Imperial College London

Department of Aeronautics

Portable shared-memory parallelisation strategies for high-order finite element codes

A thesis submitted for the degree of

Doctor of Philosophy

Jan Robert Eichstädt

CID: 00883791

Supervisor: Professor Joaquim Peiró

3rd November 2020

Abstract

This thesis investigates strategies to parallelise numerical simulation software frameworks in the context of high-order finite element solvers. Due to the recent evolution of high-performance computing (HPC) hardware from single-core central processing units (CPUs) to multi-core CPUs and the emergence of general purpose graphics processing units (GPGPUs or short GPUs), these shared memory systems are now characterised by a high degree of parallelism. They also possess a variety of architecture-specific features that must be appropriately utilised in order to fully realise their capabilities. Large-scale legacy solver frameworks thus face the question how to adapt their codebase in the most effective way, to enable performance and portability across a heterogeneous landscape of parallel HPC systems, while ensuring code maintainability.

Various approaches to address this challenge are considered using practical investigations of core numerical algorithms for high-order finite element methods in order to ensure representative performance profiles. The selected core algorithms are a parallel mesh-optimisation method that optimises elemental shapes by minimising a deformation energy, and an implicit solver of the Helmholtz equation which forms the computationally most demanding part of a complete incompressible Navier-Stokes solver.

Overall the results presented here clearly demonstrate that algorithmic adaptations have to be introduced to enable execution on modern CPU and GPU systems and to achieve reasonable performance. Key aspects for an efficient mapping of the algorithms onto the hardware are considered: colouring approaches in connection with reduction operations, alongside determining the most effective manner of utilising multi-level nested parallel algorithms on multi-level parallel shared-memory hardware. It is further shown that the layout of data structures and their placement in memory is often the most important aspect in improving performance for both CPU and GPU systems. In order to achieve efficient memory access over vectorised operations, an interleaved data layout is developed. In a novel approach for elemental evaluations that minimises memory bandwidth, performance profiling demonstrates that on GPUs faster runtimes can be achieved, despite additional arithmetic operations. The balance between the utilisation of different hardware systems is assessed using runtimes, performance modelling, and a novel operational cost metric, and demonstrates that executions of the investigated algorithms on GPUs are both faster and more cost-effective than on CPUs for the problems under consideration.

Apart from the algorithmic adaptations, two programming paradigms are investigated: the effectiveness and performance of a single codebase, augmented with a variety of of portable programming models is compared against the implementation of a small number of hardware-specific kernels in regions of computational hotspots. The analysis here suggests that although the considered portable programming models (*OpenMP*, *OpenACC*, and *Kokkos*) should be easier to maintain, robust compiler support for all hardware types and unambiguous interpretation of compiler directives is not always guaranteed. The investigated GPU-specific *CUDA* kernels are most performant and achieve two to three times faster runtimes than all other models. Further work is proposed to follow the second paradigm in order to efficiently parallelise large software frameworks for high-order flow solvers.

Copyright declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Declaration of originality

I present this thesis for consideration for the degree of Doctor of Philosophy at Imperial College London exclusively. I certify that it contains the results of my own work and research during my doctoral studies, unless otherwise stated. In such cases, I give proper acknowledgement for methodologies and data that are not my own.

Jan Eichstädt, 3rd November 2020

Acknowledgements

First and foremost, I want to thank my supervisor Professor Joaquim Peiró for his constant and very helpful guidance. I am very grateful for his belief in my abilities by supporting my initial application to pursue my PhD research under his supervision. His wide expertise has been invaluable to view the wider connections within the computational engineering field, and his emphasis on clear and concise writing and presenting has improved my own style and confidence in presenting my work. I am also very thankful to Joaquim for enabling the opportunity for collaboration with Professor Jaime Peraire from the Massachusetts Institute of Technology, which has been a unique experience to strengthen transatlantic ties and become a better researcher.

I also want to thank my senior co-author Dr. David Moxey for his advice over the last four years and for numerous discussions of ideas that guided the direction of my research. I am very grateful to Dave for directing me through the intricacies of the spectral element method and the *Nektar++* framework, and how to become a better software developer.

My research has been supported by the President's Scholarship of Imperial College London. I would like to thank the College for awarding me with this prestigious award and for its sponsors for their monetary contributions, that not only supported my living expenses in London, but moreover provided me with a connection to outstanding scholars of various scientific disciplines.

Further, I would like to thank Dr. Mashy Green, who was always happy to discuss the latest developments in computing hardware and best practices for GPU programming, which has fuelled my enthusiasm and provided me with a great start into this area.

As part of the *Nektar++* community, I owe special thanks to Dr. Chris Cantwell and Dr. Martin Vymazal for their help to find my way around the *Nektar++* codebase and for general high-performance computing issues. I very much thank Professor Spencer Sherwin for keeping a keen interest in my work and for sponsoring a short research grant via the PRISM framework to support the completion of my research.

From the Department of Aeronautics, I would like to thank Dr. Peter Vincent for his advice and feedback on my early research results and Dr. Nikki Loppi for helpful discussions. Their expertise on GPU programming for high-order methods has been very valuable for my work.

A four-year long PhD spans a very long time with hundreds of lunch and coffee breaks,

not only to discuss work, but also many aspects of life. I therefore thank my colleagues and friends Alfonso, Andrea, Jan, Julian, Giacomo, Marius, and Michela for making it such an unforgettable and worthwhile time.

Finally, I would like to thank my parents Birgit and Ralf and my sister Daniela for their loving support and their unconditional trust. They let me venture out over the seas, while always providing a safe harbour to return to.

Contents

List of Algorithms

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Major efforts within the computational fluid dynamics (CFD) domain are targeted at extending the capabilities of high-fidelity methods for industrial-scale applications.

Direct numerical simulation (DNS) methods, that can accurately model the flow of any fluid continuum, are solving the full Navier-Stokes equations and spatially resolve the domain to extremely fine so called *Kolmogorov* scales. Since the computational costs for such simulations are prohibitive in all but simple academic cases, simpler methods have been developed that do not resolve but model the fluid behaviour at the smaller scales, for example using Reynolds-averaged Navier-Stokes (RANS) simulations. Many macroscopic flow behaviours occurring in transitional, separated, or unsteady flows however depend on accurate fine-scale discretisation, posing a limit to the applicability of low-fidelity solvers. Between the two extremes, a number of high-fidelity methods, such as large-eddy simulations (LES), have been proposed that resolve the larger turbulent scales of the flow accurately and only resort to modelling for the smallest scales that are comprised of statistically homogeneous flows.

For an effective use of high-fidelity methods, a good quality spatial discretisation of the domain is necessary. While low-order methods, such as finite volume methods or classical finite element methods, are suitable for steady solvers, they are not able to accurately propagate all flow features in unsteady flows. High-order spectral element methods [94], which use shape functions of higher polynomial orders, however offer lower dispersion and diffusion errors, making them more suitable for unsteady simulations.

The range of applications that can benefit from such high-fidelity methods are vast and cover simulations of full aircraft configurations over the whole flight envelope, transient operations of aircraft engines, aero-thermodynamics of complete gas-paths in turbomachines, complex automotive vehicle aerodynamics, industrial chemical reactors and burning chambers, wind power plants in irregular terrain, tidal flow hydrodynamics, and atmospheric flows for improved weather and climate predictions, to name but a few [105, 118].

The more accessible of these problems are characterised by lower Reynolds numbers, that allow for coarser spatial and temporal discretisation, and by shorter convective time-

scales (that depend on the spatial problem size and a characteristic velocity), both reducing the number of time-steps required. Most classic CFD methods follow the method of lines (MOL) [101], where the time-steps are executed consecutively. Hence, the overall simulation's wall-clock time is limited by the the number of required time-steps and their individual wall-clock time. As the execution of each time-step has a limit to its potential parallelisation, the simulation's wall-clock time becomes – at the strong scaling limit – independent of the size of the computing system at hand. Larger spatial problems on the other hand can be addressed by using larger computing systems, so that the overall wall-clock time only increases linearly with the convective time-scale.

To obtain efficient methods and algorithms, it is therefore crucial to increase the simulation time of each time-step by utilising methods that are limited by physical phenomena that need to be resolved, rather than by numerical stability limits. A second crucial aspect is to decrease the wall-clock time of each time-step by exploiting suitable parallel algorithms.

Some typical flows in aeronautical engineering, for example turbulent boundary layer or low Mach number flows, yield a very stiff system due to the large ratio of the eigenvalues that correspond to the propagation of pressure waves versus the fluid advection. While explicit time-stepping methods can efficiently compute individual time-steps requiring only a small memory footprint, for stiff systems they also exhibit severe stability restrictions that limit the advancement in simulation time per time-step. As an alternative, implicit time-stepping methods possess a larger stability region, which however comes at the expense of larger wall-clock times per time-step and an increased memory footprint. The more suitable time-stepping method therefore depends on the physical system and its stiffness, but also on the implementation and the underlying hardware.

1.1 Objectives of this work

In this work, strategies to develop efficient codes for such high-fidelity solvers shall be investigated. In order to obtain efficient implementations, algorithmic aspects need to be considered in conjunction with the computing hardware on which the algorithms should be executed, and the programming models in which they should be expressed. These three aspects of algorithms, hardware, and programming models and their co-design are a central aspect of this thesis.

Current high-performance computing (HPC) systems are clusters of thousands to millions of compute cores that are interconnected on at least two levels, multiple cores on individual chips, and the chips in turn by an interconnect network. In this context, distributed and shared memory parallelism need to be distinguished. Individual ranks or threads in a distributed memory model rely on data being exchanged as messages between nodes in the compute network, whereas individual threads in a shared memory environment can all access the same memory space, examples being multi-core central processing units (CPUs) and co-processors or accelerators like graphics processing units (GPUs).

Here, only *shared memory* parallelisation strategies are considered, as distributed memory strategies are already in a much more robust and mature state [21].

Over the last decade, the increase in processor clock speed has stalled due to thermal limits, so HPC systems are becoming ever more powerful in terms of operation count, just by adding more compute cores. At the same time, the memory speed is increasing at a slower rate, due to the physical limits at which information can travel through circuits of the system both on chip and via the interconnect network [100]. This favours highly *parallel algorithms* with high arithmetic intensity, because it is not the operation count, but memory access and communication costs that increasingly become the bottlenecks in applications. This aspect is therefore taken into account for all algorithmic designs that are investigated.

The choice of suitable programming models is another important aspect. As the life-cycles of scientific applications are rather large, often spanning more than a decade, many current scientific computing frameworks originate from the single-core CPU era. Yet, their algorithms are often very compute-intense and require updated capabilities to benefit from the increased parallelism of modern HPC systems. Furthermore, the current and projected heterogeneity of HPC systems benefits *portable programming models*, that can address a variety of different hardware vendors and systems and allow for good code maintainability. In an evolutionary approach, a high-level performance-portable programming model allows the adjustment of the existing frameworks step-by-step by introducing parallel instructions into the code-base, but might induce a performance penalty. An alternative option is to maintain performance-optimised kernels for a small number of different hardware architectures and extend these when necessary. In both cases, it might suffice to concentrate the development efforts to a limited number of kernels, in which the applications spend the majority of the computational time.

Throughout this thesis, we[1] consider high-order spectral-*hp* element methods, as introduced in reference [63]. These utilise high-order tensor-product polynomial expansions as shape functions within each element, but also allow for element mappings to represent curved solid boundaries accurately. Such discretisations allow for high arithmetic intensity, because for elemental evaluations the number of operations scale with a higher power of the polynomial order than the number of degrees of freedom. This flexibility in tuning the arithmetic intensity makes high-order methods good candidates for efficient implementations. More details on this scaling are provided in Section 5.3.5. However, before being widely employed for industrial purposes, high-order methods need to improve on their robustness and degree of automation, on the availability of efficient high-order meshing capabilities, and on the scalability of their performance for HPC systems [105].

In order to contribute to these goals, we focus on the computational hot-spots of core-algorithms for high-order flow solvers and their spatial discretisation. As we conduct

[1] While I am the single author of this work, I use the first person plural forms for the remainder of this thesis. This way I would like to recognise the contributions of my co-authors for publications that are associated with individual chapters.

numerous parameter studies and develop numerous implementations, focussing on these hot-spots is more effective than working with a complete Navier-Stokes solver.

Considering the incompressible Navier-Stokes equation, an efficient solver can be devised using a splitting scheme, as introduced in reference [64]. Within this approach, the most compute-intense part is solving the Helmholtz equation with an implicit scheme, as explained in more detail in Section 4.2.1. The solver can be split into multiple parallel kernels, that each have their own challenges in terms of efficient parallelisation.

Mesh-optimisation methods for *a posteriori* mesh generation or mesh-adaption also contain compute-intense elemental evaluations, embedded in various optimisation schemes, as discussed in Section 3.2. Here the global optimisation problem is solved with a relaxation approach and mesh colouring, that introduces parallelism to the method.

We therefore use these core algorithms as a vehicle to investigate how to accelerate large high-order software frameworks for heterogeneous hardware platforms. We base our investigations on the *Nektar++* framework [14, 83] for the Helmholtz solver and its embedded meshing framework *NekMesh* [114] for the mesh optimisation.

Related projects that work on increasing the efficency of high-order finite element codes include *Deal.II* [4], *MFEM* [3], *DUNE* [9], and *Firedrake* [98]. As in our case, all frameworks started off with CPU support only. During the course of this thesis, *Deal.II* and *MFEM* have additionally introduced GPU support using different programming models, however not all capabilities are covered, yet. Furthermore, the presented GPU performance benchmarking results are still too limited to thoroughly inform the choice of suitable programming models and algorithmic adaptations. In terms of capabilities, *Deal.II* and *DUNE* only support quadrilateral and hexahedral elements, whereas in this work we also focus on triangular and tetrahedral elements, that allow for more flexible and robust mesh generation methods. *Firedrake* is a code-generation framework, which greatly increases user productivity, and also shows very good performance on multi-core CPU clusters. Code-generation however constitutes a fundamentally different approach to achieve performance portability, which cannot readily be adopted by legacy frameworks.

Our investigations now target modern shared-memory systems including multi-core central processing units (CPUs), co-processors and graphics processing units (GPUs). Within the context of high-order finite element methods, we compare a number of programming models and investigate the trade-off between usability, portability, and performance. In order to adapt existing methods to these heterogeneous systems with a larger number of processors, we devise a number of strategies to increase the parallelism of the algorithms and map it to the parallelism of the hardware, as well as to reduce the memory intensity. Our investigations shall also be transferable to a range of other methods and application areas within scientific high performance computing.

1.2 Outline of this thesis

The main body of this thesis is composed of four chapters:

- Chapter 2 on two central underlying themes of this work: current HPC hardware and HPC programming models.
- Chapter 3 on accelerating a mesh optimisation method using *Kokkos*, a performance portable programming model.
- Chapter 4 on a comparison of three performance portable programming models in the context of an implicit time-stepping solver of the Helmholtz equation.
- Chapter 5 on the development of performance-optimised *CUDA* GPU-kernels for the Helmholtz equation.

All four chapters contain extensive discussions on algorithmic adaptations that are required to harness the computing power of parallel shared-memory systems. While algorithms lie on the spectrum between fully serial and embarrassingly parallel, we aim to efficiently map these to the underlying hardware. On the basic level, this translates to finding strategies that map the parallelisation inherent in the common algorithms to the parallel hardware execution model. On an intermediate level, algorithms often need to be adapted to exploit more parallelism, for example using mesh colouring approaches or adapted indexing techniques. This is explored in detail in Chapters 3 and 4. On the advanced level, flow solvers need to be redesigned using inherently more parallel algorithms and methods that are less memory intense, that had formerly been discarded because of then dominant requirements, like FLOP counts. Some of these techniques are presented in Chapter 5.

The aspect of co-designing algorithms with the layout of data structures in memory is considered throughout this work. Since high-order finite element algorithms are generally memory-bound, this becomes a central performance factor. Following these lines, we develop novel vectorised algorithms with strided memory access in Chapters 4 and 5. Optimising memory access by utilising different memory spaces on GPUs is investigated in Chapter 5.

Additionally, we evaluate different programming models in terms of usability and performance portability. This is a major theme within Chapter 3 in which we describe the steps to port a legacy applications to GPUs, and in Chapter 4 in which three popular portable programming models (*Kokkos*, *OpenMP*, and *OpenACC*) are compared on multi-core CPUs and GPUs using a *mini application* of an implicit solver for the Helmholtz equation. Ultimately, we test a core part of the same algorithm in Chapter 5 using *CUDA*, a specific programming model for GPUs, in order to extend the comparison to a model that emphasises performance over portability.

From these investigations, conclusions are drawn and further related work is proposed in Chapter 6. We summarise our strategies to efficiently parallelise high-order finite ele-

ment codes for modern computing systems and to achieve increased resource utilisation. We demonstrate runtime speed-ups and cost savings that are harnessed by executing all developed algorithms on parallel shared memory systems, compared with benchmarking algorithms on CPUs. The algorithmic developments to port our methods to massively parallel hardware are presented in a general fashion, allowing transferability to other methods. We further derive recommendations to choose from either portable or hardware-specific programming models to adapt scientific applications for modern HPC systems.

1.3 List of related publications

The work presented in this thesis already resulted in several publications, both peer reviewed journals, and conference proceedings.

The work associated with Chapter 3 is published in references:

- [26] J. Eichstädt, M. Green, M. Turner, J. Peiró, and D. Moxey. Accelerating high-order mesh optimisation with an architecture-independent programming model. *Computer Physics Communications*, 229:36–53, 2018

- [27] J. Eichstädt, D. Moxey, and J. Peiró. Towards a performance-portable high-order implicit flow solver. In *AIAA Science and Technology Forum 2019*, January 2019

The work associated with Chapter 4 is published in references:

- [29] J. Eichstädt, M. Vymazal, D. Moxey, and J. Peiró. A comparison of the shared-memory parallel programming models OpenMP, OpenACC and Kokkos in the context of implicit solvers for high-order FEM. *Computer Physics Communications*, page 107245, 2020

- [27] J. Eichstädt, D. Moxey, and J. Peiró. Towards a performance-portable high-order implicit flow solver. In *AIAA Science and Technology Forum 2019*, January 2019

Parts of the work associated with Chapter 5 are published in reference:

- [28] J. Eichstädt, D. Moxey, and J. Peiró. Efficient vectorised CUDA kernels for high-order finite element flow solvers. In *UKACM Conference 2020*. figshare, April 2020

Chapter 2

Review of high performance computing hardware and software

In this chapter, we provide background information on the current high-performance computing environment that forms the backdrop of this work. Section 2.1 presents a detailed description of the current hardware landscape for high performance computing and a discussion on its implications on developing efficient implementations. Section 2.2 discusses and evaluates the extensive variety of programming models available for these hardware systems.

2.1 Modern high performance computing hardware

The development of computing hardware has been progressing at an immense pace over the last decade, and will most likely continue to do so for the foreseeable future. The prediction of further developments however remains difficult. Even within the time-span of writing this thesis, the hardware landscape has changed considerably, so that for example one hardware architecture that we benchmarked within our research (the *Intel Xeon Phi* architecture) has already seen a dead-end. Although published a few years ago, the *CFD Vision 2030 Study* [105] still provides a credible forecast of the HPC hardware landscape a decade from now.

In what follows, we will describe the current HPC architectures and their predicted development, and infer how these will shape the algorithmic pathways that shall be followed to develop efficient implementations. Starting with a historical overview in Section 2.1.1, we subsequently cover the modern CPU architectures in Section 2.1.2, describe the GPU architecture in detail in Section 2.1.3, as this forms the basis for most of the research conducted for this thesis, and derive some common trends in Section 2.1.4.

We restrict the discussion to shared memory systems, that typically make up individual nodes of larger HPC clusters, or compose a single workstation. The development of distributed memory systems (whole HPC clusters) is not discussed, because their development is comparatively static and does not require to redesign algorithms fundamentally.

