathcal{S}_1$ and $\mathcal{S}_2$ running in an alternating fashion.
- The dense case uses mapping with the compact polynomial C6 RBF and a support radius of 0.8, which is 80% of the size of domain Ω_1 or Ω_2 into x and y direction.
- The sparse case has a support radius set to 0.02 using the C6 RBF, which creates a rather sparse Φ.

In order to measure the influence of GPU memory transfers in preCICE, we use the Nvidia nvprof[3] profiler tool. It measures every call to the CUDA runtime API and every kernel launch, i.e., it covers a broader range of memory operations than our own timing measurement within preCICE. Since we are interested in memory bottlenecks, we report the runtime measurements of three CUDA API functions that are responsible for memory management (including variants of them): `cudaMalloc()`, which dynamically allocates memory on the GPU, `cudaFree()`, which releases dynamically allocated memory, and `cudaMemcpy()`, which transfers data between host and device memory. This should give a clear picture of the influence of memory allocations and especially transfers on the runtime. The profiling induces minor overhead on the global runtime of the simulation, up to a few seconds. We consider this to be negligible for a rather qualitative analysis in which we want to investigate the impact of GPU memory transfers. Both solvers are started at the same time automatically in order to avoid biased initialization times.

Dense Test Case We start with the **GPU-CPU-GPU** setting. It uses the Eigen implementation, which applies the Cholesky decomposition in order to factorize Φ, to compute the mapping. The runtime measurements of both solvers are listed in Table 4.2. It also contains the cumulative runtime of GPU memory transfers within the two deal.II solvers. These transfers are necessary as the deal.II preCICE adapter runs on the CPU as well. The deal.II solver uses the GPU for solving its part of the heat equation, any other work is done on the host side instead.

Initializing preCICE needs by far the most time with over four minutes. It includes the call to `computeMapping()`, i.e., factorizing Φ is included in this measurement. Since our RBF interpolation problem is very large, factorizing Φ poses a heavy workload for the sequential Eigen implementation. Since both solvers wait for each other, their initialization time is very similar. Mapping the data repetitively is much faster since there is now only the task of solving a triangular system via backward substitution. This is reflected by the “advance preCICE” entry Table 4.2. Dividing 18 s respectively 9 s by ten timesteps, this results in 2 s respectively 1 s per data mapping execution, which is negligible compared to the initialization time of preCICE. We can draw the conclusion that factorizing Φ is the clear bottleneck for this setup.

As we mentioned above, both solvers compute the solution of the PDE on GPUs, i.e., data must be transferred to and from the VRAM in a recurring fashion, which also induces minor overhead. $\mathcal{S}_1$ spends about 14 ms on transferring data between host and device, $\mathcal{S}_2$ about 30 ms. This is also insignificant in relation to almost 280 s that are spent on factorizing Φ.

We conclude that running both solvers on GPU and preCICE data mapping on CPU does not impose high memory conflicts if the RBF mapping problem is rather large and both domains produce computationally intense linear systems of equations. Only if the overall mathematical problem was very small, memory transfers could play an important role. However, we already concluded in the last section that the usage of GPUs makes sense if and only if Φ is not too small.

[3] https://docs.nvidia.com/cuda/profiler-users-guide/

Section	Solver $\mathcal{S}_1$	Solver $\mathcal{S}_2$
advance preCICE	**18.96 s**	**9.446 s**
assemble rhs	0.026 s	0.082 s
compute errors	0.002 s	0.005 s
initialize preCICE	**273.5 s**	**278.2 s**
output	0.174 s	0.320 s
solve system	1.058 s	5.548 s
deal.II GPU to CPU	0.003 s	0.012 s
deal.II CPU to GPU	0.011 s	0.018 s

Table 4.2: Runtime results of the coupled heat solver using Eigen for calculating the RBF mapping. Especially the setup phase takes a very long time.

Next, we analyze the **GPU-GPU-GPU** setup by having both solvers and preCICE making use of GPUs. Here, we set the CUDA QR decomposition as the desired solver for our RBF interpolation problem since Φ is a rather sparse matrix. Therefore, the workload is fully comparable to the Eigen version since `computeMapping()` also factorizes a matrix and `mapConsistent()` solves a triangular system.

Table 4.3 lists the runtime measurements. On one hand, it becomes clear that initializing preCICE, i.e., factorizing Φ, is much faster on the GPU than on the CPU, which is expected given the results of the turbine mapping experiments in Sec. 4.2.2. It is about 46 times faster than the Eigen variant. This is a significant speedup due to the extended hardware access of preCICE, enabling the simulation to finish much earlier while keeping the same precision. On the other hand, recurring data mapping is only slightly faster than the Eigen implementation by approximately 1 s to 4 s, which leaves the question of whether there is some hidden speedup potential that can be exploited. Looking at the CUDA API memory routines, the call to `cudaFree()` imposes a heavy impact on the global runtime. Being able to reduce the number of calls to `cudaFree()` could potentially speed up the GPU-accelerated RBF interpolation even more. However, we have no influence on what happens within cuSolver and have to rely on the fact that cuSolver is already highly optimized by Nvidia. Still, memory optimizations have the potential to be a good task for follow-up work to gain even higher speedups since they seem to have a huge impact.