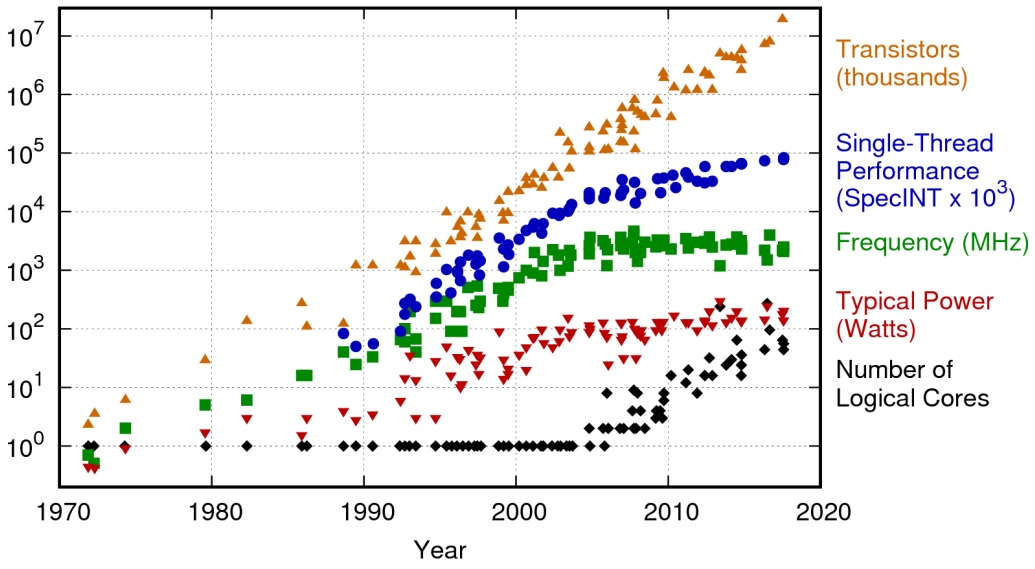

Figure 2.1: 42 years of processor trends (CPU architectures) [100].

More specialised architectures like field programmable gate arrays (FPGAs) will not be considered either, as they lack the flexibility needed for the considered algorithms.

2.1.1 Historical developments

Throughout the 1990s and early 2000s, HPC systems would almost exclusively be based on regular central processing unit (CPU) microprocessors. These are scalar single-instruction single-data (SISD) processors that are in turn specialised for scalar operations. During this time microprocessors were priced so competitively, that no specific hardware for vector processing or other parallel operations have been employed. However before that time, during the 1970s and the 1980s, vector processing machines were routinely used in HPC.

Around the year 2005, processor development has reached the energy barrier of the processor clock speed. Power consumption and hence heat generation in transistors scale with the clock speed to about the third power [81], so that hardware cooling systems would have had to become overly sophisticated and energy efficiency would have dropped rapidly. In the years leading up to that point, CPUs have been single threaded and processing performance has been increased by scaling up the clock speed, up to about 4GHz. This did not require any architectural changes and hence allowed for lower research and development costs. Instead, at this point it became necessary to develop multi-core CPUs and longer vector-lanes to further increase the performance per chip. Since then a new power law with respect to the number of CPU cores is developing. Using the new CPU architecture it is then possible to reduce the clock rate again to around 2-3GHz, while keeping the overall performance constant, thus allowing for lower electricity consumption. All these trends are illustrated in Figure 2.1.

At the same time, it was realised that the usage of graphics processing units (GPUs)

could be extended from their traditional role. Until then GPUs had been optimised for fast logical or arithmetic (floating point) operations in order to render screen output in real time. Accordingly, GPUs contain in the order of hundreds to thousands of *simple* processing cores. These are fundamentally different from CPU cores, as they are not independent units due to their lack of logic and instruction circuits. Each of these *simple* parallel cores, however, could theoretically also manipulate other, more generic data than graphics data. In the following years programming paradigms and the development of the hardware itself adopted this trend of general purpose graphics processing unit (GPGPU) computing.

The philosophy of employing GPGPUs has lead to a renaissance of the idea of employing vector processors. It allows for the option to run specific types of operations on a hardware architecture that is most suited for its type. This will improve the utilisation rate of the hardware and ultimately speed up the processing time and drive down capital investments [78]. In terms of efficient GPGPU computing, vector or other parallel operations have to be offloaded from the CPU onto the more specialised GPU or accelerator systems.

Other parallel architectures like *Intel Xeon Phi* co-processors were also making their appearance. These many-core processors have more cores than standard multi-core CPUs, but fewer and more sophisticated cores than GPGPUs. Accordingly, they provide a *hybrid* solution between multi-core CPU-clusters and GPGPUs. We believe that this niche has not been economically viable, so that their development and support has been discontinued again.

To date, many of the Top500 supercomputers consists of heterogeneous nodes that possess multi-core CPUs, many-core co-processors and GPGPUs [109]. For an illustration, see Figure 2.2, in which the supercomputers are categorised based on their accelerator family. More than one third of the systems are using Nvidia GPGPUs of different generations (*Volta*, *Pascal*, *Kepler*).

2.1.2 Multi-core CPUs

CPUs traditionally follow the SISD (single-instruction single-data) model, so that a large fraction of transistors are dedicated to dispatch and other control units than actual arithmetic or function units. After hitting the energy barrier in terms of clock rate, CPUs became more parallel on two levels. The number of independent cores has subsequently increased, but also the performance of individual cores has been improved. Accordingly, recent CPUs have multiple cores and make more and more use of techniques like pipelining, simultaneous multi-threading (SMT), and vectorisation or SIMD instructions. Pipelining allows the parallel execution of the typically five steps of the basic CPU instruction cycle, leading to a speed-up of a factor of five but a remaining latency of five cycles. In SMT two or more independent threads are sharing the execution resources of one CPU core, leading to a higher utilisation. Vectorisation is achieved by larger register widths of so far

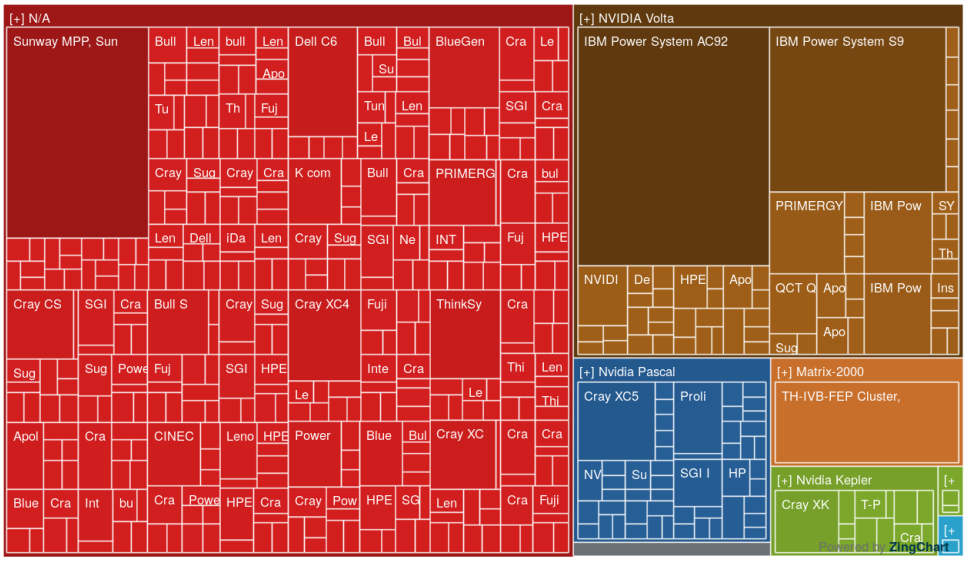

Figure 2.2: Top500 supercomputers by accelerator family, generated on Top500.org, September 2019.

128 to 512 bits that can amalgamate 2 to 8 double-precision floating-point numbers for data parallel processing. Better single thread performance can also be achieved via more elaborate path-prediction methods, that vendor-specific compilers can introduce under the hood. While the parallelism between multiple cores can be straightforwardly specified with many programming models, the parallelism within a single core is very hard to address.

These trends can be seen in Figure 2.1, showing stagnating clockspeeds after 2005, increasing number of cores, and still increasing single-thread performance.

2.1.3 Many-core GPUs

In this section, the architecture of general purpose graphics processing units (GPGPUs) is described. These are graphics processing units (GPUs), that are able to not only do their originally intended task of rendering graphics output to a screen (i.e. for computer aided design (CAD), or gaming applications), but also to do more general calculations (i.e. for high performance computing or machine learning). Some more recent GPGPUs do not even possess a graphics output bus any more, and are installed in supercomputing workstation or clusters, in which cases they are probably more accurately described as GPU-based accelerators or co-processors. In this thesis we will interchangeably use the terms *GPGPU*, *GPU-accelerator*, or simply *GPU*, to denote all GPU architectures for general purpose computing.

We describe their architecture and the way they manage the parallel execution of multiple threads. It will become clear why a sophisticated GPU memory hierarchy is necessary and which implications the GPU architecture has on its programming methods.

Processor architecture

A GPU is a separate device from the CPU and no physical memory banks are shared between them. Further, the GPU can not operate on its own, instead its operations must still be initialised by the host CPU. As mentioned above, GPUs are traditionally used to render pixel or vertex based graphics, which are inherently parallel operations. Accordingly, an architecture for computing the same instructions on many data elements in a parallel fashion is required. The GPU architecture thus follows the SIMD (single-instruction multiple-data) model, referring to Flynn's taxonomy [33]. Since only a single dispatch unit is needed to load instructions to multiple threads, a higher fraction of transistors can be dedicated to arithmetic or function units.

Each SIMD processor of a GPU allows for data-parallel operations, however no concurrent task-parallel operation is possible. To allow for more such flexibility, GPUs are composed of independent SIMD- or streaming multiprocessors (SM or SMX), up to 15 for the early Nvidia Fermi GPUs [43], up to 60 for the Nvidia Pascal GPUs [86], and up to 84 for the latest Nvidia Volta architecture [87]. A GPU can then be thought of as a massively parallel co-processor that can perform multiple-instruction multiple-data (MIMD) operations.

The details about the thread scheduling and the memory hierarchy are explained with reference to the Nvidia GPU architectures. These are the more relevant architectures for this work, since the available hardware for this research and the general HPC landscape has been dominated by these, as seen in Figure 2.2. The alternative AMD GPU architectures are similar though, for more details the reader is referred to references [76, 2]. Only at the time of completing this thesis, the AMD GPUs and *HIP* [40], a suitable programming model are becoming more popular.

Thread scheduling

The way in which threads are managed depends on the GPU hardware, but is tightly linked with the programming model or application programming interface (API), which for Nvidia GPUs is *CUDA* (Compute Unified Device Architecture, see Section 2.2.3). Data on the GPU memory are operated on by kernel functions, in which each data element occupies one *CUDA*-thread. GPU- or *CUDA*-threads are far more lightweight than CPU threads and can be scheduled and dispatched within a few cycles.

These threads are allocated into blocks, where the number of threads per block depends on the implementation, but is limited to a maximum of 1024 by the hardware [86]. Threads within a block can synchronise and they share parts of the cache memory. Depending on the implementation, blocks can then be organised in one-, two-, or three-dimensional grids on an abstraction level.

It has to be noted, that threads on one SMX are always dispatched and processed in groups of 32 threads, called a warp. A thread-block will be separated into warps and in cases where the block size is not a multiple of the warp size, some threads in the last warp

are running idle. To optimise performance, it is thus best to chose the block size as a multiple of the warp size. Further, a diverging code (i.e. in if-then statements) can lead to a divergence in the warp, as the *if*-part and the *then*-part have to be executed one after the other, with effectively half of the threads being idle.

The 32 threads of a warp are dispatched to the GPU units by the SIMD thread scheduler or warp scheduler. These GPU units are integer arithmetic logic units (ALUs), floating point units (FPUs), but also special function units (SFUs) and load-store units (LSUs). Newer architectures have dedicated single precision floating point 32 bits (FP32) ALUs, as well as double-precision floating-point 64 bits (FP64) units for improved performance. With the *Volta* architecture, so called Tensor-Cores [87] have been additionally introduced, that can execute matrix–matrix multiplications of rank 4 with reduced precision in only one cycle, a feature that is intended to serve the needs of deep learning algorithms. SFUs provide tailored circuits to execute special functions, like sinus, cosines, square roots, reciprocal value or logarithms. The LSUs transfer data between the memory spaces and the registers. They operate much faster, if all LSUs can access contiguous or neighbouring data in memory. If the data is further aligned with *CUDA* memory blocks, the memory access is said to be *coalesced* and up to 16 data elements (a half warp) can be loaded from the global memory in the same cycle.

If each SIMD processor would only process one SIMD thread at a time, waiting times for the data elements to be transferred to the registers would be inevitable. Further, all GPU units apart from one group would be idle. To avoid this, a form of pipelining is employed, in which the SIMD thread scheduler controls up to 64 independent SIMD threads or warps at a time [86]. Once a warp is ready for execution, the thread-scheduler can dispatch a new warp. This concept is also called simultaneous multi-threading. Each thread that waits in the SIMD thread-scheduler has its independent registers. It follows that for an efficient implementation, many more threads than physical function units or cores are needed.

Memory hierarchy

Apart from using a thread-scheduler, a sophisticated cache memory hierarchy is employed to further hide memory latencies. Close to the arithmetic cores, small and fast caches are employed, whereas for the larger DRAM memory only reduced bandwidth and latency is achieved. Multiple intermediate levels are required to communicate between these extremes.

An illustration of the hierarchy of the different GPU memory spaces is shown in Figure 2.3. On the lowest level, each thread accesses data from its own registers. Apart from the registers, each thread has a local memory, which acts as a spillover memory for the registers, and each block has a shared memory. They both reside on the L1 cache, which latency is not published officially, but has been established using microbenchmarking techniques in reference [61]. Two commonly used GPUs in this work, the Nvidia

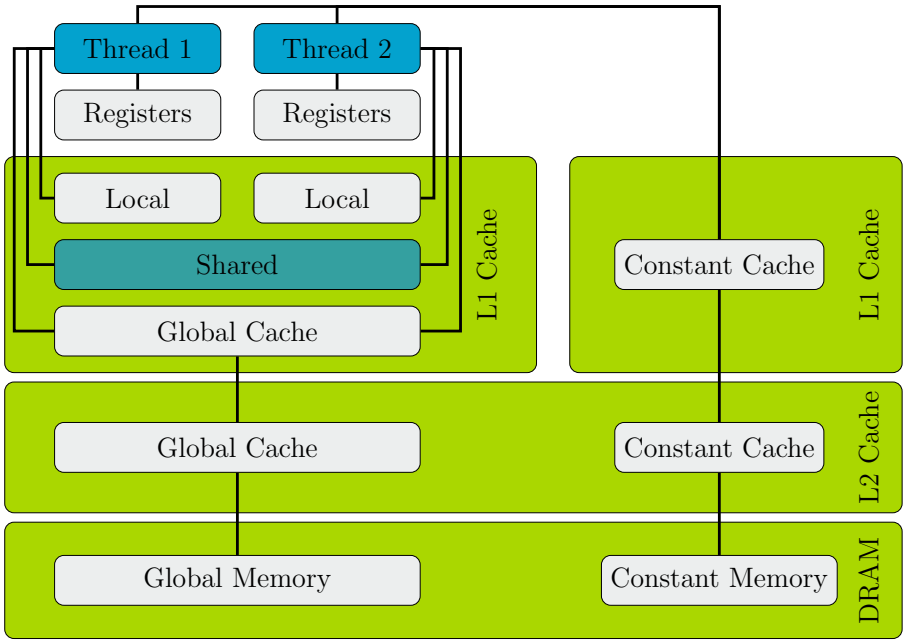

Figure 2.3: The Nvidia-*CUDA* memory model.

Titan V and the Nvidia Pascal P100, have L1 latencies of 28 and 82 cycles, respectively. All threads from all blocks have access to the GPU global memory, which resides on the DRAM memory, and is cached in L2 and L1 cache.

The latency of the global memory access from L2 cache is about 220 to 240 cycles for all recent GPUs [61]. However, the L2 cache's size is in the order of MBs, whereas the L1 cache is only 64KB per SMX on Nvidia Fermi, Kepler, Pascal, and Volta architectures[43, 85, 86, 87].

In addition, two read-only memory spaces are available, that allow threads a fast access to a larger memory space. These are constant memory and texture memory, which both use an independent cache system to speed up access time. Constant memory allows fast access, if all threads from one warp access the same memory address, otherwise they are serialised. Data on texture memory (not shown in Figure 2.3) is usually organised in a 2D pattern, so that data points that are spatially close to each other on a virtual 2D grid are also physically close on the linear memory bank. Different data patterns can be specified, that order the data on linear physical memory for efficient 1D, 2D, 3D, or other specialised texture fetches.

Since the GPU and CPU DRAM memory spaces are physically separated, transfer of memory from the CPU to the GPU, and vice-versa, is restricted by the PCIe bus speeds, which are much slower than the GPU DRAM memory. It follows, that the number and size of memory transfers between the CPU and the GPU memory should be avoided as much as possible.

Efficient programming for GPUs

The discussed GPU architecture and the *CUDA* programming model have a number of implications on how to code an efficient implementation.

Potentially the most important aspect to achieve best performance is avoiding the data transfer bottleneck between CPU and GPU memory as much as possible. In fact most implementations only show a speedup, once all kernels that are involved in the bulk of the computation (i.e. for time-stepping in CFD methods) are ported to the GPU. Alternatively, data transfers should be scheduled in a way that they are hidden by bigger computational tasks.

Once the data resides on the GPU memory, its micromanagement is of highest importance due to the high memory latency. Most HPC applications are memory- and not compute-bound, so that data elements need to be reused or shared between different threads to achieve peak performance. This can be facilitated by optimising the use of local and shared memory, which in turn is connected with the choice of thread block sizes. Optimising these low level parameters is very cumbersome, and their tuning would need to be repeated for every new architecture. It becomes clear, why a higher-level abstraction, that automatically optimises these parameters would be beneficial.

On a low level, code branching or code divergence should be avoided, as all threads in a warp always have to perform the same instruction.

For an efficient memory access, the data should be coalesced. This has implications for example on the storing and accessing of matrices. Whether a matrix is processed in row- or column-major order has therefore huge performance implications. To aid such memory access, it can also be beneficial to use zero-padding of arrays, so their overall length is a multiple of *CUDA* memory blocks.

To further reduce the memory bandwidth, is should be considered to utilise lower precision data types, i.e. single precision FP32 instead of double precision FP64 data types. This will already reduce the required memory bandwidth by 50%. This approach can further be exploited with the different kinds of Nvidia GPUs in mind, that have dedicated cores for either FP64 or FP32.

While professional GPGPU cards intended for HPC applications have a FP64 to FP32 ratio of 1:3 or 1:2, regular Nvidia GTX gaming cards have a large ratio of 1:24 or 1:32. If the use of single precision is permitted by the algorithms, it is not only desirable for the sake of computational speed, but can also be a much better economical choice, especially when configuring a workstation system with up to four gaming GPUs, since professional GPGPU cards cost about five times more than the equivalent gaming card.

In summary, a good GPU code utilises the different memory spaces effectively, minimises memory transfer, achieves coalesced memory access, and optimises the number of threads per block in order to utilise as much of the GPU capacity as possible.

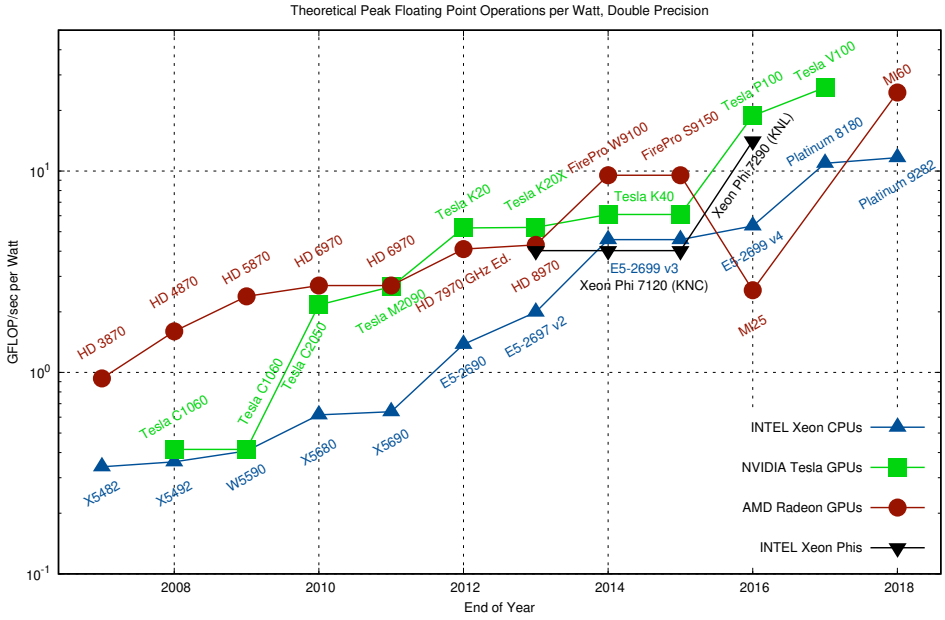

Figure 2.4: GFLOP per Watt history of different architectures [100].

2.1.4 Common HPC trends and implications

The goals of HPC hardware and software developments are, for a given task, to achieve the lowest processing time with the least capital investment. The investment for the hardware is determined by the acquisition costs and the operating costs, which are mostly driven by electricity consumption. As described above, the different CPU and GPU architectures are each intended for different kind of operations and should be employed accordingly. As for most HPC applications, compute-intense parallel operations should therefore be offloaded to an accelerator for an efficient use of hardware. This is well illustrated by Figure 2.4, that tracks the evolution of energy consumption of CPU and GPU systems. It shows that by avoiding large parts of general purpose circuits typical of CPUs, GPUs are able to be about 2-3 times as energy efficient for the same number of arithmetic operations. It follows, that implementations should be enabled to execute on co-processor architectures, to realise these energy savings, and to be future-proof by not being excluded to run on an increasing share of accelerated HPC systems. The general trend to more energy efficiency has been enabled by the decreasing size of the semi-conductor fabrication process, that roughly evolved from 45nm processes in 2007 to 7nm processes in 2018.

It is extremely difficult to predict the future development of HPC hardware, but most experts agree on a few trends indicating an ongoing paradigm shift [105]. As discussed before, the energetic limit of processor clock speeds has already lead to multi- and many-core architectures, like Intel's Xeon Phi co-processors and Nvidia or AMD general purpose graphics processing units (GPGPUs). Despite their differences, these architectures have many aspects in common, for example their hierarchical parallelism. Whereas a GPU is composed of multiple independent units called streaming multiprocessors (SMX), a CPU's

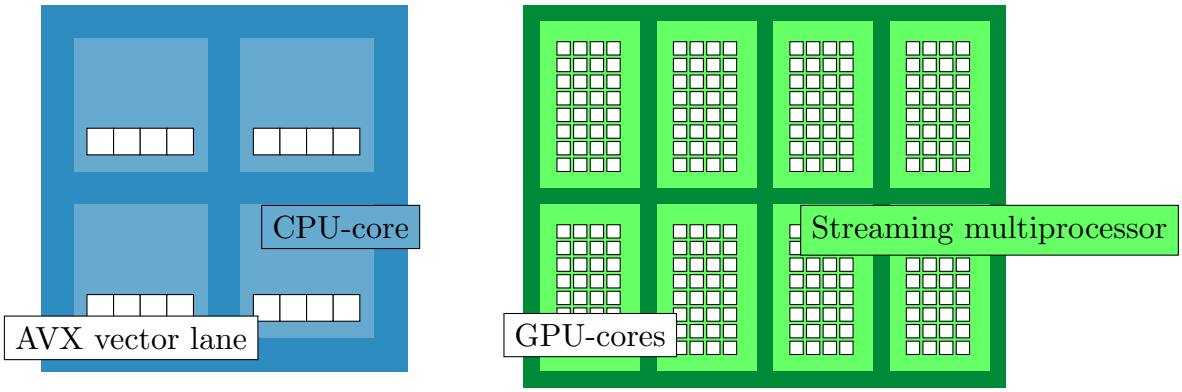

Figure 2.5: Hierarchical parallelism of current CPU and GPU architectures in comparison.

smallest independent units are its cores. On the lower parallel level both are using SIMD type parallelism, or vectorisation: on a GPU in form of 32 threads that form a warp, on a CPU in terms of AVX vector lanes of up to 8 FP64 data elements. This is illustrated in Figure 2.5. To further increase their computing power, future HPC systems will most likely be increasingly heterogeneous, consisting of specialised components of scalar processors in combination with many-core stream- or vector processors [105]. Implementations therefore need to be inherently parallel, in order to map efficiently to the parallelism of the hardware. The hierarchical structure of the architecture further makes algorithms with global reductions and synchronisations less efficient.

Additionally, the memory access bandwidth will most likely increase at a much lower rate than processing performance, as the speed at which data travels through the electric circuits is approaching its physical limits. The current trend for CPUs, Nvidia GPUs, and AMD GPUs is shown in Figure 2.6. It shows the flop-per-byte ratio or arithmetic intensity, that is the ratio between the theoretical maximum of arithmetic operations and the theoretical maximum DRAM memory bandwidths of the system. To feed enough data to the processing units, more and more sophisticated cache systems need to be developed. For algorithmic designs, it follows that global memory accesses should be avoided, and data be reused from higher cache levels.

This leads to a related trend that has not been taken into account in the *CFD Vision 2030 Study* [105], which is the new dominance of machine learning (ML) hardware. ML algorithms are often executed on GPUs and have induced a large majority of the GPU hardware demand over the last few years. While both classes of algorithms benefit from massively parallel hardware, the memory requirements for optimised ML hardware is fundamentally different than for classical HPC hardware that executes for example CFD algorithms. ML algorithms require lots of computing power in terms of dense matrix-matrix multiplications, but only low precision data types. CFD algorithms are more sparse and most often implemented with double precision data. The two different application areas are therefore characterised by largely different arithmetic intensities. For hardware developers it becomes increasingly difficult to develop a single chip architecture that serves

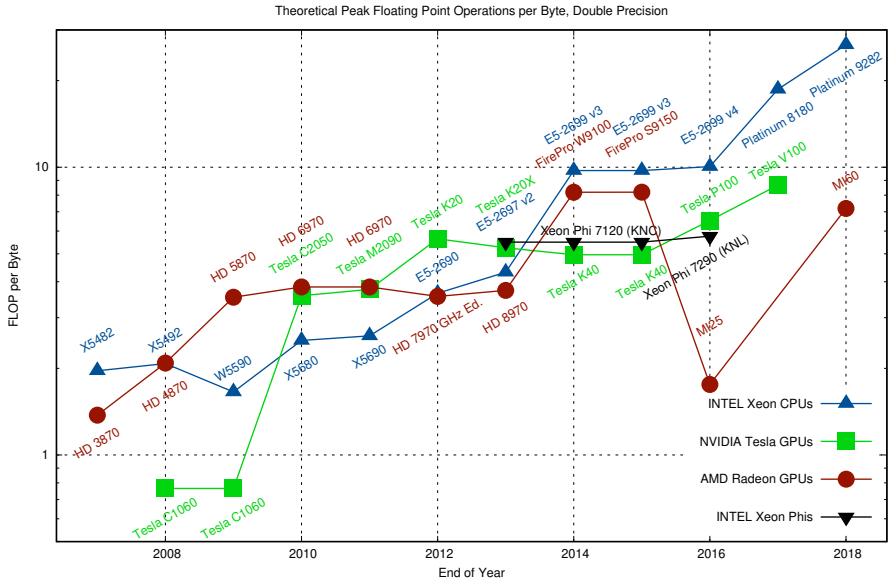

Figure 2.6: FLOP per byte history of different architectures [100].

both types of algorithms. The economical decisions undertaken by HPC hardware vendors might therefore already be inclined more towards the requirements of the ML area, as this market is projected to grow at a much higher rate.

This increasingly means that the power of the classical HPC community to influence strategic hardware developments is becoming weaker. The algorithmic development efforts of the classical HPC community must therefore anticipate the projected hardware developments and adapt its algorithms accordingly. While the scaling of algorithms remains fundamental, it is not just the FLOP count that is determining the runtime and efficiency of algorithms any more. In general, this translates to developing algorithms with inherent parallelism and few synchronisations, high arithmetic intensity, low memory footprints, and low precision requirements.

In terms of software development, effective programming in this environment of ever-changing hardware and uncertainty is hence going to remain challenging. The utilisation of programming paradigms that are not specific to only one hardware architecture will thus be beneficial.

2.2 Parallel shared-memory programming models

Parallel programming models can be viewed as an abstraction layer to address the individual processors of entire hardware system configurations and their interconnections. On a high level, two main parallel systems can be distinguished: shared memory systems in which all processors access the same memory space, i.e. multi-core CPUs; and distributed memory systems, in which each processor accesses only its own memory space, such as a single node of a HPC cluster. Naturally, distributed- and shared memory systems are often combined into so called *heterogeneous* systems, which are becoming more predominant

with the advent of GPGPUs and other many-core devices.

The *de facto* standard for programming on distributed memory systems using inter-node parallelism is *MPI* (Message Passing Interface) [117]. Since variables cannot be shared on such systems, they are passed by interchanging messages between the *MPI* ranks. *MPI* is an open-standard library that can be seamlessly called from *Fortran* and *C/C++* programs.

The dominant programming paradigm for shared memory systems using intra-node parallelism is *OpenMP* [92]. It uses compiler directives, so called *pragmas*, and is portable between different shared memory architectures and supports *Fortran* and *C/C++*. Yet, it is also possible to employ *MPI* for shared memory systems by creating one *MPI* rank per processing core. *MPI* can also be considered a good choice for task-parallel computations in conjunction with good load balancing.

However, to address heterogeneous architectures of CPUs in combination with GPUs or other accelerators, it is necessary to employ so called heterogeneous programming models. Reference [21] offers a review of these models and identifies two groups, that either integrate both the inter- and the intra-node parallelism in a PGAS (Partitioned Global Address Space) model, or use *MPI* and a different intra-node model in a hybrid approach, often termed an *MPI+X* model. Recent examples for the *MPI+X* model are numerous and show excellent scalability [53, 107, 65].

The PGAS model supports a global address space, but the threads are memory locality aware, so they can distinguish between local or shared versus remote memory access. Theoretically, this can enable an increased performance compared to employing only *MPI* on a shared-distributed memory system. The integrated PGAS model would be appealing in terms of reduced complexity and thus better maintainability. On the other hand, legacy HPC code that already employs *MPI* to achieve distributed parallelism should be easily extendable to an *MPI+X* model. In practical terms, no PGAS model that would support *C/C++* and modern heterogeneous architectures like GPGPUs were identified at the start of this research. Only very recently, the *Julia* programming model [11] shows some indication that it is developing in this direction.

In this work, we therefore restrict our investigation to shared memory models, being the *X* in an *MPI+X* model.

A few classifications can be devised. Models that are specific to one type of hardware architecture only are called *native* models, as opposed to *portable* models that can address multiple hardware types. Depending on the abstraction level, it can be distinguished between low-level models that enable or require extensive control of the program close to the hardware, versus high-level models with a higher abstraction in which compilers will make the low-level control decisions. These can be aspects like the explicit placement of data in memory, or thread scheduling.

2.2.1 Requirements

To assess the suitability of different programming models for scientific computing, three main criteria are investigated: usability, portability, and performance.

The current coexistence and future uncertainty of different HPC hardware architectures favours an easily maintainable code. This is ideally achieved by supporting a single codebase that allows portability across architectures by using different compilation targets, instead of maintaining specific and separate implementations for each architecture. Current architectures that should be supported are multi-core CPUs, GPGPUs (ideally from both main vendors, Nvidia and AMD) and (at least at the time of starting this research) co-processors like Intel *Xeon Phi's*. It is also very desirable to have a codebase that is *performance portable*, meaning it can achieve nearly optimal performance on all such architectures, without architecture-specific branches in the code-base.

A maintainable code-base would also avoid to over-optimise individual kernels, so that the physical equations remain recognisable and hence accessible. A high-level programming paradigm, where the developer can focus on the actual method is thus in favour over a low-level language. Another aspect is to only rely on third-party libraries or tools that promise a continued support of their capabilities. Especially in an academic setting it is beneficial to only work with open-source projects, so that no part of the code relies on commercial products. This is especially relevant for compilers. Some more cutting-edge programming language specifications are first only supported by commercially available compilers and require a few more years of development effort of the community before these specifications are also supported by open-source compilers like *gcc*.

Many scientific codes have the aspiration to be suitable for high-performance scientific and industrial applications. They must therefore efficiently utilise the underlying HPC hardware, not least to receive runtime grants on leading HPC clusters. However, many academic simulations are one-off tasks, rather than industrial production runs, so that efficiency alone should not be the sole priority either.

2.2.2 CPU-specific models

A small number of parallel models for multi-core CPUs are commonly employed. These stem from the time prior to the emergence of GPU accelerators and are not portable.

Pthreads

POSIX-threads, or short *Pthreads* [111] is a very early programming paradigm for multi-threading on CPUs. The compiler support is wide, but setting up the thread-scheduling in an application is a bit more involved. In this model, individual work packages are dispatched to a queue, and subsequently scheduled to free hardware threads for execution. As this procedure adds very little overhead, it yields a very flexible model for parallel execution of multiple heterogeneous work packages, and can achieve very good load balancing.

OpenMP

The traditional *OpenMP* specification, up to version 3.1 [91] only applied to multi-threading on CPU architectures. As of today, there is very good compiler support of its specification, both commercially and open-source. The parallel instructions are given with compiler directives or so called *pragmas*, which allows *C/C++* or *Fortran* code to be retrofitted for multi-threading. By default, this model operates a fork-join model, in which a group of threads are scheduled at the same time and the program flow only proceeds once all threads of this group have completed their work. In case of non-balanced workload over threads, this can compromise the utilisation of the hardware and hence performance.

Vector-level parallelism

To address the parallelism of the CPU, the programming models (for example *OpenMP* or *Pthreads*) schedule one thread per core (or two in case of hyper-threading). Vector level parallelism is much harder to be achieved; it can be done explicitly by using assembler-level intrinsic instructions or with a number of implicit options like compiler auto-vectorisation, by using Intel *Cilk Plus C/C++ Extensions for Array Notations* or the Intel *IPP* or *MKL* libraries [58]. Compiler auto-vectorisation has to be applied with caution, as compilers do not always find the best SIMD instructions. This applies especially for AVX512 (Intel vector lanes with 512bits) instructions, that induce a reduction of processor clock-rate when executed.

2.2.3 GPU-specific models

The models introduced in this section are targeting GPUs of selected vendors and are not portable to multi-core CPUs.

CUDA

To instruct a GPU to compute general non-graphics data, Nvidia supports the compute unified device architecture (*CUDA*) language [88] as a language extension to *C/C++*. *CUDA* and the corresponding *nvcc* compiler are restricted to Nvidia hardware and are propriety, but royalty-free products. Consequently, when employing *CUDA* for execution on Nvidia GPUs, additional code-bases for other architectures would need to be maintained. Furthermore, computational kernels need to be defined at quite a low level and allow careful GPU memory management in order to be efficient.

To optimise data placement on memory automatically, the *PORPLE* framework [75] has been proposed, which is based on *CUDA*. The code achieved good performance, however, the authors did not make their proof-of-concept code available.

HIP

The *C++ Heterogeneous-Compute Interface for Portability* [40], or short *HIP*, is a very recent open standard programming model promoted by AMD. Its syntax is very similar to *CUDA*, but it can be compiled for GPUs of both predominant vendors, for AMD using the *hcc* compiler, and for Nvidia using the *nvcc* compiler. It can be considered the first programming API for AMD GPUs that offers performance and usability, contrary to the previous option of using *OpenCL*. Porting of existing *CUDA* code to *HIP* is straightforward, and the performance of *HIP* code compared with *CUDA* code on Nvidia GPUs is on par in early investigations of sparse matrix solvers.

2.2.4 Portable models

A number of programming models that are portable between CPU and GPU architectures, or other future accelerators are presented henceforth.

OpenCL and related models

A very general parallel programming languages is *OpenCL* (Open Computing Language [67]). It is a framework for programs that can be executed across heterogeneous platforms including multi-core CPUs, GPUs from Nvidia and AMD, co-processors and FPGAs. *OpenCL* is a language extension to *C* and requires kernels to be defined at a low level and tailored and tuned towards the individual architectures. Thus, while in theory this framework is portable, it is not *performance portable* and would require multiple architecture-kernels to be maintained, compromising its usability.

As an intermediate layer on top of *OpenCL*, Khronos has developed the API (application programming interface) *SPIR* (Standard Portable Intermediate Representation) [68], which allows developers to maintain one language front-end for multiple accelerator architectures. Additionally, they developed another abstraction layer called *SYCL* [69] that supports *C++* language features for host and device code. Again, only one source code would need to be maintained that can be compiled to the *OpenCL* back-end and hence execute on all *OpenCL* supported architectures, including Nvidia and AMD GPUs. The entire *SYCL* and *OpenCL* ecosystem is open standard and aims to conform with the upcoming *C++17* standard. At the time of starting this thesis, there has been little experience with *SYCL* within the HPC community. The performance of *SYCL*, given the *OpenCL* back-end that generally needs lots of fine-tuning, will require some careful evaluation. Early studies in references [113, 10] present benchmarks of *SYCL* implementations that have been run on different parallel architectures and show very poor performance compared with native *OpenCL* or *OpenMP*.

Recently, however, Intel has released a beta version of its *Intel oneAPI* [59], that is heavily based on *SYCL*, and additionally provides debugging, profiling, and integrated development engines (IDEs) infrastructure.

Some academic proof-of-concept studies have been presented, for example in reference [44] with an *OpenMP* to *OpenCL* compiler, that achieves very good performance on Nvidia and AMD GPUs. The compiler automatically determines which parts of the program are worth offloading to the GPU and which are executed on a multi-core CPU using *OpenMP*. They achieve superior performance over a hand-coded expert *OpenCL* implementation using loop interchanges, index reordering of arrays and promoting and pre-fetching suitable variables to shared or private memory.

Further, the *PENCIL* model [7, 6] has been proposed: a *C*-based language extension with a compiler backend that translates directives inspired by *OpenMP* to *OpenCL*. They achieve performance portability, but the project has been cancelled at this point in time.

Performance comparisons between *CUDA* and *OpenCL*

Even though AMD GPUs are not as widespread, they still have the potential to be competitive with Nvidia GPUs. The AMD Radeon7970 and the Nvidia GTX580 are compared with the full NAS benchmark suite using double precision in reference [44]. For both devices optimised *OpenCL* kernels are used and the AMD GPU is at least *on par* for all tests. It can be considered a fair comparison, as the difference in acquisition costs and electricity consumption of the two GPUs are within 10-15%

Contrary to objections within the HPC community, the *OpenCL* API is actually capable of achieving comparable performance with *CUDA* on GPUs [31, 70, 99] and with *OpenMP* on CPUs [102, 73, 112].

However, this requires very careful tuning to address the specifics of the different hardware architectures. Yet, for maintainability, it is not feasible to rely on these procedures. Instead, linear algebra libraries and higher level abstractions would be required.

Obvious choices to be integrated into a *OpenCL* code would be the *clBLAS* [39] or the *clMAGMA* [16] libraries. GEMM implementations of these two libraries on Nvidia GPUs, however, yield only about 50% of the performance compared with *cuBLAS* [99]. Interestingly, reference [99] also shows that it is actually possible to achieve equally performant *OpenCL* kernels on Nvidia GPUs. Still this only applies to cases where Nvidia released support for all hardware features to *OpenCL*, which is only the case up to *OpenCL* version 1.2. It follows that currently, on Nvidia's Kepler or newer GPU architectures, the performance of *OpenCL* kernels is inherently limited. Reference [99] also shows that on AMD GPUs, the *clBLAS* or the *clMAGMA* libraries perform very poorly compared to their auto-tuned *OpenCL* kernels. So even while AMD GPUs show very good hardware characteristics, these observations can explain that their usability has been severely limited by the lack of an accessible and performant programming model. This could change from now on with the release of the *HIP* programming model.

OpenACC and related models

Another portable model is *OpenACC* [90], which works with compiler directives in the spirit of *OpenMP* to offload specific parts of the program from CPUs to GPGPUs or co-processors. The use of compiler directives allows porting legacy code incrementally to an accelerator. However these can be ambiguous, giving the developer less direct influence on how to parallelise and distribute the workload. Even though the specifications should address all accelerators, the compiler support for Nvidia GPUs is far more developed than for AMD GPUs. Multiple compilers offer a host-fallback compilation target, that can yield multi-threaded code for CPU executions, too. Numerous applications of *OpenACC* can be found in the literature, but no *thorough* performance comparison to *OpenMP*, or *OpenCL* has been identified. In reference [96] the applicability of *OpenACC* is investigated and a satisfactory performance is found, but the authors felt limited by the alterations needed for coalesced memory access. *OpenACC* is portable across multi- and many-core architectures and achieves good performance compared to a native *CUDA* implementation in references [74, 49], but only around 50% of the *CUDA* performance in reference [56]. *OpenACC* data layouts are fixed, thus a contiguous or coalescent memory access cannot be guaranteed for all architectures at the same time.