Finally, we look at the **CPU-GPU-CPU** case in which both heat solvers run on the CPU and preCICE calculates the mapping on the GPU using the QR decomposition. Having it this way allows us to verify that most of the memory management overhead happens within preCICE. Table 4.4 lists the runtime measurements of this setup. Most of the time is spent on solving the PDE equations within the CPU solvers. Still, about 12 s are consumed by `cudaFree()` in $\mathcal{S}_1$, which leads to the conclusion that our implementation in preCICE creates some overhead, however, it is negligible compared to potential bottlenecks that arise when using CPUs.

Sparse Test Case We also test how the iterative solvers work in recurring coupling scenarios. GMRES is used as solver algorithm. As already mentioned, the iterative solvers do not have a setup phase that requires Φ to be factorized, but instead, they take an approximate solution of

Section	Solver $\mathcal{S}_1$	Solver $\mathcal{S}_2$
advance preCICE	**14.79 s**	**10.19 s**
assemble rhs	0.022 s	0.065 s
compute errors	0.002 s	0.005 s
initialize preCICE	**4.623 s**	**5.937 s**
output	0.185 s	0.328 s
solve system	1.517 s	4.492 s
CUDA mem. alloc	0.114 s	0.226 s
CUDA mem. copy	0.883 s	2.774 s
CUDA mem. free	9.117 s	2.717 s

Table 4.3: Runtime results of the coupled heat solver using the CUDA QR decomposition for calculating the RBF mapping. Initializing preCICE, i.e., factorizing the system matrix, is much faster than using Eigen.

Section	Solver $\mathcal{S}_1$	Solver $\mathcal{S}_2$
advance preCICE	**720.4 s**	**93.84 s**
assemble rhs	1.363 s	5.099 s
compute errors	0.063 s	0.238 s
initialize preCICE	**8.321 s**	**14.51 s**
output	0.778 s	2.49 s
solve system	82.66 s	697.4 s
CUDA mem. alloc	0.151 s	0.258 s
CUDA mem. copy	0.004 s	0.013 s
CUDA mem. free	12.444 s	5.049 s

Table 4.4: Runtime results of the coupled heat solver using the CUDA QR decomposition in preCICE and running both solvers on the CPU. Especially freeing VRAM consumes a lot of time. However, most time is spent on solving PDEs on the CPU.

λ and try to iteratively converge against the true solution. The `computeMapping()` method only assembles the matrix in our matrix-based setup. Therefore, it is expected that the initialization phase of preCICE is shorter than in the case of the CUDA QR decomposition. However, advancing the simulation requires GMRES to restart iterating every time, given a new right-hand side f of the system. In this setup, we observe that the iterative solver does work well. However, repeatedly mapping the data takes about 9 s longer than in the QR variant, even though we created a sparser mapping that iterative solvers should benefit from. This indicates that in a recurring mapping scenario, precomputed factorizations of Φ might be favorable unless it is a mostly sparse matrix. Looking at the memory transfer times reveals that `cudaMemcpy()` is now the major factor in GPU memory management as opposed to `cudaFree()` in the QR implementation. This suggests that these calls mainly happen within the corresponding libraries, Ginkgo and cuSolver.

Section	Solver $\mathcal{S}_1$	Solver $\mathcal{S}_2$
advance preCICE	**26.99 s**	**22.03 s**
assemble rhs	0.024 s	0.10 s
compute errors	0.002 s	0.005 s
initialize preCICE	**1.55 s**	**2.60 s**
output	0.1867 s	0.36 s
solve system	1.51 s	4.75 s
CUDA mem. alloc	0.12 s	0.25 s
CUDA mem. copy	16.23 s	2.99 s
CUDA mem. free	1.71 s	1.69 s

Table 4.5: Runtime results of the coupled heat solver using GMRES on a GPU for calculating the RBF mapping. Since there is no matrix factorization involved, setup phase is faster than before, the recurring mapping computation takes more time.

4.4 Discussion

The last two sections evaluated the performance and efficiency of our GPU-based RBF mapping implementation. We now discuss different aspects of our work that address technical points as well as usability of preCICE itself, given that accelerator cards have proven to be efficiently applicable in preCICE.