To overcome this obstacle towards performance portability, an *OpenACC* extension [55] is proposed, that supports a polymorphic data layout using the *Rose* [42] source-to-source translator. However, at this state their implementation is only a proof-of-concept and not publicly available.

OpenMP and related models

Starting from version 4.0 [92], the *OpenMP* specification supports truly heterogeneous computing. Additionally to multi-threading on the CPU, which acts as the host processor, offloading parallel instructions to a device, for example a GPU, can be specified. This feature would make this model even more portable than *OpenACC*, which can only specify the parallelism either on the device, or on the host. At the time of writing this thesis, open-source compiler support has been very limited, though. More details are discussed and explored in Chapter 4.

Kokkos

Kokkos [17] is a library for *C++11*, that supports different parallel architectures, while maintaining only one code base. The supported back-ends are *OpenMP* and *Pthreads* [111] for multi-threading on CPUs and accelerators like Intel *Xeon Phi's*, as well as *CUDA* to support parallel execution on Nvidia GPU's. AMD GPUs are not yet supported. This scheme is illustrated in Figure 2.7.

Kokkos addresses data parallelism, but so far no task parallelism with task scheduling algorithms similar to *MPI*, *Pthreads* or *Qthreads* [120]. The programming model is part of the *Trilinos* HPC library [51] developed by Sandia National Laboratories, which is for

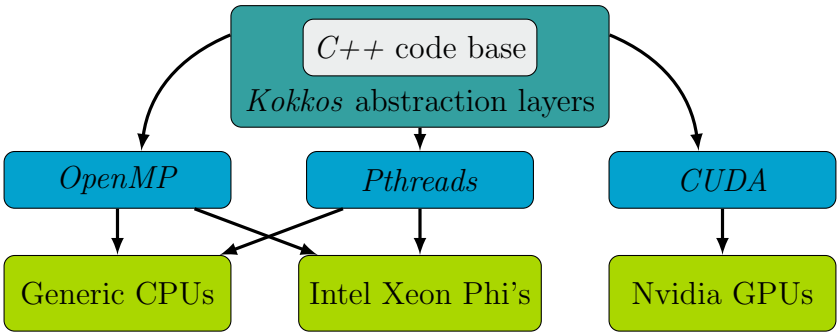

Figure 2.7: Implementing the *Kokkos* programming model for multiple backends.

example implemented in *LAMMPS* [97], a major molecular dynamics code. The syntax of *Kokkos* is general and architecture independent, but the back-end still leverages architecture specific features, like the utilisation of texture or shared memory on GPUs. Up to three levels of hierarchical parallelism can be specified. A defining feature of this paradigm is its variable data layout for multidimensional arrays, that ensures contiguous or coalesced memory access, respectively, for all architectures. Currently, the *Kokkos* community also started to incorporate BLAS libraries at different levels of the nested parallelism.

Kokkos has started to gain traction within the HPC community, and a continued support for the next couple of years is expected. In reference [17], a continuous-Galerkin finite-element mini-application is assessed using *Kokkos* against native *OpenMP* and *CUDA* implementations. The *Kokkos* variant with *CUDA* back-end was found to run about 13% faster than the native *CUDA* implementation on a Nvidia Kepler architecture. The *Kokkos-OpenMP* variant was about 10% slower than a native *OpenMP* implementation on the Xeon Phi architecture. Other scientific applications of Sandia National Laboratories have been implemented in *Kokkos* in references [57, 47, 110]. The authors ported serial or *MPI* legacy-code to *Kokkos* and concluded that it is a suitable and performance portable paradigm.

Expressed as an application flow diagram shown in Figure 2.8, the default CPU program flow is shown on the left hand side. The host kernel will be characterised by a multi-threaded execution. For the minimal GPU program flow, the right-hand side elements are required, that is data copying to the device and back to the host, as well as the device kernel, that executes the parallel instructions on the device or accelerator. Host and device kernel are both based on the same code-base, only compiled with different system targets.

Other portable abstraction libraries

Other similar models based on libraries with multiple backends are *RAJA* [54], developed by Lawrance Livermore National Labs and *OCCA* [79], developed at Virginia Polytechnic Institute and State University. Both libraries have been found in a less mature state than *Kokkos*, though.

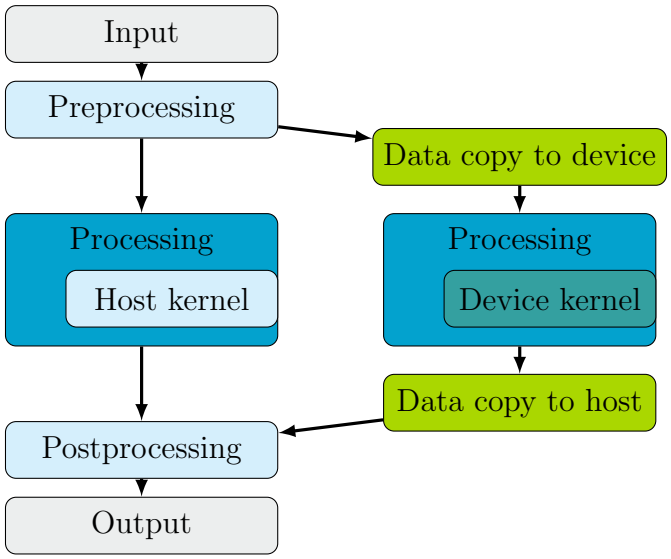

Figure 2.8: A simple program flow diagram using the *Kokkos* programming model.

At the Massachusetts Institute of Technology, *Julia*, a completely new programming language is developed, as presented in reference [11]. It however only gained more traction in the community towards the very end of this thesis, otherwise it would have been a promising candidate to explore in more detail. This language is extremely flexible, it can be used as an interpreted as well as a compiled language. It works without using type definitions, but they can be added later to yield more performance. It contains interfaces for many common parallel libraries like *MPI*, *clBLAS*, or *cuBLAS*, but there is also active development for a complete parallel abstraction layer in the spirit of a PGAS model.

To offer a higher level programming approach for GPU executions, Nvidia has developed the open-source *Thrust* library [89], based on the *Standard Template Library* (STL) functionality. For GPU only execution, it is interoperable with *CUDA*, but for a fallback CPU execution it can also be compiled for *OpenMP* or Intel's *Threaded Building Blocks* (TBB) library. The *Thrust* library though, has not been developed with nested parallelism in mind and could only be realised by specifying so called functors, leading to a very convoluted syntax.

For the ISO *C++17* standard, different libraries have pitched their implementations to be made the standard of a parallel template library [106]. These are for example a thin-wrapper around the Nvidia *Thrust* library, the *SYCL*-library from Khronos, a parallel STL based on *C++ AMP* [80] algorithms, and others.

The European Union project *PEPPHER* [66] proposed a programming framework for performance portable computing on heterogeneous multi-core architectures. The two main elements of this framework are the *StarPU* [5] run-time system, which is a *C* language extension and the *SkePU* [30] parallel skeleton library, with the interface of a *C++* template library. *SkePU* supports *CUDA*, *OpenCL* and *OpenMP* back-ends. A salient feature of both *SkePU* and *StarPU* is their integrated *MPI* support, allowing for seamless execution

on multi-node clusters. The successor *SkePU2* is under active development and seems a promising programming model, however, the transfer of legacy code to the skeleton templates is deemed rather cumbersome. Further, nested parallelism would have to be realised by passing functors.

Multi-device support on a single node

One aspect that is worth considering is the support of multiple GPGPUs on a single node. Different models have been developed, but it still remains a challenging domain. Using only a single programming model, an extension of *OpenMP* directives is proposed in reference [126], or similarly an extension of *OpenACC* directives in reference [48]. Alternatively, combinations of two shared parallel programming models are proposed in reference [125], using *OpenMP* to schedule one host thread per device and *OpenACC* for the parallelism on the device. Being an *MPI + X* model, the *MPI + OpenACC* model will also work on multi-node heterogeneous systems. Further, similar hybrid *MPI+X* models, combining one *MPI* rank per device with various accelerator-addressing programming models like *CUDA*, *Kokkos* or *OpenCL* are expected to work.

2.2.5 Architecting future-proof codes

Three main strategies can be considered to support a heterogeneous parallel hardware landscape.

First, if code maintainability is the highest priority, the codebase should be ported step-by-step to a performance portable programming model. This should start with the computational hotspots to quickly realise improved performance and support for a range of hardware systems. This strategy is followed in Chapters 3 and 4. As discussed before, there are numerous candidates for these kinds of programming models, indicating a developed problem awareness in the community and a demand to port legacy software to heterogeneous parallel systems. However, it is expected that this software landscape will consolidate, so that it is beneficial to identify the most promising candidates early on.

Second, if on the other hand performance is the highest priority, it is probably necessary to develop multiple codebases of highly optimised code for a range of hardware systems that should be supported. To still avoid code duplication as much as possible, it would suffice to only have specialised code for the compute intense kernels, and ignore for example pre- and post-processing routines. Currently it would be sufficient to maintain two codebases only, considering the prevalent HPC hardware systems. The first one for multi-core CPUs and Intel's Xeon Phi co-processors, that could be based on *MPI* or *OpenMP* plus an efficient model to address the vectorisation-level parallelism. The second code base would target the GPU architectures, using *CUDA* for the currently dominant Nvidia GPUs, or resorting to the recent development of the *HIP* model, that supports both Nvidia and AMD GPUs. Every few years, it might be necessary to add a new backend of the compute intense kernels to the code-base. We have been following this approach in Chapter 5,

where we develop optimised *CUDA* kernels.

A third alternative that has been adopted by multiple groups is to design a new framework from scratch that comprises enough flexibility to address a broad range of current and future architectures. Examples of such high-order codes include *PyFR* [122] for flux reconstruction methods, *Firedrake* [98] for FEM methods, and *OpenSBLI* [60] for finite difference methods. These frameworks consist of a high-level interface to specify the physical system that is to be solved, with an abstraction layer underneath this which translates high-level syntax to various low-level languages that can target different backends. Using a set of optimised BLAS libraries or bespoke kernels for each type of architecture, alongside good auto-tuning algorithms, can yield very competitive performance and flexibility at the same time.

The use of object-orientation programming with all these models, however, must be done with care: while for example seamless integration of *CUDA* kernels into *C++* code is possible, there are restrictions on what can be done within a kernel in terms of data management, which could otherwise have been conveniently hidden with object-oriented features. Most notably are the limitations to offload instances of classes (for example individual mesh elements of different types) to the GPU and iterate over all instances and implicitly call their member functions and data, especially if they stem from class inheritance and virtual functions.

Chapter 3

Accelerating a variational mesh optimiser using a portable programming model

In this chapter, we investigate the suitability of the *Kokkos* programming model for improving the performance of a mesh optimisation method within the high-order mesh generator *NekMesh*. The method minimises a deformation energy to distribute the elemental nodes optimally over the domain. All nodes are assigned to multiple coloursets, where nodes of the same colour can be processed concurrently, thus enabling a first level of parallelism. We discuss the method's adaptation and implementation for modern parallel shared-memory CPU and GPU platforms using *Kokkos*, presenting the first scientific contribution for GPU-based mesh-optimisation algorithms. We develop an element and node indexing strategy that is not object-oriented and associative, but works with plain data structures that are required for the underlying *CUDA* programming model. Additionally to the outer parallelism level over to be optimised mesh nodes, we further introduce an inner level of parallelism over the quadrature points of the elements. We show increased performance on CPUs using the *Kokkos* model with plain data structures compared with a native *Pthreads* implementation with associative data structures, and further speed-ups and cost reduction by running on GPUs without any hardware-specific code optimisation.

The work associated with this chapter has been published in references [26, 27].

This chapter is organised as follows: Section 3.1 gives a brief motivation and the objetives for this study. Section 3.2 outlines the formulation of the variational framework introduced in reference [115] and describes how the method can be effectively parallelised. Section 3.3 introduces *Kokkos* and the practical steps taken to accelerate the optimisation method. In Section 3.4, we then perform a thorough performance analysis of the optimised code to demonstrate the effectiveness of the strategy on a variety of architectures, including a standard multicore CPU system, a manycore CPU and various GPU platforms. Further, a novel cost metric to compare different types of architectures is introduced. In Section 3.5, we conclude with a discussion of the results and the wider impact of the work.

3.1 Introduction

The execution of finite element methods requires the discretisation of the domain into multiple elements with no overlap and no gaps. This tesselation of the domain is generally described as a *mesh*. High-order meshes allow the accurate representation of curved CAD (computer-aided design) boundaries, contrary to linear meshes that always introduce a spatial discretisation error on curved boundaries. For finely resolved flows this error can lead to undesired artefacts in the flow solution field, such as spurious pressure waves and the generation of spurious entropy. Especially for turbulent flow that is modelled with high-fidelity turbulence models this will introduce unphysical effects into aeroacoustic or transitional flows. Additionally, a coarser mesh with high-order shape functions leads to more accurate solutions than a fine linear mesh with the same number of degrees of freedom. The better error-convergence obtained with high-order meshes also improves the effectiveness of higher order temporal discretisation methods.

A common way to obtain high-order meshes is to deform an initial coarse linear mesh, which can be obtained using one of many available linear meshing tools, to conform with the curved boundary specified by the CAD geometry. This process will likely yield very distorted or inverted elements close to the boundary, as the introduction of curvature into the element frequently leads to self-intersection. Therefore a second step is required, that corrects invalid elements through a boundary-induced mesh deformation, so that curvature is introduced into elements connected and in close proximity to the curved surface. Several different techniques for this step have been proposed in the literature, which can be broadly classified into two categories: *elastic analogies* where the mesh is treated as a solid body and the curvature acts as a force on the body, e.g. [124, 34, 95], and *energy minimisation techniques* in which a functional representing mesh distortion is minimised to optimise mesh quality and correct invalid elements, e.g. [103, 37].

These techniques in general are computationally expensive, since they require either the solution of a partial differential equation or a non-linear optimisation to obtain the corrected mesh. In an industrial setting, where geometries are typically extremely complex and meshes can consist of millions or billions of elements, this process can be computationally prohibitive. Modern design lifecycles also demand the generation of meshes in the order of minutes or hours, typically on only a single high-performance workstation.

Apart from generating and optimising initial meshes, the optimiser can also be employed for other mesh adaptation purposes. In reference [77], it is combined with a pressure sensor to perform r-adaptation of high-order meshes to achieve better shock-capturing in compressible flows, without changing the topology of the mesh. It could also be employed in simulations of moving geometries, for example in aero-elastic settings to dynamically deform the mesh around the wing. For these applications, it is especially desirable to have an efficient and parallel mesh-optimisation implementation available.

Although modern high-performance computing systems are providing more computational power than ever seen before, this potential has yet to be realised in the area of

high-order mesh generation. We do however note that in the area of linear mesh generation, first implementations for increased efficiency that run on general purpose GPUs have been started to be presented, as seen for example in reference [57].

The purpose of this work is to investigate how the variational framework introduced by Turner, Peiró and Moxey in reference [115], which encompasses many of the *a posteriori* techniques introduced above, can effectively make use of modern heterogeneous architectures. While the acceleration of this specific algorithm is a commendable task in its own right, this study shall also serve as an example on steps and pitfalls associated with porting a large HPC software framework and the improvements that can be realised.

To achieve this in a sustainable manner that does not require the maintenance of a codebase for each architecture, we make use of the performance portable programming model *Kokkos* [17], which is one of the models that have been introduced to overcome the inherent difficulties of architecting programs for various different computing systems. The use of this model allows us to use a single codebase to support a range of vastly different modern hardware, including 'traditional' multicore CPUs (e.g. Intel Xeon CPUs), manycore coprocessors (e.g. Intel Xeon Phi processors) and general purpose GPUs (e.g. Nvidia GPUs).

This chapter shall make a number of contributions. The most significant in terms of its application area, namely high-order mesh generation, is the demonstration of a methodology that is capable of effectively using manycore CPU and GPU hardware to reduce mesh optimisation runtimes. To the best of our knowledge, this is the first time that this is presented in the literature, particularly in the context of GPUs.

We conduct extensive performance tests on a variety of parallel hardwares, and examine scalability, as well as hardware utilisation in terms of achieved FLOP rates, achieved memory bandwidth, and warp efficiency. We also present a novel approach to model operating costs of different hardwares, which serves as a path to answer the question which hardware systems are ultimately more suitable for the investigated method. Additionally, through the discussion of the techniques used to accelerate our code, we also provide valuable insight into the steps required to port existing code to new architectures. Although in this case we are using the *Kokkos* model, the experiences described here are relatively broad and extend past the specific choice of programming model being used. Amongst other aspects, we discuss to what degree object-oriented data structures need to be rewritten, in order to comply with any current GPU-based programming model. This work is therefore valuable in the context of existing HPC users who intend to port their code to new architectures and who aim to promote sustainability of their software.

3.2 Method

We start with a brief overview of the variational framework outlined in reference [115]. The motivation for the framework is to reform partial differential equations (PDEs) arising from *a posteriori* techniques based on solid body analogies into an energy minimisation

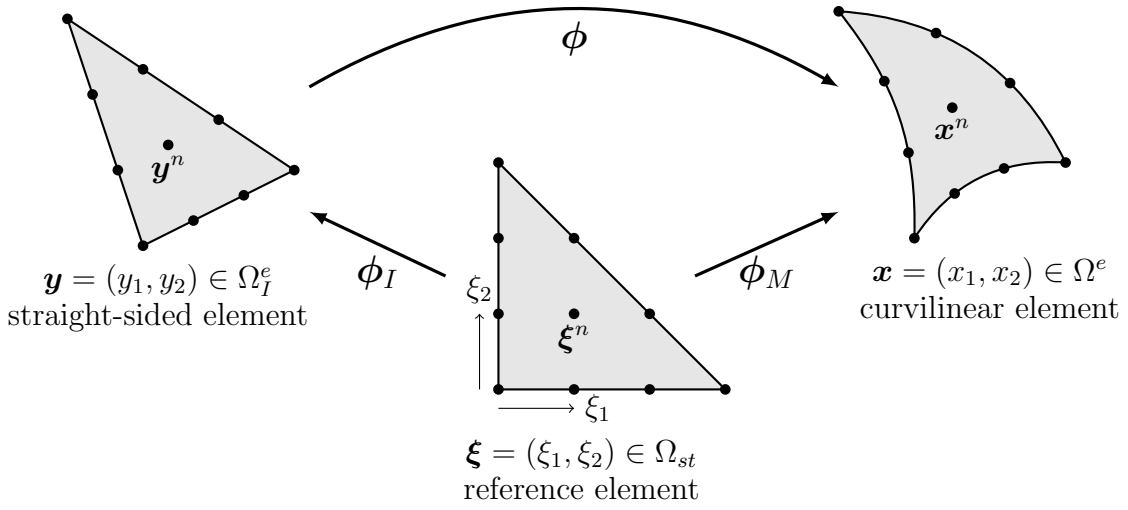

Figure 3.1: Mapping relations between the reference, straight-sided, and curvilinear elements

problem through the calculus of variations. In this manner, we are able to utilise a range of optimisation approaches from the literature under a single framework by minimising a functional $\mathcal{E}$. The techniques outlined below are implemented within the meshing tool *NekMesh* [114], which is part of the spectral/*hp* element framework *Nektar++* [14].

A general overview of the process is as follows. As a first step of the mesh generation, we obtain a linear mesh of a given CAD geometry using either the internal *NekMesh* mesh generation tools, or from external programs such as *Gmsh* [38]. We then apply the variational framework inside *NekMesh* in order to transform the linear mesh into a high-order mesh, correct invalid elements and generally improve the mesh quality. The sections below outline the mathematical background of the framework and describe the parallelisation strategy, which has been used in reference [115] to demonstrate good performance and robustness in the processing of meshes of the order of 10 million degrees of freedom.

3.2.1 Numerical evaluation of the energy functional

The definition of the energy functional to be minimised relies on a mesh deformation tensor $\nabla\boldsymbol{\phi}$, where $\boldsymbol{\phi}$ is a mapping between an 'ideal' straight-sided mesh Ω_I and the boundary-conforming curvilinear mesh Ω, as shown in Figure 3.1. We then write $\boldsymbol{\phi} : \Omega_I \to \Omega$, so that the Cartesian coordinates $\boldsymbol{x} \in \Omega$ are related to the ideal coordinates $\boldsymbol{y}$ through the relationship $\boldsymbol{x} = \boldsymbol{\phi}(\boldsymbol{y})$. The energy functional can then be defined as the integral

$$\mathcal{E}(\nabla\boldsymbol{\phi}(\boldsymbol{y})) = \int_{\Omega_I} W(\nabla\boldsymbol{\phi}(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}, \tag{3.1}$$

where the strain energy function W will depend on the chosen material constitutive model. Currently supported choices include models of linear elasticity [124], isotropic hyper-elasticity [95], the Winslow equations [34], and a distortion-measure energy [37].

We note that the implementations of the four models resulted in identically structured algorithms with similar performance. In the following, we focus on the hyper-elastic strain energy function for a compressible neo-hookean material [13], which resulted in the best quality meshes between the four models in testcases presented in reference [116]. For the selected model, W is given by

$$W = \frac{\mu}{2}(I_1^{\boldsymbol{C}} - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2, \tag{3.2}$$

where $J = \det \nabla \boldsymbol{\phi}$ is the volumetric deformation or Jacobian. For physically admissible deformations without penetration of matter, J has to be positive. λ and μ are the Lamé constants, which we write in terms of the Young's modulus E and Poisson's ratio ν, so that $\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$ and $\mu = \frac{E}{2(1+\nu)}$. Using this formulation it becomes clear that Young's modulus is just a scaling factor and the energy functional only depends on Poisson's ratio, which we set to $\nu = 0.45$. $I_1^{\boldsymbol{C}}$ is the first invariant of the right Cauchy Green tensor, which in our context is given by $I_1^{\boldsymbol{C}} = \mathrm{tr}(\boldsymbol{C}) = \mathrm{tr}(\nabla \boldsymbol{\phi}^T \nabla \boldsymbol{\phi})$.

The strain energy functions W have a singularity at $J = 0$, so that W asymptotically tends towards infinity. This is a desirable property if the mesh is valid, i.e. all elements fulfil $J > 0$, as it prevents the mesh from tangling, however this is undesirable if the mesh is initially invalid, as it prevents untangling. As is common in these approaches for both linear and high-order meshes, e.g. in references [35, 36], we apply a regularisation to the volumetric deformation J, that ensures the regularised version J_R remains positive and very small. It follows that the deformation function W no longer exhibits a singularity, and becomes very large for invalid and near-invalid elements driving the optimisation away from such configurations. The proposed regularisation is given as

$$J_R = \frac{1}{2}\left(J + \sqrt{4\delta^2 + J^2}\right). \tag{3.3}$$

Following reference [35] δ is a small number, which is set to

$$\delta = \begin{cases} \sqrt{\epsilon^2 + 0.04(J_{\min})^2}, & \text{if } J_{\min} < 0 \\ \epsilon, & \text{otherwise} \end{cases}, \tag{3.4}$$

where $J_{\min}$ is the minimum Jacobian of all mesh elements, and ϵ is a sufficiently small number, with $\epsilon << J_{\min}$. In this work we choose $\epsilon = 10^{-4}$, which results in a good compromise between robustness and convergence rate for our mesh cases. We note that for general cases smaller values closer to the machine epsilon might be required.

Up to this point, the outline of the method refers to a whole mesh. We now focus on the implementation of optimising the energy functional on a mesh consisting of multiple elements. The energy functional of the whole mesh $\mathcal{E}$ is accordingly calculated as a sum

of the elemental contributions

$$\mathcal{E}(\nabla\boldsymbol{\phi}(\boldsymbol{y})) = \sum_{e=1}^{N_{el}} \int_{\Omega_I^e} W(\nabla\boldsymbol{\phi}(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}, \tag{3.5}$$

with Ω_I^e being an initial undeformed straight-sided element. Finite element shape functions within *NekMesh* are evaluated on a reference element Ω_{st} with coordinates $\boldsymbol{\xi} \in \Omega_{st}$. Hence, another mapping between the reference element Ω_{st} and the straight-sided element Ω_I^e can be defined as $\boldsymbol{\phi}_I : \Omega_{st} \to \Omega_I^e$, as shown in Figure 3.1. Applying the coordinate transformation, the energy functional becomes

$$\mathcal{E}(\nabla\boldsymbol{\phi}) = \sum_{e=1}^{N_{el}} \int_{\Omega_{st}} W\left[\nabla\boldsymbol{\phi}_M(\boldsymbol{\xi})\nabla\boldsymbol{\phi}_I^{-1}(\boldsymbol{\phi}_I(\boldsymbol{\xi}))\right] \det(\nabla\boldsymbol{\phi}_I)\,\mathrm{d}\boldsymbol{\xi}\,. \tag{3.6}$$

The ideal mapping $\boldsymbol{\phi}_I$ is affine and can be written analytically by combining linear finite element shape functions. It is independent of $\boldsymbol{\xi}$ for tetrahedral elements, so it is computed only once per element and stored. The curvilinear mapping $\boldsymbol{\phi}_M$ is an isoparametric mapping for nodal high-order element discretisations, given by

$$\boldsymbol{\phi}_M(\boldsymbol{\xi}) = \sum_{n=1}^{N} \boldsymbol{x}^n \boldsymbol{\ell}_n(\boldsymbol{\xi}), \tag{3.7}$$

where N is the number of nodes of the element and $\boldsymbol{\ell}_n$ are the Lagrange polynomial interpolants, which conform with $\boldsymbol{\ell}_m(\boldsymbol{\xi}^n) = \delta_{nm}$.

The above equality relies on the definition of a set of points $\boldsymbol{\xi}^n \in \Omega_{\mathrm{st}}$ that define the isoparametric mapping, for which we use a 3D nodal basis on each tetrahedral element, also known as Hesthaven or α-optimised points [52]. We note that the evaluation of the functional itself, as in Equation 3.6, is performed using a different set of points conforming to a quadrature rule proposed by Witherden and Vincent [123]. This quadrature rule has positive weights at all evaluation points thus increasing the robustness of the optimisation procedure. This is advantageous because the use of quadrature rules with negative weights close to the vertices of the element leads to unrealistic displacements of these nodes during the optimisation. This is discussed further in reference [115].

3.2.2 Parallelisation and node-colouring

Instead of optimising the node positions in a global approach, we apply a relaxation method, that solves a set of local optimisations. This is possible, since individual nodes only affect the energy functional of elements they are connected with. A node that is interior to one element, will only affect the energy contribution of that one element; inter-elemental boundary nodes will affect all elements they are connected with, which in case of tetrahedra is two for face-nodes, around 4 to 10 for edge-nodes and around 14 to 50 for vertex nodes, depending on the connectivity of the mesh. Figure 3.2 illustrates

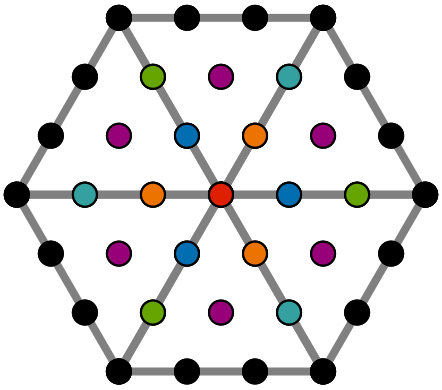

Figure 3.2: Node colouring scheme for a domain of six triangular elements. Nodes of the same colour can be processed concurrently.

this concept: nodes of the same colour can be processed concurrently as the respective operations involved in the evaluation of the energy functional are independent.

The functional for the optimisation of a free-to-move node i towards lower energy functionals becomes

$$\mathcal{E}_i(\nabla\boldsymbol{\phi}) = \sum_{e \ni i} \int_{\Omega_e} W(\nabla\boldsymbol{\phi})\,\mathrm{d}\boldsymbol{y}\,, \tag{3.8}$$

where $e \ni i$ denotes the set of all elements that own node i. This set of elements hence spans all parts of the mesh that influence the energy functional of node i.

This local approach has the disadvantage that the optimisation might get stuck in local extrema. It is hence the challenge to employ an optimisation strategy that has a high degree of success in finding the global minimum. The main advantage of this strategy, however, is the potential for parallelisation, as each node typically only affects a small fraction of elements of the total mesh. Using node colouring, the mesh is split into independent sets that can be processed in parallel. The splitting is subsequently repeated until all free-to-move nodes have been processed once. This makes the strategy very well suited for massively parallel hardware like GPUs.

We note that employing a relaxation method is sufficient for this mesh-optimisation problem, as we are only interested in smoothing local distortions in the mesh, typically close to the CAD boundary. As we base our approach on linear meshes with a good distribution of elements, the utilisation of a multigrid method is not necessary to smooth any long-range distortions of the mesh. It might even be counter-productive for meshes where element sizes intentionally vary by large ratios.

3.2.3 Optimisation strategy

The minimisation of the energy functional is performed with a Newton method with truncated steps. This method requires the evaluation of the gradient vector $\boldsymbol{G}$ and the Hessian matrix $\boldsymbol{H}$ of the energy functional $\mathcal{E}$, where the derivatives are calculated with respect to the position of the node undergoing optimisation. These are calculated analytically using

the formulae in Appendix B of reference [115]. The coordinates of the free-to-move nodes are then updated in the optimisation step k according to

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^{k} - \alpha \boldsymbol{H}^{-1}\boldsymbol{G}, \tag{3.9}$$

with α being a step size parameter with $0 < \alpha \leq 1$. A reverse line search using the Wolfe condition is applied to find a value of α that guarantees the node moves towards a smaller energy level. Each free-to-move node is moved every optimisation step (unless no smaller energy level can be found) and the optimisation steps are repeated until a convergence criteria is fulfilled. In addition to the Jacobian regularisation discussed previously, a regularisation is applied to badly conditioned Hessians, to restore the symmetric positive-definite property in the presence of invalid or highly distorted elements, and hence increase the robustness of the method.

As the termination criterion of the global optimisation, the infinity norm between the node positions before and after the latest optimisation step is considered. If the norm is smaller than a small fraction of a characteristic elemental length L, in this work typically chosen to be $\epsilon_{conv} = 10^{-6}L$, the optimisation is terminated.

3.2.4 The complete algorithm

Algorithm 3.1 illustrates how the various evaluations are combined to yield the complete algorithm. The procedure EvaluateFunctional is further broken down in Section 3.3, where the crucial modifications that have been introduced in order to enable hardware portability are explained.

It is noted that instead of using a Gauss-Seidel-like relaxation method, a Jacobi-like relaxation method could be employed. Such a method would not require a node-colouring approach and would hence be more naturally parallelisable. It would however incur the costs of duplicating the storage space for all node coordinates and show worse convergence properties. We found that the sizes of each colourset are large enough to provide sufficient parallelism to fully occupy all CPU cores or GPU streaming multiprocessors, even for relatively small meshes. Hence we follow the Gauss-Seidel-like relaxation method here.

3.3 Accelerating the method

This section discusses the acceleration of the variational framework outlined in the previous section using the architecture independent programming model *Kokkos*. We begin with a brief overview of the characteristics of performance portability that we wish to exploit in this work, and discuss some of the potential choices of programming models, as well as our motivation for using the *Kokkos* framework. We then outline the computational work required to port the initial version of the variational framework within *NekMesh* to an efficient *Kokkos* implementation.

Algorithm 3.1 Variational mesh optimisation

```
1: procedure GlobalOptimise(N)          ▷ N is the set of all nodes to be optimised
2:     C ← ColourNodes(N)                          ▷ divide nodes into coloursets C
3:     call CopyData(N)  ▷ deep copy from host to device memory when using GPUs
4:     while ‖x^{k+1} − x^k‖_∞ > ε_conv do                ▷ using convergence criterion
5:         for all colour-sets c ∈ C do
6:             for all nodes n ∈ c do in parallel
7:                 E, G(E), H(E) ← EvaluateFunctional(x_n^k)
               ▷ evaluate functional, derivatives and Hessian, based on node coordinates x
8:                 while α > α_ε do                ▷ reverse line search, start with α = 1
9:                     x^temp ← x_n^k + α H(E)^{-1} G(E)   ▷ move node in the descent direction
10:                    F ← EvaluateFunctional(x^temp)        ▷ evaluate functional only
11:                    if F ≤ E + f_Wolfe then                ▷ using Wolfe condition
12:                        x_n^{k+1} ← x^temp                          ▷ new minimum found
13:                        break
14:                    end if
15:                    α ← (1/2) α
16:                end while
17:                x_n^{k+1} ← x_n^k                    ▷ unable to optimise, reset node
18:            end for
19:        end for
20:    end while
21: end procedure
```

3.3.1 Choosing the programming model

As described in Section 2.1, the current coexistence and future uncertainty of different HPC hardware architectures benefit an easily maintainable code. This can be achieved by supporting a single codebase that allows portability across architectures instead of maintaining specific and separate implementations for each architecture. It is also very desirable to have a codebase that is *performance portable*, meaning it can achieve nearly optimal performance on all architectures.

We only consider shared memory systems in this chapter, since the use of a single workstation is the most likely scenario for the generation of meshes with a few million elements. However, for problems requiring the generation of extremely large numbers of elements in the mesh, the work here could be extended to a distributed memory model with *MPI* and using a domain decomposition approach.

Different options for portable programming models have been presented in Section 2.2: the low-level language extensions *OpenCL* [67], or high-level libraries and directives such as *OpenACC* [90], *SYCL* [69] or *Kokkos* [17]. There has been little experience with *SYCL* in the community, however a significant performance overhead compared to its only backend *OpenCL* has been reported. *OpenACC* is a very widespread heterogeneous programming model, but its compiler directives are somewhat ambiguous and do not guarantee a performance portable memory access pattern. At the time of conductiong the work of this chapter there has been no *OpenMP* compiler support for GPGPUs, yet.

Algorithm 3.2 Calculating the energy functional of a node with the initial implementation

```
1: procedure EvaluateFunctional_Initial(x_n)
2:     E, G(E), H(E) ← 0, 0, 0
3:     X ← x ∈ {e ∋ n}                    ▷ coordinates x of all elements e that own node n
4:     for all coordinate directions d do
5:         ∇φ_{M_d} ← D_d X_d             ▷ compute deformation tensor in direction d using dgemm
6:     end for
7:     for all elements e ∋ n do in serial          ▷ all elements e that own node n
8:         for all quadrature points i ∈ e do in serial
9:             w ← w_i det φ_I(ξ̃^i)               ▷ mapping determinant weighted by quadrature
10:            E ← E + W(∇φ_{M_i}) · w
11:            if calculating gradient and Hessian then
12:                for all coordinate directions j do
13:                    G(E)[j] ← G(E)[j] + ∂_{n_j} W(∇φ_{M_i}) · w        ▷ analytic gradient
14:                    for all coordinate directions k do
15:                        H(E)[j,k] ← H(E)[j,k] + ∂_{n_j n_k} W(∇φ_{M_i}) · w   ▷ analytic Hessian
16:                    end for
17:                end for
18:            end if
19:        end for
20:    end for
21: end procedure
```

We therefore adopted *Kokkos* (see Section 2.2.4) as our programming model because of the good performance it was deemed to attain, and its compatibility with the existing *NekMesh* codebase.

3.3.2 The initial *Pthreads* implementation

With a view to later discuss the development of the portable *Kokkos* implementation, we start by describing the initial implementation of the variational mesh optimisation algorithm.

Initial data structures

The initial CPU implementation of the variational meshing optimisation organises the data by using shared pointers to *C++* objects. This way the complex associations between the set of nodes and elements can be organised clearly and descriptively, ensuring good code maintainability. Each free-to-move node is an instance of the *node* class and each mesh element is an instance of the *element* class. The objects need to be shared because nodes on the surface of elements are part of all its neighbouring elements. Hence, a given *node* object has one or more *element* objects connected to it. At the same time, a given *element* object owns multiple free-to-move *node* objects and other fixed nodes to fill up the ranks. The mapping gradients, $\nabla\boldsymbol{\phi}_M$, for each node are further stored as member variables of each node instance.

Initial algorithm

All nodes within one colourset are processed in parallel, on the outer level of parallelism. Algorithm 3.1 illustrates how this process fits into the overall mesh optimisation algorithm. The parallelism is enabled by a lightweight dynamic thread-scheduler that is based on *Pthreads*.

Within the optimisation of each node, the energy functional at its initial position is evaluated first, along with its spatial derivatives and its Hessian. These values in turn depend on the contribution of all elements this node is connected with, as given in Equation 3.6 and illustrated in Algorithm 3.2.

Since the whole function EnergyFunctionalInitial is performed using a single thread only, the expensive calculation of the mapping $\nabla\phi_M$ is amalgamated using the level-3 BLAS call `dgemm`. The remaining steps of the function are calculated in a serial loop over all quadrature points of all elements that own the to be optimised node.

The code syntax of the initial implementation is provided in Listing 3.1.

3.3.3 Developing the *Kokkos* implementation

To make our application portable, the initial implementation is adapted to utilise the *Kokkos* programming model. Even though the codebase should be changed as little as possible having maintainability in mind, some changes have been required, primarily in order to comply with the new programming model, and secondarily to harvest the low-hanging performance benefits. While it is possible to use *Kokkos* for CPUs by only replacing the thread-scheduling, more fundamental design changes are required to achieve portability to GPUs because compliance with the *CUDA* backend of *Kokkos* needs to be ensured. This is due to the high complexity of achieving maximal performance on GPU architectures, compared with more established CPU architectures. These changes will in turn be beneficial for later CPU executions, too, because parallelism instructions on the vector lanes are expressed, and data access is simplified. This section continues to discuss the design changes in terms of re-factoring data structures and introducing hierarchical parallelism. A short example given in Listing 3.2 illustrates the changes introduced in the codebase to accommodate the new programming model.

Developing the *Kokkos* data structures

The *Kokkos* programming model with the *CUDA* backend requires all data and data dependencies to be expressed in plain arrays. It follows that the associative object-oriented data structures need to be re-factored. The resulting data structures and the rationale for its system is described hereafter.

All nodes of a given element need to be included for the elemental evaluations and processing steps. It has therefore been decided to store all nodal coordinates of one element contiguously, to achieve a more efficient memory access. However, the duplicated storage of boundary nodes requires increasing the memory by a factors of 4.2, 3.1 and 2.6

Listing 3.1: Managing of node coordinates and parallelism in the initial implementation

```
// shared pointer of node objects
vector<vector<boost::shared_ptr<NodeOpti>>> opti_nodes;
// loop over all colourseets in serial
for (int cs = 0; cs < opti_nodes.size(); cs++)
{
    // getting size of colourset
    const int nodes = opti_nodes[cs].size();
    // create Pthread jobs to optimise each node
    vector<Thread::ThreadJob *> jobs(nodes);
    for (int node = 0; node < nodes; node++)
    {
        jobs[node] = opti_nodes[cs][node]->Optimise();
    }
    // send of parallel jobs and wait for their completion
    QueueJobs(jobs);
    Wait();
}

// the initial node optimisation routine
void NodeOpti::Optimise()
{
    // getting the number of connected elements from a member
       variable of the node
    const int elmt = m_elmts.size();
    // loop over all connected elements in serial
    for (int el = 0; el < elmts; ++el)
    {
        // getting the node coordinates from the node member
           variables
        x = m_node->m_x;
        y = m_node->m_y;
        z = m_node->m_z;
        // loop over all quadrature points in serial
        for (int qp = 0; qp < m_utilities->n_qp; ++qp)
        {
                //calculations per quadrature point
        }
    }
}
```

Listing 3.2: Managing of node coordinates and parallelism in the full *Kokkos* implementation

```
// loop over all colourSets in serial
for (int cs = 0; cs < css; cs++)
{
    // getting size of colourset
    const int nodes = nodes_array(cs);
    // using Kokkos-syntax to create parallel teams for each node
    Kokkos::parallel_for (team_policy (nodes, Kokkos::AUTO)
        , KOKKOS_LAMBDA (const member_type& teamMember)
    {
        const int node = teamMember.league_rank();
        // getting the number of connected elements
        const int elmts = elmts_array(cs,node);
        // loop over all connected elements in serial
        for (int el = 0; el < elmts; ++el)
        {
            // getting the node indices
            const int elid = elid_array(cs,node,el);
            const int localnodeid = localnodeid_array(cs,node,el);
            // getting the node coordinates based on the indices
            x = X(elid, localnodeid);
            y = Y(elid, localnodeid);
            z = Z(elid, localnodeid);
            // using Kokkos-syntax to create parallel threads for
               each quadrature point
            Kokkos::parallel_for (Kokkos::TeamThreadRange (
               teamMember, qps)
                , [&] ( const int qp)
            {
                //calculations per quadrature point
            });
        }
    });
}
```

for tetrahedral meshes of orders 3, 4 and 5, respectively. Yet, even with this duplication of node data, the mapping $\phi_I : \Omega_{st} \to \Omega_I^e$ of each quadrature node occupies most of the memory space, typically by more than 90%.