First and foremost, we want to emphasize that we do not only allow for significant speedups in RBF data mapping with our approach, but we also fill a gap in preCICE by implementing support for accelerator cards. Whereas the other implementations, which rely on Eigen and PETSc, target single-core respectively multi-node environments, our work addresses multi-core CPUs and data-parallel accelerator cards. Nowadays, it is very common to have compute clusters that hold many separate high-end GPUs. It is often easier to reserve a single GPU in scheduler-managed clusters than reserving many CPU cores or compute nodes, which simplifies the usage of preCICE data mapping methods in distributed computing environments. On top of that, the GPU-based solver can run on consumer hardware as long as CUDA or HIP is supported. Therefore, it can also be used on personal computers or even laptops that have an additional GPU installed, making our work accessible to a very broad range of users.

In Sec. 3.3, we described the configuration options for our GPU-based solvers. The last two sections have shown that configuring the solvers (including preconditioners) and, in the case of iterative solvers, the maximum number of iterations and residual criterion does have a major impact on the performance of the RBF mapping. For instance, we set the maximum number of iterations to $10,000$ for all meshes of the *coarse* series. Iterative solvers could finish much earlier if this number was reduced in the case of the iterative solvers converging much earlier to the same solution, which makes them “waste” time on iterating without any improvement. However, finding the correct configuration highly depends on the specific mapping problem, its size, the RBF that is used, and more. Hence, we cannot formulate a golden rule beforehand. It is up to the users of preCICE to find the best criteria and solvers tailored to their problem. In Chapter 5, we provide a decision-making diagram that ought to help the users to find the correct configuration for RBF data mapping.

In terms of runtime, it is favorable to use the matrix-based approach whenever possible in order to have Φ and A_M precomputed. However, if the required memory space of Φ exceeds the available VRAM, it is always the best option to go for the matrix-free implementation instead of the CUDA unified memory or PETSc-based implementations in preCICE. Matrix factorization methods such as QR require a matrix to be explicitly stored, hence we cannot offer such algorithms as matrix-free variants. As a result, preCICE-coupled simulations using the matrix-free approach need to work with iterative solvers. Since every iteration is computationally much more expensive when Φ has to be assembled every time it is used, the maximum amount of iterations might be reduced if the accelerator card cannot offer enough computational power to execute a high number of maximum iterations. If that is the case, the accuracy of the RBF interpolation might potentially suffer. However, we have shown that even a few hundred or even fewer iterations can already produce acceptable results with a mapping error being in a feasible range. Therefore, our matrix-free approach does not replace the matrix-based counterparts. Instead, it complements them by offering the users of preCICE a possibility to solve even larger RBF interpolation problems. Since the matrix-free kernel only needs a few Gigabytes of VRAM for the largest turbine meshes, it allows to scale up to extreme problem sizes if a high-end GPU is available. But it also allows users with consumer cards to work with really large mapping problems as they often lack VRAM but do offer a high computational power. Hence, the matrix-free approach is also of interest to users that do not have access to the high-end compute GPUs that are mainly available in compute clusters.

Finally, we also want to mention further approaches for solving the RBF system that we tested but did not work well. We discussed algebraic multi-grid (AMG) in Sec. 2.4.1 as one possible preconditioning technique. It is available as a preconditioner as well as a standalone solver in Ginkgo. We evaluated the performance of the AMG preconditioner in combination with CG for smaller turbine grids. Whereas the number of iterations decreased for some cases by a significant factor, one iteration of the AMG-preconditioned CG takes about 100 times as long as the non-preconditioned counterpart. This observation coincides with experiences made by other users of Ginkgo (cf. the Github discussion on AMG[4]). Hence, using the AMG preconditioner in its current state is not a feasible option; however, with improvements in the future, it might become one. The ILU preconditioner (as mentioned in Sec. 2.4.1) did not lead to results in a feasible amount of time. We therefore consider it to be unsuited for our RBF mapping problem.

[4]https://github.com/ginkgo-project/ginkgo/discussions/1111 (last accessed on April 12, 2023)

5 Conclusion and Outlook

5.1 Summary and Conclusion

This thesis investigated the efficient usage of accelerator cards in the coupling library preCICE. We looked at RBF data mapping in preCICE and parallelization strategies to speed up the mapping process.

We looked at two scenarios that provide valuable insights into RBF data mapping on GPUs. The turbine grid mapping cases from ASTE allow us to investigate and judge the behavior and performance of different solvers and data-parallel backends when it comes to solving a mostly ill-conditioned RBF interpolation problem (cf. Sec. 4.2). The coupled heat solver reflects a real simulation use case of preCICE with multiple timesteps being simulated (cf. Sec. 4.3). This case makes it possible to measure the influence of memory transfers from host to device and vice versa. It also provides insights into the behavior of the different solvers in recurring mapping scenarios.

Looking at the results of the turbine grid mappings, it becomes clear that the usage of GPUs leads to potential improvements in all scenarios. The first time-consuming component of RBF mapping is the assembly of the matrices Φ and A_M. Modeling the assembly in a sequential fashion typically results in two nested `for`-loops. We approached this by using a 2D thread grid on the GPU, which results in a speedup of a factor up to $1,000$. We also provide a data-parallel OpenMP implementation that is only slightly slower. Afterward, we looked into solving the system $f = \Phi \cdot \lambda$ efficiently on GPUs. Looking at both, a rather sparse Φ and rather dense Φ, it became clear that iterative solvers such as CG and GMRES perform well on sparse system matrices, whereas direct decompositions such as QR are the better choice for denser matrices. The matrix-free implementation is a viable alternative that should be used whenever matrix-based methods reach VRAM limitations. We have shown that this approach is able to obtain good solutions for very large RBF interpolation problems in a feasible amount of time.

Based on these observations, we provide a decision-making strategy in Figure 5.1. Its purpose is to guide users of preCICE when it comes to configuring RBF data mapping. The best configuration highly depends on available hardware, the RBF itself, and the size of the problem. In summary, using GPUs is almost always the preferred strategy unless there are no accelerator cards available, the problem size is too small or the GPU is outperformed by a highly parallelized PETSc configuration, which, however, requires a huge amount of CPU cores or compute nodes.

The coupled 2D heat solver reflects a more realistic use case scenario of preCICE. We tested different setups for both coupled solvers and preCICE. We investigated the case in which both solvers run on a GPU and preCICE on a CPU (**GPU-CPU-GPU**). Having a very large coupling interface and hence a very large RBF mapping problem, we have seen that the bottleneck is clearly the CPU-powered mapping and not in memory transfers within both GPU-accelerated solvers. We verified this observation by investigating the **GPU-GPU-GPU** setup. Having all three components

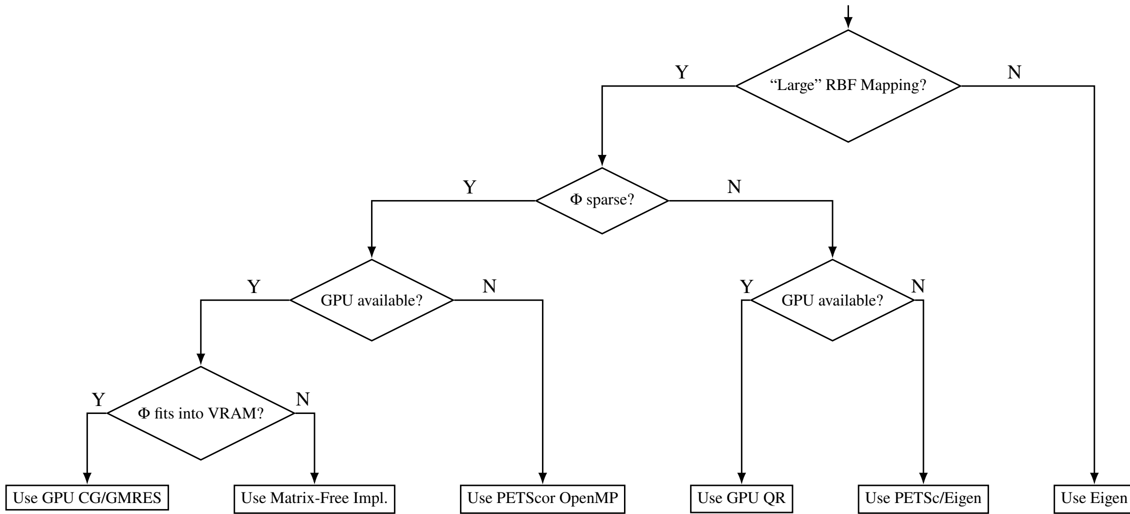

Figure 5.1: A decision strategy that helps to decide whether using a GPU for RBF interpolation is the best choice. The most important factors are problem size and sparsity of the system matrix Φ.

leveraging a GPU gave a speedup of a factor larger than 40. This clearly proves that using our GPU-accelerated version of preCICE does not only allow for larger problem sizes, but it also significantly accelerates entire simulations. This is of special interest if a simulation is repeatedly executed because every single run of the simulation can now be finished much faster. Testing the **CPU-GPU-CPU** case, i.e., both heat solvers run on CPU only, revealed that there is some memory management overhead within our preCICE implementation, however, they are insignificant compared to the speedup factors that we were able to achieve.

Our GPU-based implementation can easily be expanded in the future whenever new solver algorithms are introduced in Ginkgo. The newly implemented `GinkgoRadialBasisFctSolver` class allows for easy extension. On top of that, we added CUDA and HIP support to the preCICE CMake toolchain. Therefore, it is also possible to add new solver algorithms by using cuSolver for example. This makes our work interesting for potential future research, which we discuss in the next section.

To sum up the findings of this thesis, we have successfully shown that efficient application of accelerator cards in preCICE is possible and has the potential to speed up RBF data mapping by a significant factor. The GPU solver must be carefully configured w.r.t. the properties of the RBF interpolation problems, however, we provide an option for almost all cases that could possibly arise. We enabled preCICE to leverage the huge compute power that is provided by GPUs, which nowadays can be used in almost every compute cluster.