Even though the coordinates of each free-to-move node are updated only once per optimisation step, the coordinate arrays of *all* elements that own the node need to be updated. The information for this update is provided by a first array containing the number of elements a node is connected with. Each node instance then requires the element ID, as well as the local-node ID determining the position of the node within its element. Listing 3.2 can be consulted for an illustration of this approach.

One main group of utility data for all elemental evaluations are the quadrature points and weights for the type and order of elements in the mesh. These are potentially accessed by multiple elemental operations at a time and hence stored in the fast read-only texture

memory. The second main group consists of the array describing the mapping of each node $\phi_I : \Omega_{st} \mapsto \Omega_I^e$ from the reference element to the straight-sided element. A 3×3 matrix and its determinant will be stored for each node on the global memory. The values are constant throughout the optimisation, as only the mapping ϕ_M between the straight-sided element and the curved element is altered. If the mesh is too big to be stored on the global memory of the GPU, this mapping function could be recalculated each time. This would free 10 out of 13 (the others being the three spatial node coordinates) double precision floating-point values stored with each node, at the rather small cost of re-evaluating the mapping each timestep.

Using the *Kokkos* programming model, all these arrays are implemented as *Kokkos-Views*. These are multidimensional arrays, that exhibit a polymorphic layout that is specified at compile time. The layout of a 2D array can be either column or row major, depending which memory access pattern is more performant on the underlying hardware.

Mapping the algorithmic parallelism to the *Kokkos-CUDA* model

Algorithm 3.3 presents the implementation developed for performance portability using the *Kokkos* programming model. A few changes compared to the initial Algorithm 3.2 have become necessary to achieve good performance. Just as in the initial algorithm, all nodes within one colourset are processed in parallel. Additionally though, we introduce a second nested level of parallelism over the processing of all quadrature points of an element. Due to algorithmic divergences of the optimisation method, it is not possible to have a very fine-grained parallelism at the node level. It is hence essential that a nested level of parallelism is specified that uses parallelism on the *CUDA*-thread level.

Within this algorithm, all elements connected with a particular node are still processed sequentially, even though they could be processed in parallel. It was found to be the faster option, unless different block sizes per node were to be defined depending on the number of connected elements, which is not possible with the current version of *Kokkos*.

The inner level of parallelism processes the individual quadrature points of one element using the *Kokkos::Thread* parallelism. The contribution of each quadrature point have to be added up, using the *Kokkos* options of either a parallel reduction or individual atomic operations. Since the number of quadrature points for typical elements is rather small, *Kokkos::atomic_add* operations are utilised. The alternative of storing all contributions and subsequently doing a *Kokkos::parallel_reduce* operation, however, was found to be slower due to its function overhead.

Another difference compared to the initial algorithm is the way $\nabla\phi_M$ is calculated. As multiple threads operate on the inner parallelism, the thread of each quadrature point calculates its corresponding part of the mapping. The calculation repeatedly involves the processing of all nodal coordinates of this element, which are hence loaded into shared memory for faster access using the relevant *Kokkos* functionality. Employing a BLAS or CuBLAS call within this inner level of parallelism for the small matrix-vector multiplica-

Algorithm 3.3 Calculating the energy functional of a node with the full *Kokkos* implementation

procedure EvaluateFunctional_Kokkos($\boldsymbol{x}_n$)
  $\mathcal{E}, \boldsymbol{G}(\mathcal{E}), \boldsymbol{H}(\mathcal{E}) \leftarrow 0, \boldsymbol{0}, \boldsymbol{0}$
  for all elements $e \ni n$ **do in serial** ▷ all elements e that own node n
    $\boldsymbol{X} \leftarrow \boldsymbol{x} \in e$ ▷ coordinates $\boldsymbol{x}$ of element e only, load to shared memory
    for all quadrature points $i \in e$ **do in parallel**
      $\nabla \boldsymbol{\phi}_{M_i} \leftarrow \mathcal{D}_i \boldsymbol{X}_i$
      ▷ compute deformation tensor for this element and this quadrature point only
      $w \leftarrow w_i \det \boldsymbol{\phi}_I(\tilde{\boldsymbol{\xi}}^i)$ ▷ mapping determinant weighted by quadrature
      $\mathcal{E} \leftarrow \mathcal{E} + W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$ ▷ use atomic operation to update $\mathcal{E}$
      if calculating gradient and Hessian **then**
        for all coordinate directions j **do**
          $\boldsymbol{G}(\mathcal{E})[j] \leftarrow \boldsymbol{G}(\mathcal{E})[j] + \partial_{n_j} W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$
            ▷ analytic gradient, use atomic operation to update $\boldsymbol{G}(\mathcal{E})$
          for all coordinate directions k **do**
            $\boldsymbol{H}(\mathcal{E})[j,k] \leftarrow \boldsymbol{H}(\mathcal{E})[j,k] + \partial_{n_j n_k} W(\nabla \boldsymbol{\phi}_{M_i}) \cdot w$
              ▷ analytic Hessian, use atomic operation to update $\boldsymbol{H}(\mathcal{E})$
          end for
        end for
      end if
    end for
  end for
end procedure

tions is not deemed to be beneficial, due to increased memory copying.

A major challenge for our overall algorithm is that every node might undergo a different path within the optimisation algorithm. As seen in Algorithm 3.1, the number of iterations required within the reverse line search can vary from node to node thus each node requires independent processing in the current algorithm. This does not result in a noticeable performance penalty using thread parallelism on CPUs. However it has bigger performance implications for GPU executions due to the rather sparse individual node operations. Alternatively, an amalgamation of operations for multiple nodes could result in denser operations on a larger *CUDA* block, which in general is more performant. Such an amalgamation, though, would require a significant rearranging of our algorithms, which is outside the scope of this work.

We thus allocate one *CUDA* block per node, as only they can be executed independently by different streaming multiprocessors of the GPU. A more fine-grained parallelism of allocating one *CUDA*-thread per node would result in excessive code divergence. Since all 32 threads of one *CUDA*-warp (which is also the smallest reasonable *CUDA*-block size) have to execute the same instruction, a vast proportion of threads would be idle at each cycle. Our algorithm only deals with small matrices and vectors, corresponding to the rather low polynomial orders of the elements, so we actually specify the smallest reasonable *CUDA* block size of 32 threads.

Algorithm 3.4 New node colouring scheme

```
1: procedure ColourNodes(N)
2:     N_VertexEdge ← all vertex and edge nodes in N
3:     N_VertexEdge ← Sort(N_VertexEdge)        ▷ Sort in descending order of connected elements
4:     N_Face ← all face nodes in N
5:     N_Volume ← all volume nodes in N
6:     C_VertexEdge ←CreateColourSets(N_VertexEdge)
7:     C_Face ←CreateColourSets(N_Face)
8:     C_Volume ←CreateColourSets(N_Volume)
9:     return C = C_VertexEdge ∪ C_Face ∪ C_Volume
10: end procedure
11: procedure CreateColourSets(N)
12:     while N is not empty do
13:         M ← all uncoloured nodes in N
14:         Create a new colourset c
15:         while M is not empty do
16:             Select the next node n ∈ M, include n in c
17:             Let A ← the set of elements connected to node n
18:             Remove n and all nodes in A from M
19:         end while
20:     end while
21:     return all colour sets c
22: end procedure
```

To process the maximum number of blocks, the register limit would need to be 32 double floats per *CUDA* thread, however, this would lead to extensive memory copying from the registers to the cache and vice versa. We have tested different settings of register limits and found that 128 double floats per *CUDA* thread results in the best run times on all three considered GPUs.

3.3.4 Improving the node colouring algorithm

A few aspects concerning the node balancing need to be discussed. Firstly, it is favourable to have a sufficient number of nodes within each colourset to achieve full occupancy of the utilized hardware. In a naive node colouring implementation with a random treatment of the free-to-move nodes, however, the tail of the colour sets consisted only of a few nodes, thus slowing down the overall algorithm. The naive implementation here corresponds to a single call of the procedure CreateColourSets($\boldsymbol{N}$) in Algorithm 3.4, using the complete and unordered set of nodes $\boldsymbol{N}$.

A second aspect is to construct sets in which each node requires the same amount of processing steps. This can be realised by constructing colour sets of interior nodes only and others for element boundary nodes which are connected with the same or at least similar numbers of elements.

We now implement a new node colouring scheme taking into account the two mentioned

aspects, as given in Algorithm 3.4. All free-to-move vertex and edge nodes are treated in descending order of their number of connected elements. This way nodes connected to similar numbers of elements are grouped together. Further, nodes connected to few elements are treated later, allowing to fill in the gaps of untreated elements in the mesh that have been induced by the nodes connected to many elements. This scheme results in a very small tail of colour sets that do not have a sufficient size. The face and the interior nodes are treated separately each, this way creating colour sets that are equal in size and in the number of connected elements.

Compared with the naive node colouring, the sorted node colouring improves runtimes on all considered architectures for typical large meshes by around 2%. While each sorted colourset is processed more efficiently, the number of colour sets is slightly increased, adding an overhead that compromises the overall efficiency gain.

3.4 Performance evaluation

To evaluate the performance of the *Kokkos* implementation of the variational framework, we first outline the methodology for the tests, and compare the initial CPU-only implementation of [115] against different variants of the *Kokkos* implementations. Then we expand the tests to consider a wider range of multicore and manycore CPU and GPU architectures, and assess the performance of the implementation at a range of polynomial orders in terms of hardware occupancy and the relative cost per degree of freedom.

3.4.1 Methodology

Since the number of choices of parameters in this study is potentially very wide, we first consider sensible limitations in order to examine key aspects of the implementation.

The first point to consider is the choice of functional $\mathcal{E}$ in the variational framework, which represents the solid body model to be adopted. We note that there are minor performance differences between the four different methods used within the variational framework, since each functional has its own unique characteristic under the minimisation process, meaning that the number of iterations required to converge to an optimised mesh may vary considerably. In this work, we consider only the hyper-elastic functional, since this was shown in [115] to consistently yield higher quality meshes. We do however note that for an entirely complete evaluation, the quality of the resulting meshes need to be taken into account. This is however outside the scope of this thesis.

Secondly, in this work we only consider meshes consisting of tetrahedral elements and not, for example, hybrid meshes of e.g. tetrahedra and prismatic elements. This choice enables us to remove an aspect of load balancing from our implementation, since different element types require different computational requirements in the calculation of the energy functional. This restriction therefore allows us to consider a well-balanced problem which is more capable of attaining higher peak performance of the hardware. We note that, as

shown in [115], in order for the optimisation algorithm to work reliably, the field values on each elemental node need to be evaluated using quadratures of at least four orders higher than the polynomial order of the element, using distributions of points that have positive quadrature weights. As nodal bases and associated quadrature rules satisfying this property for tetrahedral elements are only known up to 9th order, our tests are confined to tetrahedra of up to 5th order.

Finally, we consider only meshes which are initially valid, so that all nodes undergo the same path in the optimisation routine, therefore allowing for a more accurate performance evaluation. We found that nodes of invalid elements typically require up to ten times more operations within the reverse line search than nodes of valid elements. We only evaluate the parallel part of the optimisation algorithm, neglecting the preprocessing and the initial data copying to the device. All timings are taken for 10 optimisation steps and are an average of five runs.

3.4.2 The different implementations

In this performance evaluation, we compare four different implementations based on the initial and the adapted algorithm. These are three versions to be executed on CPUs and one to be executed on GPUs, where the first two are based on Algorithm 3.2 and the latter two on Algorithm 3.3:

1. **Initial native *Pthreads*.** The first implementation is the baseline that achieves multi-threading on a CPU with a native *Pthreads* implementation using a light-weight thread-manager and utilises associative object-oriented data layouts.

2. **Intermediate *Kokkos-OpenMP*.** For the second implementation the direct thread scheduling of *Pthreads* has been replaced with a *Kokkos* parallel for loop using the *OpenMP* backend. Alternatively, a *Pthreads* backend could have been specified, however, was found to perform slightly worse.

3. **Full *Kokkos-OpenMP*.** The third implementation is a full *Kokkos* version utilising plain data structures and is compiled with the *OpenMP* backend. Again, the *OpenMP* backend gave a superior performance over the *Pthreads* backend and has been selected. Using the same backend for version 2 and 3 further allows a better comparison of the influence of changing the data structures.

4. **Full *Kokkos-CUDA*.** The fourth version's code-base is identical to version 3, but is compiled for the *CUDA* backend and to be run on a GPU, enabled by the portability of *Kokkos*.

3.4.3 Utilised hardware

For the multicore CPU system an Intel(R) Xeon(R) E5-2670v3 @ 2.30GHz processors is employed, consisting of 2 sockets of 12 cores each and allowing up to 48 threads using

	Nvidia Tesla K40	Nvidia GTX 1070	Nvidia Tesla P100
Architecture	Kepler 3.7	Pascal 6.1	Pascal 6.0
Streaming Multiprocessors	15	15	56
FP64 (DP) Cores / SM	64	4	32
FP32 (SP) Cores / SM	192	128	64
GPU Boost Clock	875MHz	1987MHz	1480MHz
Peak FP64 GFLOPs	1680	238.44	5304.32
DRAM Memory	12GB	8GB	12GB
Price (April 2017)	$3489	$455	$4912
Thermal Design Point	235W	150W	250W

Table 3.1: Selected GPU performance specifications.

hyper-threading. The two sockets can achieve a maximum theoretical performance of 883 DP-GFLOPs (double precision gigaflop per second), assuming AVX2 (Advanced Vector Instruction 2.0) and FMA (fused multiply-add) operations are used. The operating system of the CPU machine is Debian 8.7, and as the compiler gcc 4.9.2 with the *-O3* optimisation flag has been used.

As a manycore accelerator system we employ an Intel Xeon Phi 7210 of the *Knights Landing* architecture, consisting of 64 cores operating at 1.3GHz with a boost clock of 1.5GHz, and using up to 4 hyper-threads per core. We operate the device in the *flat* setting, making use of the 16GB of fast MCDRAM. The Xeon Phi 7210 has a theoretical peak performance of 3072 DP-GFLOPs and a thermal design point of 215W.

As GPUs we employ a Nvidia Tesla K40, a Nvidia GTX 1070, and a Nvidia Pascal P100. Some performance specifications are given in Table 3.1. The executables for the Tesla K40 and the Tesla P100 have been compiled on a *CentOS* Linux 7.3.1611 server, using gcc 4.8.5 with the *-O3* flag. The GTX 1070 machine operates on an *OpenSUSE* Tumbleweed system and the utilised compiler has been gcc 5.4.1 with the *-O3* flag. The numbers of FP64 or double precision (DP) cores per streaming multiprocessor (SM) show that the Tesla cards are intended for DP-intense HPC applications, whereas the GTX card offers mostly single precision cores and is hence intended as a gaming card. The low price tag however can still make it a good option for some applications. On all cards we utilise the DP cores only, as our tests show that the algorithm requires higher accuracy to obtain reasonable results. On all three GPUs we further limit the register size per thread, as discussed in Section 3.3.3, using *-maxregcount 128*.

3.4.4 Strong scaling of CPU implementations

The first of the two main points to assess is the performance of the full *Kokkos* CPU implementation (version 3) compared to the initial CPU implementation (version 1). Evaluating the intermediate implementation (version 2) allows to separate the performance influence of the *Kokkos* thread scheduling and the *Kokkos* data structures.

For the strong scaling exercise we use an initially valid, but non-optimised mesh of 33K

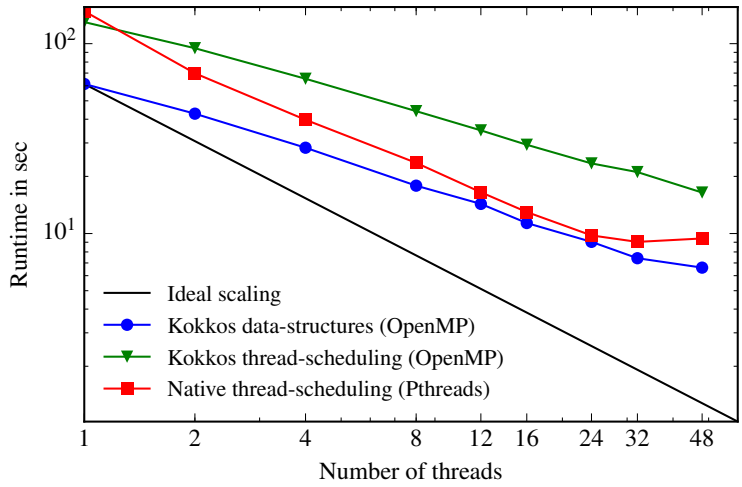

Figure 3.3: Strong scaling of the parallel part of different CPU implementations executed on a 24-core CPU

elements and 400K degrees of freedom, using 1 to 48 threads. The elements are third-order tetrahedra and an over-integration of fourth order is applied. The timings for all three CPU versions can be found in Figure 3.3.

We clearly observe that the scaling of all versions is roughly linear, but with different slopes. A negative effect of using hyper-threading with the native *Pthreads* version is relatively pronounced. For up to 24 threads (1 thread per core), this version achieves a very good scaling of 64%, but no further speed-up using hyper-threading can be realised. The full *Kokkos* implementation achieves a scaling of 20%. Throughout the range the intermediate *Kokkos* version shows a similar scaling, but 2 to 2.5 times slower run times than the full *Kokkos* version. Using the full number of threads, the full *Kokkos* version is 1.42 times faster than the initial *Pthreads* implementation and 2.48 times faster than the intermediate version.

For the strong scaling test using the full *Kokkos* version and one thread, 18.32% of the theoretical FLOPs (considering AVX2 and FMA) have been achieved, and still 3.54% using 48 threads. These numbers have been calculated as

$$\%(\text{peak FLOPs}) = \frac{\text{achieved FLOPs}}{\text{peak FLOPs}} \frac{\text{maximum threads}}{\text{utilised threads}}. \tag{3.10}$$

These results indicate that the thread-scheduling of *Kokkos* performs worse than our native *Pthreads* thread-scheduler. This could be caused by the difference between the dynamic thread scheduling of the native *Pthreads* version and the fork-join thread scheduling of the *Kokkos* versions. The execution times using one thread, however, show that the *Kokkos* thread scheduling overhead is at least on par with *Pthreads*. Hence the *Kokkos* infrastructure allows for similar performance, but the thread-scheduling could be improved. The performance deficit might be much smaller in cases with well balanced loads per thread, but in our case of varying work sizes per node-colour group the effect is signific-

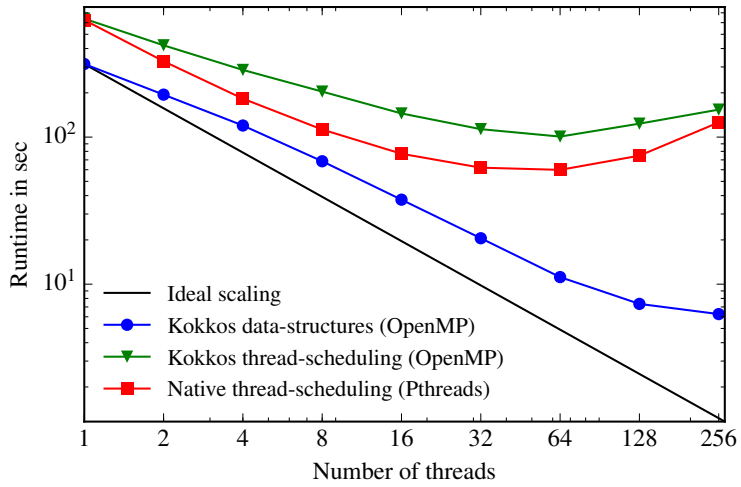

Figure 3.4: Strong scaling of the parallel part of different manycore implementations executed on a *Knights Landing* (KNL) accelerator with 64 cores

ant.

A second important observation is that the new data layout results in faster run times, due to an improved memory efficiency. The improved efficiency can be assigned both to the effects of the polymorphic *Kokkos* view layout, but also to the general re-factoring of the data structures, that has been a precondition for using *Kokkos* views. The change from the associative data layout of shared pointers to flat data arrays will in itself result in faster memory access. Moreover, using 32 or 48 threads and thus in the domain of hyper-threading, the new *Kokkos-OpenMP* version performs far better than the two other versions. This is a encouraging result, considering that we used a portable programming model and did not optimise the algorithms or data structures for a specific architecture.