5.2 Outlook

There are many different directions for follow-up work. We want to discuss future research possibilities from both perspectives, the mathematical and the technical. This includes research questions that are out of the scope of this thesis.

First and foremost, there are more algorithms for solving a linear system of equations than presented in this thesis. For example, Ginkgo provides extensions of CG and GMRES such as "biconjugate gradient (BiCG)", "BiCG stabilized (BiCGSTAB)" and "compressed basis GMRES (CB-GMRES)". It could be interesting to investigate whether the RBF data mapping can benefit from these solver algorithms in some cases. As already mentioned in Sec. 4.4, we evaluated the AMG method provided by Ginkgo and concluded that it is not able to provide speedups in its current state. However, it is likely to be reworked in the future, i.e., it might become a very promising alternative to CG and GMRES and could be implemented into preCICE.

Next, there is the question of whether preCICE can efficiently make use of multi-GPU environments, i.e., multiple accelerator cards are available in one or many compute nodes. In its current state, our implementation can make use of more than one GPU if MPI is enabled for distributed-memory parallelism (comparable to the PETSc implementation). However, it does not dynamically distribute the workload by differentiating the different GPUs via their unique ID. If multiple MPI ranks are running on the same node, they will all access the same GPU, which results in having fewer computational resources compared to using multiple different cards. Investigating a more efficient application of multiple GPUs in a MPI setting has the potential to provide further speedups. A baseline to start with is to distribute the workload of all ranks equally across the number of accelerator cards, e.g., by taking the MPI rank modulo the amount of GPUs.

Another direction for potential future research is provided by Fasshauer and Zhang (2009). They discuss how ill-conditioned RBF interpolation systems can be preconditioned s.t. they are easier to solve in terms of runtime. The preconditioning approach is based on their own *approximate moving least-squares (AMLS)* approximation method. The system of linear equations $Ac = f$ is converted into $APc = f$. Note that this is not an equivalent reformulation since P is only used on the left side of the equation. Hence, this transformation corresponds to a basis change that ought to reduce the condition number of the interpolation matrix. P itself is created by calculating the sum of the following matrix product:

$$P^{(n)} = \sum_{k=0}^{n} (\mathbb{I} - A)^k \tag{5.1}$$

where A refers to the original RBF system matrix. The authors also provide a recursive definition of $P^{(n)}$ for their computationally efficient algorithm:

$$P^{(k)} = P^{(k-1)}(2\mathbb{I} - AP^{(k-1)}) \tag{5.2}$$

Clearly, there are many (sparse) matrix products. Referring back to Sec. 2.1, these are BLAS routines and can therefore be parallelized on accelerator cards. Moreover, RBF evaluations are also necessary for the whole procedure and have been discussed in this thesis. As a result, follow-up work can investigate the use of this preconditioning technique in preCICE by adapting our implementation. It should be easy to implement since this procedure could be added as a Ginkgo preconditioner and therefore be perfectly integrated into the solver flow. The authors conclude that this technique works reasonably well in terms of acceleration if $\kappa(A) < 10^{20}$ and $\kappa(AP^{(n)}) \ll \kappa(A)$.

Finally, our work does also support other research activities that are currently ongoing in preCICE. One particular topic is the “partition of unity (PU)” approach for data mapping. The idea is to split the global domain into M potentially overlapping partitions and associate a weight function to each subdomain s.t. for every vertex x (in the discrete case) the following holds:

$$\sum_{j}^{M} w_j(x) = 1 \tag{5.3}$$

Now that we have the weighted influence of each subdomain on each vertex, we can form a local approximation u_j for each subdomain. The global approximant results as a weighted sum of local approximants times their weight for vertex x:

$$\mathcal{P} = \sum_{j}^{M} u_j w_j(x) \tag{5.4}$$

This approach requires the solution of M interpolation problems if RBFs are used in order to get M approximations. The cuSolver library supports so-called “batched” factorizations on top of the default QR decomposition. Instead of factorizing one matrix into Q and R, it can do it for n matrices in parallel. However, the batched QR decomposition requires all M matrices to have the same sparsity pattern, which is an unlikely case in this approach. Still, GPUs can potentially speed up the solving process of M interpolations systems. This has the potential for huge speedups of the PU approach and can definitely be useful in the future in order to realize the PU principle in preCICE.

Bibliography

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. D. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen (1999). *LAPACK Users' Guide*. Third. Philadelphia, Pennsylvania, USA: SIAM (cit. on p. 23).

Anzt, H., T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, E. S. Quintana-Ortí (Feb. 2022). "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing". In: *ACM Transactions on Mathematical Software* 48.1, 2:1–2:33. ISSN: 0098-3500. DOI: 10.1145/3480935. URL: https://doi.org/10.1145/3480935 (cit. on p. 22).

Arndt, D., 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, D. Wells (2022). "The deal.II Library, Version 9.4". In: *Journal of Numerical Mathematics* 30.3, pp. 231–246. DOI: 10.1515/jnma-2022-0054. URL: https://dealii.org/deal94-preprint.pdf (cit. on p. 57).

Bolz, J., I. Farmer, E. Grinspun, P. Schröder (July 2003). "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid". In: *ACM Trans. Graph.* 22.3, pp. 917–924. ISSN: 0730-0301. DOI: 10.1145/882262.882364. URL: https://doi.org/10.1145/882262.882364 (cit. on p. 18).

Cheng, J. (2014). *Professional Cuda C programming*. en. Wrox programmer to programmer. Indianapolis, IN: John Wiley and Sons, Inc. ISBN: 978-1-118-73932-7 (cit. on p. 9).

Chourdakis, G., K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H. Bungartz, L. Cheung Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, O. Koseomur (2022). "preCICE v2: A sustainable and user-friendly coupling library". In: *Open Research Europe* 2.51. DOI: 10.12688/openreseurope.14445.2. URL: https://doi.org/10.12688/openreseurope.14445.2 (cit. on pp. 1, 2, 12, 42, 49, 57).

Chourdakis, G., D. Schneider, B. Uekermann (Feb. 2023). "OpenFOAM-preCICE: Coupling OpenFOAM with External Solvers for Multi-Physics Simulations". en. In: *OpenFOAM® Journal* 3, pp. 1–25. ISSN: 2753-8168. DOI: 10.51560/ofj.v3.88. URL: https://journal.openfoam.com/index.php/ofj/article/view/88 (cit. on pp. 56, 57).

Cuomo, S., A. Galletti, G. Giunta, A. Starace (2013). "Surface reconstruction from scattered point via RBF interpolation on GPU". In: *2013 Federated Conference on Computer Science and Information Systems*, pp. 433–440 (cit. on p. 19).

Dahmen, W., A. Reusken (2022). *Numerik für Ingenieure und Naturwissenschaftler: Methoden, Konzepte, Matlab-Demos, E-Learning*. de. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-662-65180-3 978-3-662-65181-0. DOI: 10.1007/978-3-662-65181-0. URL: https://link.springer.com/10.1007/978-3-662-65181-0 (cit. on p. 10).

Ding, Z., G. Mei, S. Cuomo, N. Xu, H. Tian (Oct. 2018). "Performance Evaluation of GPU-Accelerated Spatial Interpolation Using Radial Basis Functions for Building Explicit Surfaces". en. In: *International Journal of Parallel Programming* 46.5, pp. 963–991. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-017-0538-6. URL: http://link.springer.com/10.1007/s10766-017-0538-6 (cit. on p. 19).

Edwards, H. C., C. R. Trott, D. Sunderland (2014). “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257 (cit. on p. 22).

Fasshauer, G. E. (2007). *Meshfree Approximation Methods with Matlab*. WORLD SCIENTIFIC. DOI: 10.1142/6437. eprint: https://www.worldscientific.com/doi/pdf/10.1142/6437. URL: https://www.worldscientific.com/doi/abs/10.1142/6437 (cit. on p. 11).

Fasshauer, G. E., J. G. Zhang (2009). “Preconditioning of Radial Basis Function Interpolation Systems via Accelerated Iterated Approximate Moving Least Squares Approximation”. en. In: *Progress on Meshless Methods*. Ed. by A. J. M. Ferreira, E. J. Kansa, G. E. Fasshauer, V. M. A. Leitão. Dordrecht: Springer Netherlands, pp. 57–75. ISBN: 978-1-4020-8820-9 978-1-4020-8821-6. DOI: 10.1007/978-1-4020-8821-6_4. URL: http://link.springer.com/10.1007/978-1-4020-8821-6_4 (cit. on p. 67).

Flynn, M. J. (Sept. 1972). “Some Computer Organizations and Their Effectiveness”. en. In: *IEEE Transactions on Computers* C-21.9, pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071. URL: http://ieeexplore.ieee.org/document/5009071/ (cit. on p. 5).

Heidrich, W. (Jan. 2005). “Computing the Barycentric Coordinates of a Projected Point”. In: *J. Graphics Tools* 10, pp. 9–12. DOI: 10.1080/2151237X.2005.10129200 (cit. on p. 39).

Householder, A. S. (Oct. 1958). “Unitary Triangularization of a Nonsymmetric Matrix”. en. In: *Journal of the ACM* 5.4, pp. 339–342. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/320941.320947. URL: https://dl.acm.org/doi/10.1145/320941.320947 (cit. on pp. 13, 14).

Idais, H., M. Yasin, M. Pasadas, P. González (Oct. 2019). “Optimal knots allocation in the cubic and bicubic spline interpolation problems”. en. In: *Mathematics and Computers in Simulation* 164, pp. 131–145. ISSN: 03784754. DOI: 10.1016/j.matcom.2018.11.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0378475418302957 (cit. on p. 43).