As a final test, we repeat the strong scaling exercise using the same mesh on the manycore accelerator. The averaged timings of our three CPU implementations are shown in Figure 3.4. In this manycore environment, which is additionally hindered by stronger memory bandwidth restrictions, the effect of using simpler data structures can clearly be seen, with the full *Kokkos* implementation being by far the best performing. With the *Kokkos* thread scheduling and data structures a difference of one order of magnitude with a 9.5 times lower runtime than with the native *Pthreads* version is achieved. This indicates the suitability of the *Kokkos* data arrays for highly parallel shared memory systems. Up to 64 threads (1 thread per core) we obtain a very good scaling of 44% for the full *Kokkos* version, compared with 16% for the native version. Further, only the full *Kokkos* version can benefit from hyper-threading using up to 256 threads, whereas it is found counter-productive for the other two versions. Additionally, the one-thread performance of the manycore accelerator is about five times slower than the multicore CPU for all three versions. This can not be explained by the lower clock-speed alone, but is again caused by the more restricted cache memory bandwidths.

3.4.5 Performance comparison between architectures

Using the portable *Kokkos* programming model, we already achieved better performance on the CPU, compared to our initial implementation. The second main point to assess is if further performance benefits can be realised running the same algorithm on GPUs. We emphasize that, asides from specifying some arrays to be loaded to texture memory and others to shared memory (which can be specified using the *Kokkos*-syntax), no specific GPU optimisation has been undertaken. This section therefore aims to compare the suitability of CPU vs GPU systems, using the *Kokkos-CUDA* version on three different GPUs and the best performing CPU implementation, the full *Kokkos-OpenMP* version, on the CPU system and an Intel Xeon Phi of the *Knights Landing* architecture.

In this section we consider a set of four different meshes, all using the same geometry and the same number of elements, but using different polynomials orders and therefore different degrees of freedom (DOF), as shown in Table 3.2. This table also shows the average number of DP-GFLOP (double precision gigaflop) required by our algorithm for 10 optimisation steps, so it contains exactly the number of operations performed within our timing interval. The GFLOP measurements were obtained using the Nvidia Visual Profiler [88].

Element order	Elements	DOF	DP-GFLOP	FLOP/DOF
2^{nd}	95,982	338,469	163.6	483.5
3^{rd}	95,982	1,197,165	628.3	524.8
4^{th}	95,982	2,899,428	1,824.8	629.4
5^{th}	95,982	5,733,204	4,943.9	862.3

Table 3.2: Statistics of the architecture comparison set of meshes; including degrees of freedom (DOF) and double precision FLOP for 10 steps of the mesh optimisation algorithm.

The averaged run times obtained on our CPU system, the three GPU systems and the KNL accelerator are scaled by the DOF and given in Figure 3.5. Figure 3.6 additionally shows how much faster the GPU and accelerator run times are compared with the CPU run times. The main observation is that a faster runtime on the GPUs with the latest Pascal architecture is realised. The GTX 1070 is 150% to 200% faster, whereas the Tesla P100 is even 275% to 350% faster than the Xeon E5-2970v3 CPU. The performance on the Tesla K40, however, is worse than on the CPU system. The Intel Xeon Phi 7120 accelerator is up to 150% faster than the CPU.

The run times vary for different element polynomial order, which is due to two opposing effects. Firstly, the arithmetic intensity (the average number of operations or FLOP required by our algorithm per degree of freedom) increases with element polynomial order, as seen in Table 3.2, which is a common observation for high-order methods. Secondly, the algorithm maps with varying efficiency to the different hardware. As a straightforward measure we consult the percentages of theoretical peak FLOPs that are achieved, given in Figure 3.7. The higher the polynomial order, the more efficiently the hardware is utilised,

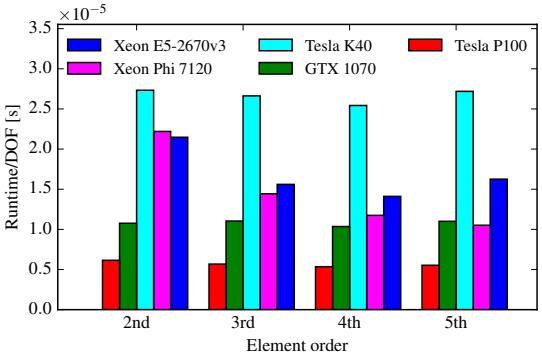

Figure 3.5: Run times normalised by DOFs for meshes with varying element order on different systems.

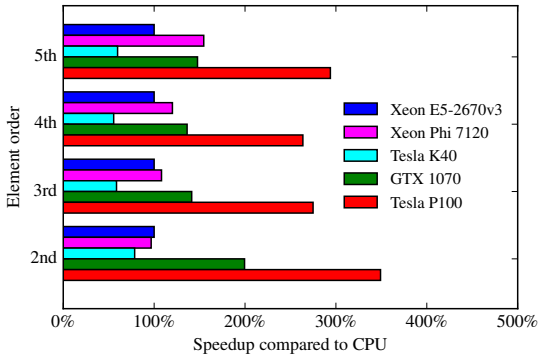

Figure 3.6: Speed-ups of different GPU/accelerator systems compared to CPUs for meshes with varying element order.

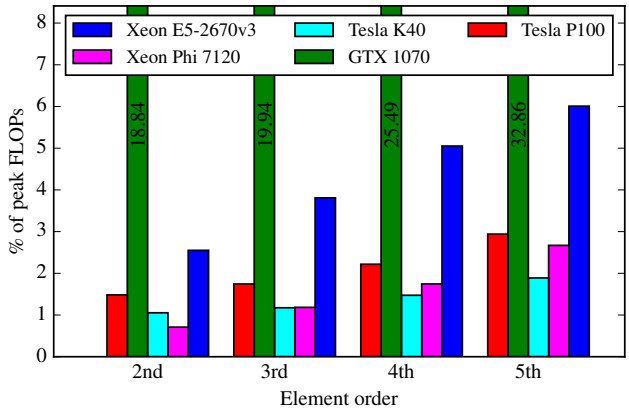

Figure 3.7: Achieved percentages of theoretical peak FLOPs (DP) of different systems for meshes with varying element order.

across all systems. This is due to more compact data structures of high-order elements, that allow a more efficient memory access. Both effects combined, 4th order elements achieve the optimum run time per DOF for all GPU- and the CPU systems. The Xeon Phi accelerator benefits from even higher polynomial orders.

An important observation is, though, that our architecture-independent algorithm can not fully utilise the large number of DP cores on the Tesla GPUs, whereas the 4 DP cores per SM on the GTX card can be well utilised. The evaluation of the memory bandwidth below explains this effect, which is most probably caused by the different ratio of DP-compute cores to load-store-units (LSUs) and thus available bandwidth.

3.4.6 Detailed GPU performance evaluation

Opposed to CPUs, it is not possible to execute a code only on a specific number of streaming multiprocessors of a GPU. Hence no meaningful scaling exercise for single GPUs can be devised. Yet, there are other meaningful performance metrics that can be evaluated for GPU executions. These are readily available using the *nvprof* profiler, that is part of

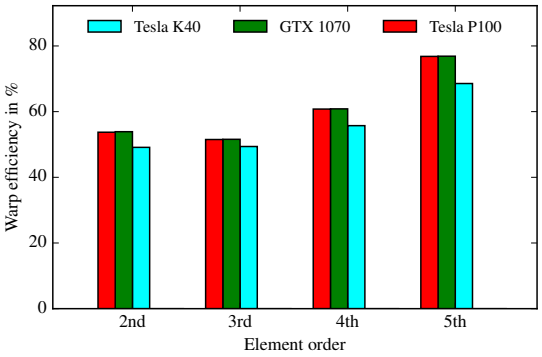

Figure 3.8: Warp efficiency of different GPU systems for meshes with varying element order.

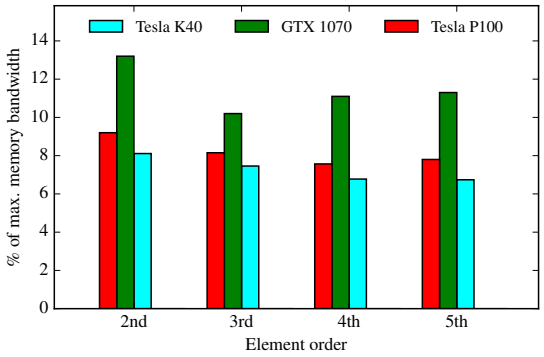

Figure 3.9: Achieved percentage of theoretical peak bandwidth of the device memory for different GPU systems and meshes with varying element order.

the *CUDA* package [88]. The equivalent profiling metrics for modern CPUs are very difficult to examine reliably, so we choose to omit those here.

Apart from the achieved percentage of theoretical peak FLOPs, we consider the warp efficiency. It is defined as the average percentage of threads in a warp (a *CUDA*-warp always consists of 32 threads) that are performing useful work. This metric is very important as any improvement will directly corresponds to an equally large improvement in percentage of peak FLOPs. The results are shown in Figure 3.8. The higher the polynomial order of the mesh elements, the higher is the observed warp efficiency. A considerable amount of time is spend evaluating the integrand at the quadrature points of an element, which are assigned a thread each. Higher-order elements have more quadrature points and can hence be distributed better to fill up multiples of 32 threads or one warp. The slightly lower efficiency of the Tesla K40 might be explained by the different *CUDA* compute capability. This trend partly explains why a higher element order results in a higher percentage of theoretical peak FLOPs. The other crucial limiting factor is the memory access.

To this end we evaluate the memory bandwidth of the DRAM on the device. It is a metric to evaluate how well the data structures are mapped to the physical memory in order to allow efficient coalesced memory-access. The percentage of the theoretical peak bandwidth is given in Figure 3.9. The important observation is that – independently of the GPU and the polynomial order – the utilisation of the memory bandwidth is rather low. We can infer theoretically from our algorithm that the calculations performed for each node deal with many small individual arrays. We have already discussed that at the same time our algorithm is very memory intense, requiring large register sizes. This will lead to a low degree of coalesced memory accesses and hence result in the low observed memory bandwidth. Further, we also did not undertake any attempts to optimise the memory operations. Our algorithm also reuses a lot of data in the optimisation loops, so it is not loaded and written to the DRAM memory every time, but instead it is probably

cached on L1 and L2 cache. It is therefore very likely, that the algorithm is throttled by L1 or L2 bandwidth, too.

On the GTX gaming GPU, the memory bandwidth to DP-core ratio is much higher than on the Tesla HPC cards, resulting in a higher percentage of peak FLOPs.

3.4.7 Cost comparison between hardware systems

There is no established procedure to compare the overall efficiency between very different architectures, like CPUs and GPUs. The two fundamental constraints for HPC users are time to solution and cost budgets. Time to solution can be compared straightforwardly between all kind of systems. But without taking costs into account it would be too simplistic to compare the hypothetical case of a 1,000$ CPU system with a 5,000$ GPU system and claim the GPU system performs better on the grounds of achieving half the run time for a given simulation. Provided the applications achieve a good scaling, better run times can most often be gained by spending a higher budget. It follows that the costs to operate a certain hardware system need to be taken into account.

The biggest annualised operating costs contributor for a HPC server is the computing hardware acquisition cost, which has also been considered in [119]. This reference proposes a novel cost metric termed resource utilisation, which is the product of run time and the hardware acquisition cost, with the units $\$\times s$. However, this is only a combined time-cost metric when hardware acquisition costs are dominant and not annualised, as in the case of a local workstation. In a HPC server case, where the user pays for a certain length of user-time, however, the product of costs per time and run time has the unit $, and it becomes a pure cost metric. The resource utilisation metric also does not include electricity consumption, maintenance and peripheral costs, which are significant and also need to be carefully considered.

System	monthly price in $
Xeon E5-2670v3	913.53
Xeon Phi 7120 + RAM	774.56
Tesla K40 + host	1001.64
GTX 1070 + host	499.56
Tesla P100 + host	1412.81

Table 3.3: Averaged monthly prices for equivalent systems on bare-metal cloud services, as of April 2017.

Cloud computing providers take all these costs into account when determining their consumer price. Academic HPC facilities, too, will consider the same cost contributors to meet their annual cost budget, even though prices for user-time are not charged as such. We therefore propose a cost metric based on prices of bare-metal cloud-computing providers. Only renting *bare-metal* and not *elastic* cloud services would guarantee the algorithms to be executed on a specific hardware and thus allow for an accurate perform-

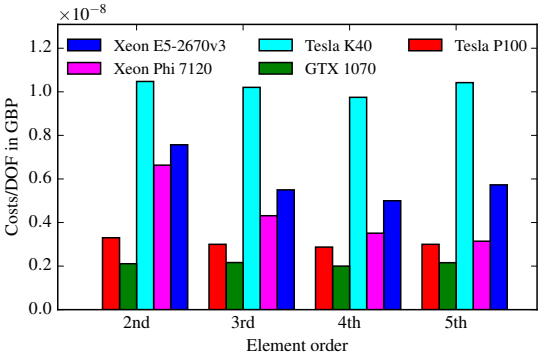

Figure 3.10: Bare-metal cloud-computing costs normalised by DOFs for meshes with varying element order on different systems.

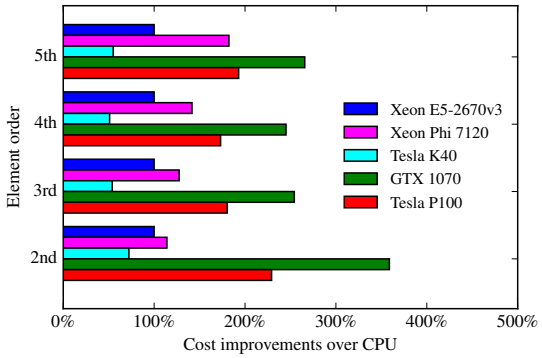

Figure 3.11: Bare-metal cloud-computing cost improvements of different GPU/accelerator systems compared to CPUs for meshes with varying element order.

ance and cost evaluation. We gathered the monthly operating prices from three major providers (Amazon Web Services, Google Cloud Platform, IBM Cloud) for equivalent systems that we used for our performance runs, as of April 2017. For the GPU systems we considered the monthly prices of the GPU itself and added the monthly price of a simple host system consisting of an 8-core Xeon E5-2650v3 processor and 64GB RAM. For the Xeon Phi accelerator system we considered the device plus 64GB of RAM. The averaged prices for the equivalent systems are given in Table 3.3. Based on theoretical performance to price, we would consider a fair charge for a Tesla K40 compared to a Tesla P100 to be lower, however.

The costs of our runs are calculated as the product of run time and monthly price for the system and normalised by the DOFs. The absolute numbers are given in Figure 3.10 and the cost improvements compared to the CPU system are given in Figure 3.11. The main observation is that both Pascal GPUs achieve a cost advantage over the CPU system of 170% to 225% for the Tesla P100 and an even higher 250% to 365% for the GTX 1070. The Tesla K40 is not cost efficient, due to both the low hardware utilisation and the currently high price tag. The Xeon Phi accelerator achieves a cost advantage of 115% to 180%. Comparing the different polynomial orders of the elements, unsurprisingly, the exact same trends as for the run times can be observed. For all GPU architectures and the CPU, the 4th order element is the most efficient, while this effect is only pronounced on the CPU. The cost variation of the Intel Xeon Phi accelerator is the strongest, being hardly competitive using 2nd order elements, but being in the same range as the Pascal GPUs for 5th order elements.

We can compare the two considered constraints costs and run times in a single scatterplot, see Figure 3.12. This conveys the found results very effectively; compared with the CPU, the two Pascal GPUs Tesla P100 and GTX 1070 perform better on both metrics. The choice which hardware to employ can then be made depending on which constraint is more important for the individual user. The Xeon Phi accelerator performs better than

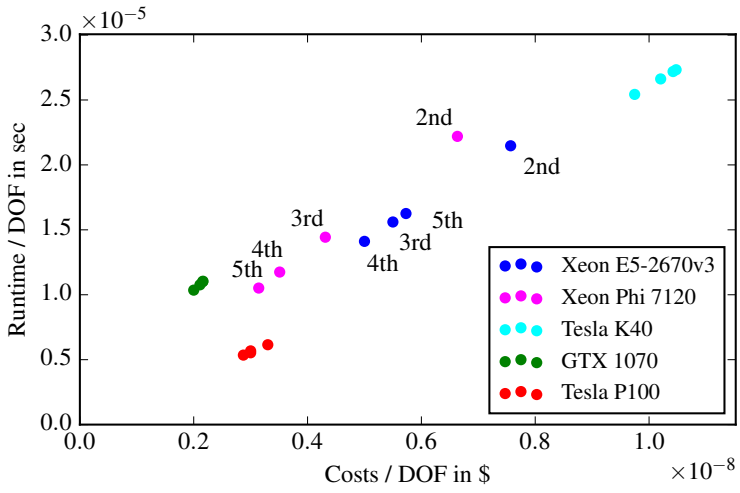

Figure 3.12: Comparison of bare-metal cloud-computing costs and runtimes per DOF for different systems and varying element order.

the CPU, too, but is never better than the two Pascal GPUs, in terms of Pareto efficiency. The high performance variation of the CPU and the Xeon Phi accelerator depending on the element order is further clearly shown.

3.5 Conclusion

We have implemented a higher-order mesh optimisation algorithm with an architecture independent programming model using the *Kokkos* library. The general mesh optimisation algorithm allows the correction of invalid meshes and the optimisation of valid high-order meshes using a variational energy functional. The implementation has been presented with tetrahedra of second to fifth order and the hyper-elastic energy functional. The new *Kokkos* implementation is based on polymorphic data structures that allow efficient memory access on both CPU and GPU architectures. The implementation further makes use of thread-scheduling with an *OpenMP* backend on CPUs and a *CUDA* backend on GPUs. For the *CUDA* backend we use *Kokkos* functionalities to specify the memory locality of certain data. However, we like to emphasise that we did not attempt to optimise our algorithm for any specific parallel architecture. Yet, we have shown that we can port our high-order mesh optimisation algorithm to GPUs and Xeon Phi accelerators, and obtain both a time and cost advantage compared to CPU runs.

Looking into the results in more detail, we found evidence that *Kokkos* data structures are processed more efficiently than the initial associative ones that use arrays of shared pointers. This effect was especially pronounced on Xeon Phi *Knights Landing* accelerators with a runtime difference of almost one order of magnitude. This shows the importance of efficient data structures on manycore devices and future hardware systems following this trend. The *Kokkos* thread-scheduling, however, was found to lack some efficiency, probably because of the underlying fork-join model.

We have additionally developed a cost metric that is based on monthly prices for equivalent systems on bare-metal cloud-computing servers. Only with both run-times and a cost metric, a fair comparisons between different architectures is possible. Compared with our CPU system, the Tesla P100 is 170% to 220% and the GTX1070 250% to 360% more cost efficient. The Xeon Phi accelerator's cost efficiency increases from 115% up to 180% with increasing element order and hence arithmetic intensity.

We have to acknowledge that the *Kokkos-CUDA* implementation achieves only a very low efficiency on the Tesla GPUs, most probably due to the rather sparse operations and an inefficient memory access. To improve this considerably, either detailed code optimisation or a rewriting of the underlying algorithm would be necessary. Both aspects, though, defeat our attempt in porting an initial CPU version to other architectures with minimal effort using an architecture independent programming model.

Once a full *Kokkos* implementation that works with the *CUDA* backend has been completed, it is deemed to offer good code maintainability. However, we have shown that legacy code relying on heavy object oriented programming (OOP) features requires substantial refactoring of data structures and associated data management. A programming model that would avoid such a refactoring step would be highly beneficial.

The *Kokkos* library further hides parts of the parallel codebase when using the Nvidia Visual Profiler, making it more complex to realise code optimisation. When compiling for the *CUDA* backend it is also still necessary to tune the applications in terms of optimal block sizes and register limits, which would ideally be automated in a fully architecture independent programming model.

Chapter 4

A comparison of portable programming models using a Helmholtz solver mini-application

In this chapter, we consider the application of three performance-portable programming models *OpenMP*, *OpenACC*, and *Kokkos* in the context of a high-order spectral element code. We aim to evaluate whether the use of these models allows code developers to deliver high-performance solvers for computational fluid dynamics simulations that are capable of effectively utilising both many-core CPU and GPU architectures. Using the core elliptic Helmholtz solver for the incompressible Navier-Stokes equation as a benchmarking guide, we evaluate the performance of these models on a range of unstructured meshes and give guidelines for the translation of existing codebases and their data structures to these models. To this end we explored how to parallelise the elemental operations using a *CUDA*-based parallelisation model. We employed a matrix-free formulation with a parallelisation that assigns one mesh element per *CUDA*-thread in conjunction with customised small BLAS kernels and interleaved memory layouts. We further establish a path to minimise the challenge of parallelising a continuous Galerkin discretisation by using a mesh colouring approach.