Kandekar, C., A. Ravikumar, D. Höche, W. E. Weber (Mar. 2023). “A partitioned computational framework for damage evolution in stress corrosion cracking utilizing phase-field”. en. In: *PAMM* 22.1. ISSN: 1617-7061, 1617-7061. DOI: 10.1002/pamm.202200211. URL: https://onlinelibrary.wiley.com/doi/10.1002/pamm.202200211 (cit. on p. 2).

Langtangen, H. P., A. Logg (2016). *Solving PDEs in Python*. en. Cham: Springer International Publishing. ISBN: 978-3-319-52461-0 978-3-319-52462-7. DOI: 10.1007/978-3-319-52462-7. URL: http://link.springer.com/10.1007/978-3-319-52462-7 (cit. on p. 57).

Lawson, C. L., R. J. Hanson, D. R. Kincaid, F. T. Krogh (Sept. 1979). “Basic Linear Algebra Subprograms for Fortran Usage”. en. In: *ACM Transactions on Mathematical Software* 5.3, pp. 308–323. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/355841.355847. URL: https://dl.acm.org/doi/10.1145/355841.355847 (cit. on p. 6).

Lewis, J. P., F. Pighin, K. Anjyo (2010). “Scattered data interpolation and approximation for computer graphics”. en. In: *ACM SIGGRAPH ASIA 2010 Courses on - SA ’10*. Seoul, Republic of Korea: ACM Press, pp. 1–73. ISBN: 978-1-4503-0527-3. DOI: 10.1145/1900520.1900522. URL: http://portal.acm.org/citation.cfm?doid=1900520.1900522 (cit. on p. 11).

Lindholm, E., J. Nickolls, S. Oberman, J. Montrym (Mar. 2008). “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. en. In: *IEEE Micro* 28.2, pp. 39–55. ISSN: 0272-1732. DOI: 10.1109/MM.2008.31. URL: http://ieeexplore.ieee.org/document/4523358/ (cit. on p. 5).

Lindner, F., M. Mehl, B. Uekermann (May 2017). *Radial Basis Function Interpolation for Black-Box Multi-Physics Simulations* (cit. on p. 11).

Ljungkvist, K. (2014). “Matrix-Free Finite-Element Operator Application on Graphics Processing Units”. en. In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, M. Alexander. Vol. 8806. Series Title: Lecture Notes in Computer Science. Springer International Publishing, pp. 450–461. ISBN: 978-3-319-14312-5 978-3-319-14313-2. DOI: 10.1007/978-3-319-14313-2_38. URL: http://link.springer.com/10.1007/978-3-319-14313-2_38 (cit. on p. 35).

Meister, A. (2015). *Numerik linearer Gleichungssysteme*. de. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-07199-8 978-3-658-07200-1. DOI: 10.1007/978-3-658-07200-1. URL: http://link.springer.com/10.1007/978-3-658-07200-1 (cit. on pp. 13–17).

Naseri, A., A. Totounferoush, I. González, M. Mehl, C. D. Pérez-Segarra (Aug. 2020). “A scalable framework for the partitioned solution of fluid–structure interaction problems”. en. In: *Computational Mechanics* 66.2, pp. 471–489. ISSN: 0178-7675, 1432-0924. DOI: 10.1007/s00466-020-01860-y. URL: https://link.springer.com/10.1007/s00466-020-01860-y (cit. on p. 2).

Naumov, M., M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, R. Strzodka (Jan. 2015). “AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods”. en. In: *SIAM Journal on Scientific Computing* 37.5, S602–S626. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/140980260. URL: http://epubs.siam.org/doi/10.1137/140980260 (cit. on p. 18).

Nvidia Corporation (Nov. 2006). *Technical Brief - NVIDIA GeForce 8800 GPU Architecture Overview* (cit. on p. 8).

OpenMP Architecture Review Board, B. de Supinski, M. Klemm (2021). *OpenMP application programming interface specification version 5.2*. Independently Publishing. ISBN: 9798497370195 (cit. on p. 10).

Saad, Y., M. H. Schultz (1986). “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 7.3, pp. 856–869. DOI: 10.1137/0907058. eprint: https://doi.org/10.1137/0907058. URL: https://doi.org/10.1137/0907058 (cit. on p. 16).

Schrader, T. P., D. Schneider, B. Uekermann (2023). *Replication Data for: Efficient Application of Accelerator Cards for the Coupling Library preCICE*. DOI: 10.18419/darus-3404. URL: https://doi.org/10.18419/darus-3404 (cit. on p. 41).

Schulte, M. (2022). *Chapter 6 - Iterative Methods for Sparse Linear Systems* (cit. on p. 16).

Siegel, J., J. Ributzka, X. Li (2011). “CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator”. en. In: *Journal of Algorithms* 5.2 (cit. on p. 19).

Trott, C., L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, G. Womeldorff (2021). “The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing”. In: *Computing in Science Engineering* 23.5, pp. 10–18. DOI: 10.1109/MCSE.2021.3098509 (cit. on p. 22).

Trott, C. R., D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilke (2022). “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4, pp. 805–817. DOI: 10.1109/TPDS.2021.3097283 (cit. on p. 22).

Uekermann, B., H.-J. Bungartz, L. C. Yau, A. Rusch (Oct. 2017). *Official preCICE Adapters for Standard Open-Source Solvers*. en (cit. on p. 2).

Uelschen, M. (2019). *Software Engineering Paralleler Systeme: Grundlagen, Algorithmen, Programmierung*. de. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-25342-4 978-3-658-25343-1. DOI: 10.1007/978-3-658-25343-1. URL: http://link.springer.com/10.1007/978-3-658-25343-1 (cit. on p. 7).

Williams, S., A. Waterman, D. Patterson (Sept. 2009). *Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*. en. Tech. rep. 1407078, p. 1407078. DOI: 10.2172/1407078. URL: http://www.osti.gov/servlets/purl/1407078/ (cit. on p. 43).

All links were last followed on April 12, 2023.

A LU and Cholesky Decomposition on GPUs

One of the latest features of Ginkgo is the support for the LU and Cholesky decomposition algorithms. In order to test them, we reuse the C-TPS C2 test case from Sec. 4.2.2 with a support radius of $20h$ s.t. the system matrices are not too sparse. We take all meshes from the *coarse* series up to $h = 0.006$. The LU decomposition exploits the fact that the sparsity pattern of Φ is symmetric, which leads to a significant performance increase. The results of this measurement are shown in Figure A.1. LU results are averaged across five runs, Cholesky is measured only once since its runtime is very high.

It becomes clear that the Cholesky decomposition yields runtime results that are completely infeasible for preCICE. The matrix factorization for $h = 0.006$ did not finish in a reasonable amount of time, hence its data point is missing in the plot. However, the runtime exceeds 100 s for all mesh sizes equal or lower to $h = 0.01$. This is too high for efficient usage in preCICE. The LU decomposition performs much better with the decomposition time being in an acceptable time range. It takes about ten seconds to decompose Φ for mesh size $h = 0.008$. However, $h = 0.006$ requires more than 30 seconds which is about ten times more than the QR decomposition. We conclude that using the QR decomposition is much more advisable than using LU and Cholesky.

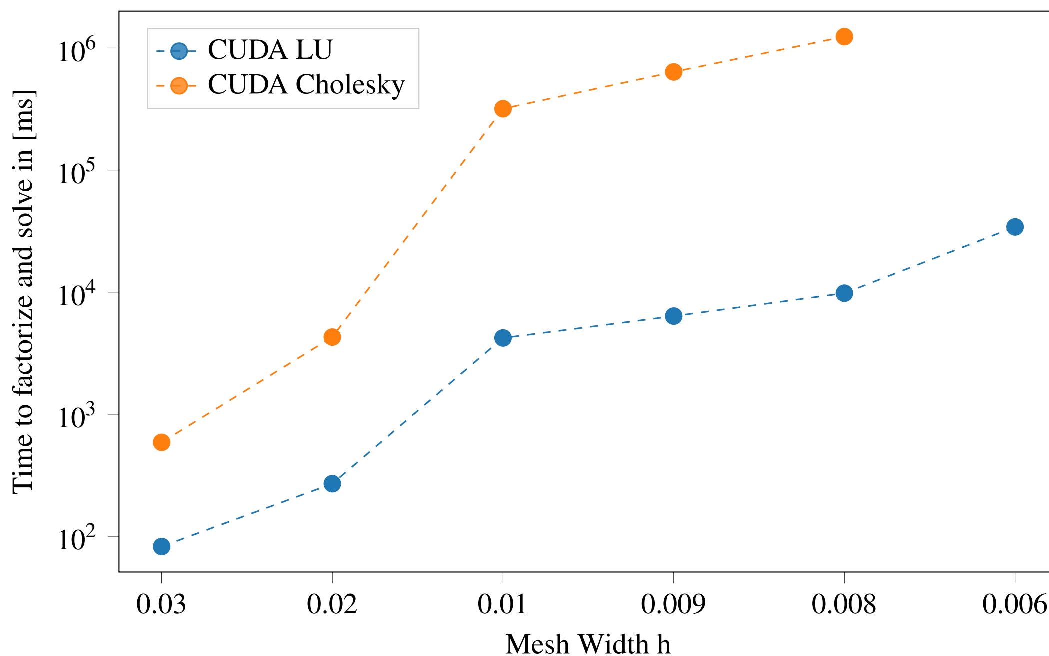

Figure A.1: The time that it takes to decompose the system matrix and solve the resulting triangular system using LU and Cholesky on CUDA. It can be seen that Cholesky needs much more time than any other solver tested in this thesis.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Friolzheim, 13.04.2023

place, date, signature