The work associated with this chapter has been published in references [29, 27].

This chapter is structured as follows: We continue with an introduction to this work in Section 4.1. In Section 4.2 we outline the mathematical model under consideration for the Helmholtz solver. The programming models and the parallel implementation of the solver is subsequently discussed in Section 4.3. We conduct a thorough performance evaluation in Section 4.4 on both many-core CPU and GPU hardware, before giving conclusions and a summary in Section 4.5.

4.1 Introduction

The aim of this work is to assess three promising portable programming models, namely *Kokkos* [17], *OpenMP* [92], and *OpenACC* [90], in the context of high-order methods for CFD. Aspects to be compared are the implementational effort and code maintainability, as well as the performance and its portability across architectures. In particular, we focus on computational hot-spots of an implicit time-stepping solver for the incompressible Navier-Stokes equation. Implicit time-stepping can be advantageous for turbulent flow with low Mach numbers, that avoid severe CFL number restrictions associated with explicit schemes.

This work provides novelty from a number of perspectives. The first aspect is to assess the performance portability of individual programming models in a representative environment. Although these aspects are regularly assessed, the corresponding results are not meaningful enough for our purpose, due to their standardised nature, which is discussed in the following. Common benchmark problems are most often based on kernels for BLAS operations, or so-called *mini-applications* or *mini-apps* for a variety of solver types. Examples of these can be found in a number of parallel benchmark suites, e.g. [23, 50, 18, 19, 108]. In this approach, stripped-down standalone versions of the core algorithms that form complete solvers are implemented. This allows for a simpler development environment, whilst still ensuring generally representative performance characteristics. Such mini-apps then allow for easier performance benchmarking and experimentations with programming models and other aspects, such as utilised hardware or compiler versions. While each mini-app solver can consist either of bespoke or of standard BLAS-type kernels, the relative weights of these kernels and their interactions will still be an important factor in the overall performance, so that isolated evaluations of individual kernels alone will not yield meaningful enough results. The mini-application that comes the closest to our purpose would be *miniFE* of the *Mantevo* suite [50]: however, this is only for a linear finite element solver, whereas we are interested in high-order finite element solvers.

Here we follow this approach to develop a mini-application to model the performance characteristics of a complete incompressible Navier-Stokes flow solver, as implemented within the spectral/*hp* element framework *Nektar++* [14]. While we make use of the *Nektar++* framework for pre- and post-processing, the time-stepping loop that we monitor for performance has been cleared of typical *Nektar++* features and flattened to generic *C++* code, before applying the parallel programming models. Further to this, we note that the complete incompressible Navier-Stokes implementation requires particularly heavy implementation effort that does not necessarily affect computational performance (e.g. from specialised boundary conditions). We therefore select a simpler, more lightweight solver, that still contains the relevant features of the complete solver; in our case this will be an implicit solver for the 2D Helmholtz equation.

As discussed in Section 4.2.1, this forms the essential implicit solve for both pressure Poisson and velocity correction steps within the commonly used fractional time-stepping

scheme for incompressible Navier-Stokes equations [64]. The matrices arising from the discretisation are positive definite and, unless the mesh is highly distorted, well-conditioned and thus we employ the conjugate gradient method with a Jacobi preconditioner. Yet, we note that the convergence of the chosen preconditioner is not independent of the mesh size, both in terms of number of elements and polynomial order of the elements.

The algorithm is executed in a matrix-free fashion, by which we mean that neither a large sparse assembled global matrix nor a series of dense local elemental matrices are constructed. Instead, the action of the matrix-vector multiplication is evaluated via the definition of the discrete Helmholtz operator, in order to reduce the memory bandwidth requirements of the algorithm and improve arithmetic intensity. As spatial discretisation we employ triangular elements with a modal continuous Galerkin (CG) formulation from reference [63]. The modal basis is formed with a tensor product, so that we can use the efficient sum-factorisation technique within elemental operations.

A second novel aspect of this work is the focus on implicit as opposed to explicit time-stepping routines. Certainly the combination of using explicit time-stepping for high-order methods alongside performance portable and architecture independent methods running on both CPUs and GPUs has been demonstrated in solvers such as *PyFR* [122]. Additionally, work by other finite element groups such as *deal.II* [72] and *Dune* [9] on mostly tensor product quadrilateral and hexahedral elements has been conducted, but typically from the CPU perspective. In this work, we aim to demonstrated how architecture-independent programming can be applied from the context of implicit time-stepping.

A third aspect is the focus on shared-memory parallelisation strategies. While current HPC architectures are characterised by a high degree of parallelism, it does not only come in the more traditional form of distributed memory parallelism, though, as would be the case for clusters of single-core CPUs. Instead the parallelism comes in the form of intra-node or shared memory parallelism over multiple compute cores. Previously, a good *MPI* distributed memory implementation that achieves perfect weak and strong scaling, could be realised by ensuring the problem size per *MPI*-rank is sufficient and by hiding the communication costs. However, the additional levels of parallelism bring applications closer or beyond the limits of sufficient problem sizes per rank. Instead, the more fine-grained shared memory parallelism needs to be additionally employed. This introduces the challenges of identifying suitable parallel algorithms, as well as programming-models and compilers that can map the algorithmic parallelism to the underlying hardware. We evaluate different kind of algorithms that form part of our mini-application, and investigate how suitable each of them are for modern GPU systems. Especially for reduction-type operations, care needs to be taken to achieve good efficiency. It is this aspect that often results in less ideal strong-scaling within a compute node. Additionally, the layout of data structures needs to be reconsidered. In order to load data efficiently to the vector lanes of CPUs and the threads of a GPU, the data structures need to be vectorised or *interleaved*, too.

4.2 Method

While the Helmholtz equation describes multiple physical phenomena, here we focus on its occurrence within solvers for the incompressible Navier-Stokes equation. Due to its relevance, numerous solvers for the Helmholtz equation have been proposed (for example fast multipole methods [46]), which are often more efficient than the method we employ here. However, we need to consider that our Helmholtz solver needs to seamlessly integrate into the overall Navier-Stokes solver, restricting us to use a solver based on high-order spectral element discretisations and irregular domains. Furthermore, this work's aim is not to present the most efficient method, but to take a typical method of high-order flow solvers and investigate its parallel performance using three different portable programming models.

4.2.1 Relevance of the Helmholtz equation

For low Mach number flows, the Navier-Stokes equation can be simplified by ignoring compressibility effects, which yields the incompressible Navier-Stokes equation. This simplifies the continuity and the momentum equation and decouples them from the energy equation. The governing equations are:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = -\nabla p + \nu \nabla^2 u + f \tag{4.1}$$

$$\nabla \cdot u = 0\,, \tag{4.2}$$

where u is the velocity vector, p is the specific pressure (including density), ν is the kinematic viscosity, and f is a field force vector.

This system of equations can be solved in a coupled way, or by applying a splitting scheme where the velocity and pressure system are decoupled. This leads to a smaller system to be solved, which increases the efficiency, however introduces an additional splitting scheme error.

In this derivation we follow the original approach proposed in the work of Karniadakis, Israeli, and Orszag [64], but note it has later been refined in reference [45].

The high-order splitting scheme consists of three steps: first, an explicit advection of the non-linear terms, second, the solution of the pressure Poisson system, and third, solving a Helmholtz system to enforce the viscous terms and velocity boundary conditions. Both the second and the third step are solved by implicit schemes that lead to linear systems of equations.

In the first step, the non-linear term $N(u) = u \cdot \nabla u$ is advanced in time using a suitable explicit time-stepping method. As the stability limit of such schemes is smaller than for the implicit time-stepping of the remaining terms, a so called sub-stepping or sub-cycling method is often employed. Here multiple small explicit time-steps are performed to advance the solution by the same physical time as used by a single step of the implicit solver.

In the second step, the pressure Poisson system given by

$$\nabla^2 p^{n+1} = \nabla \cdot \left(\frac{\hat{u}}{\Delta t} - \hat{N} \right) \tag{4.3}$$

is solved, where $\hat{u} = u^n$ and $\hat{N} = N^n$, if a one-stage integration scheme is chosen.

In the third step, the pressure obtained from the second step is used to solve for the velocity u^{n+1}. This Helmholtz problem is given by

$$\left(\nabla^2 - \frac{\gamma_0}{\nu \Delta t} \right) u^{n+1} = - \left(\frac{\gamma_0}{\nu \Delta t} \right) \hat{u} + \frac{1}{\nu} \nabla p^{n+1} \,, \tag{4.4}$$

where $\gamma_0 = 1$, if a one-stage integration scheme is chosen. For multi-stage integration schemes different values for $\hat{u}$, $\hat{N}$, and γ_0 have to be specified.

Solving the incompressible Navier-Stokes equation with the presented approach highlights the central role of a solver for the Helmholtz equation. (It can also be employed for the Poisson equation, by setting the mass term to zero.) The remaining parts to obtain an efficient incompressible Navier-Stokes solver would be an explicit solver for the non-linear advection terms, and an implementation of the pressure and outflow boundary conditions. Efficient GPU implementations of explicit solvers are shown to be relatively straightforward, as due to the locality of the scheme data can be effectively streamed through the GPU.

4.2.2 Galerkin approximation

We consider the solution of the inhomogeneous Helmholtz equation for a scalar field variable u, defined in a domain Ω with boundary $\Gamma = \Gamma_D \cup \Gamma_N$, represented by the system

$$\nabla^2 u - \lambda u = f \qquad \text{on } \Omega \tag{4.5}$$

$$u = u^D \qquad \text{on } \Gamma_D \tag{4.6}$$

$$\frac{\partial u}{\partial n} = g_N \qquad \text{on } \Gamma_N \,, \tag{4.7}$$

where λ is a real positive coefficient, f is a forcing function, and Γ_D and Γ_N denote the parts of the boundary where we impose Dirichlet and Neumann boundary conditions, respectively.

Defining the inner products

$$(a, b)_\Omega = \int_\Omega ab \, d\Omega \,, \qquad \langle a, b \rangle_\Gamma = \int_\Gamma ab \, d\Gamma \,, \tag{4.8}$$

the weak form of the system is obtained through the inner product with a shape function v, as

$$\left(v, \nabla^2 u \right)_\Omega - \lambda \left(v, u \right)_\Omega = \left(v, f \right)_\Omega \,. \tag{4.9}$$

We can now apply the divergence theorem to reduce continuity requirements and get

$$(\nabla v, \nabla u)_\Omega + \lambda (v, u)_\Omega = \left\langle v, \frac{\partial u}{\partial n} \right\rangle_\Gamma - (v, f)_\Omega \,, \tag{4.10}$$

where the boundary integral allows to directly specify the Neumann boundary conditions g_N. The Dirichlet boundary conditions u^D are lifted from the solution $u = u^H + u^D$, so that we retain the unknown homogeneous term u^H on the left-hand-side of the equation and all the known terms are moved to the right-hand side, namely

$$\left(\nabla v, \nabla u^H\right)_\Omega + \lambda \left(v, u^H\right)_\Omega = \langle v, g_N \rangle_{\Gamma_N} - (v, f)_\Omega - \left(\nabla v, \nabla u^D\right)_\Omega - \lambda \left(v, u^D\right)_\Omega \,. \tag{4.11}$$

We can now discretise this expression to obtain a linear system of the form

$$\boldsymbol{H}\, \hat{u}^H = \hat{b}\,, \tag{4.12}$$

where $\boldsymbol{H}$ is the matrix of coefficients of the discrete Helmholtz operator, $\hat{u}^H$ is a vector containing the discrete unknowns and $\hat{b}$ is the vector that results from the evaluation of the right-hand side using the known values of the forcing function and boundary conditions. The detailed construction of the discrete global Helmholtz operator $\boldsymbol{H}$ is given in Section 4.2.4.

4.2.3 Implicit solver

The linear Helmholtz equation system is to be solved iteratively using a Krylov subspace method. As the Helmholtz operator is symmetric and positive definite, we use a variant of the conjugate gradient method, which is reordered for reduced communication, as introduced in reference [20].

The method is initialised with an arbitrary solution $\hat{u}_0^H$:

$$r_0 = \hat{b} - \boldsymbol{H}\hat{u}_0^H \tag{4.13}$$

$$w_0 = \boldsymbol{C} r_0 \tag{4.14}$$

$$s_0 = \boldsymbol{H} w_0 \tag{4.15}$$

$$\beta = 0 \tag{4.16}$$

$$\alpha = \frac{r_0^T w_0}{r_0^T w_0}\,, \tag{4.17}$$

where r is the residual vector, $\boldsymbol{C}$ is a suitable preconditioner, w and s are the conjugate gradient vectors, and β and α are step sizes. The method then loops over the following

steps k, until the residual is smaller than a set tolerance $\epsilon > \epsilon_{tol}$:

$$p_{k+1} = \beta p_k + w_k \tag{4.18}$$
$$q_{k+1} = \beta q_k + s_k \tag{4.19}$$
$$\hat{u}^H_{k+1} = \alpha p_{k+1} + \hat{u}^H_k \tag{4.20}$$
$$r_{k+1} = r_k - \alpha q_{k+1} \tag{4.21}$$
$$w_{k+1} = \boldsymbol{C} r_{k+1} \tag{4.22}$$
$$s_{k+1} = \boldsymbol{H} w_{k+1} \tag{4.23}$$
$$\beta = \frac{r^T_{k+1} w_{k+1}}{r^T_k w_k} \tag{4.24}$$
$$\alpha = \frac{r^T_{k+1} w_{k+1}}{s^T_{k+1} w_{k+1} - r^T k + 1 w_{k+1} \cdot \beta_{k+1}/\alpha_k} \tag{4.25}$$
$$\epsilon = r^T_{k+1} r_{k+1}\,. \tag{4.26}$$

Here p and q are search direction vectors and $\hat{u}^H_{k+1}$ is the updated solution to the linear system that is to be solved.

The method accounts for a left preconditioning matrix $\boldsymbol{C}$, for which numerous options could be utilised. For finite element schemes that are used here, the global matrix has a block structure, with each block corresponding to one mesh element. It is thus beneficial to exploit this structure in a preconditioner for a finite element scheme.

Instead of being used as linear solver, the Gauss-Seidel and the Jacobi method can be used to construct a block preconditioner. The *symmetric block Gauss Seidel* (SGS) method is a better approximation for the inverse of the system matrix than the *Jacobi* method. The Jacobi method however is trivial to parallelise, whereas SGS is sequential. The SGS method can be further refined by neglecting some off-diagonal blocks to give a reduced off-block order (ROBO-SGS) method, as proposed in reference [12]. This slightly decreases the convergence rate of the method, since some higher order terms are neglected, but significantly reduces the memory footprint and the number of operations per iteration.

For simplicity and for a minimal memory footprint we use a diagonal Jacobi preconditioner. We acknowledge that the Jacobi preconditioner is not effective in all cases, but the focus of this work is on efficient operator evaluations and not on developing efficient preconditioning techniques.

4.2.4 Discretisation of the Helmholtz operator

We now give a brief overview of the discretisation of the Helmholtz equation, which is described further in reference [63]. The previous algorithm is applicable to any spatial discretisation, but the size, structure and numerical properties of the matrices will depend on our choice of discretisation. Here we will be using a continuous Galerkin projection over triangular elements with a hierarchical modal basis. We note that the central aspect of using a matrix-free scheme is avoiding the assembling of the complete global matrix of

the Helmholtz operator. Instead we are breaking down the global operator into a sum of elemental operators. This implies that the global coefficient vector $\hat{w}_g$ needs to be mapped first to the element local coefficients in a scatter-type operation:

$$\hat{w}_l = \boldsymbol{\mathcal{A}}\hat{w}_g \,. \tag{4.27}$$

Since the global-to-local matrix $\boldsymbol{\mathcal{A}}$ is very sparse, it is implemented using a mapping vector v_{map} and a sign vector v_{sign} of length n_{local} as

$$\hat{w}_l[i] = v_{sign}[i]\ \hat{w}_g[v_{map}[i]] \quad \text{for } i \in [0 : n_{local}) \,. \tag{4.28}$$

On each element e the elemental Helmholtz operator can then be applied as

$$\hat{s}_l^e = \boldsymbol{H}^e \hat{w}_l^e \quad \text{for } e \in [0 : n_{el}) \,. \tag{4.29}$$

Finally, the local coefficients need to be mapped back to the global coefficients in a gather-type operation:

$$\hat{s}_g = \boldsymbol{\mathcal{A}}^T \hat{s}_l \,. \tag{4.30}$$

Equivalently to the global-to-local operation this is implemented as

$$\hat{s}_g[v_{map}[i]] = \hat{s}_g[v_{map}[i]] + v_{sign}[i]\ \hat{s}_l[i] \quad \text{for } i \in [0 : n_{local}) \,. \tag{4.31}$$

Here it can be seen that boundary modes of neighbouring elements need to be added. In a parallel implementation this basic implementation can lead to a race-condition.

The elemental Helmholtz operation is the combination of the Laplacian operator and the mass operator and can be expanded as

$$\hat{s}_l^e = \boldsymbol{H}^e \hat{w}_l^e \tag{4.32}$$

$$\hat{s}_l^e = [\boldsymbol{L}^e + \lambda \boldsymbol{M}^e] \hat{w}_l^e \,. \tag{4.33}$$

Using the matrix notation for a 2D element this can be further expanded as

$$\hat{s}_l^e = [(\boldsymbol{D}_{x_1}\boldsymbol{B})^T \boldsymbol{W} \boldsymbol{D}_{x_1}\boldsymbol{B} + (\boldsymbol{D}_{x_2}\boldsymbol{B})^T \boldsymbol{W} \boldsymbol{D}_{x_2}\boldsymbol{B} + \lambda \boldsymbol{B}^T \boldsymbol{W} \boldsymbol{B}]\ \hat{w}_l^e \,. \tag{4.34}$$

The basis matrix $\boldsymbol{B}$ with $\boldsymbol{B}[m(i,j)][n(p,q)] = \phi_{n(p,q)}(\xi_{m(i,j)})$ denotes the backward-transform from transformed space to physical space, where $\xi_{m(i,j)}$ denotes a set of quadrature points on this element and $\phi_{n(p,q)}$ are basis functions defined on the reference element. The function $n(p,q)$ is an ordering of the entries in directions p and q to a vector consistent with the local degrees of freedom in physical space $\boldsymbol{u}$ on each element. Similarly, $m(i,j)$ is an ordering of the entries in directions i and j to a vector consistent with the local degrees of freedom in transformed space $\hat{\boldsymbol{u}}$ on each element. We note the use of a hierarchical modal basis gives rise to independent basis functions so that $\phi_{n(p,q)}(\xi_{m(i,j)}) = \psi_p(\xi_{1i})\psi_q(\xi_{2j})$. The

diagonal weight matrix $\boldsymbol{W}$ with $W_{m(i,j)} = J_{m(i,j)} w_i w_j$ includes the Jacobian determinant J (with the Jacobian being the mapping between the standard element and the curvilinear element on coordinates x), as well as the quadrature weightings w_i and w_j in each of the coordinate directions. Following [63], we select a Gauss-Lobatto-Legendre quadrature for w_i and a Gauss-Radau quadrature for w_j to mitigate the effects of the singularity in the collapsed coordinate Duffy mapping. The derivative matrices $\boldsymbol{D}_{x_1}$ and $\boldsymbol{D}_{x_2}$ are

$$\boldsymbol{D}_{x_1} = \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_1} \right) \boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_1} \right) \boldsymbol{D}_{\xi_2} \tag{4.35}$$

$$\boldsymbol{D}_{x_2} = \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_2} \right) \boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_2} \right) \boldsymbol{D}_{\xi_2} \,, \tag{4.36}$$

with the diagonal coefficient matrices $\boldsymbol{\Lambda}$. The block-diagonal derivative matrices $\boldsymbol{D}_{\xi_i}$ are

$$\boldsymbol{D}_{\xi_i} = \frac{\partial}{\partial \xi_i} \,. \tag{4.37}$$

To minimise the operator count for an efficient CPU implementation, the elemental matrices $\boldsymbol{W}$ for the mass operator have been precomputed and stored. For the elemental Laplacian operators, the following diagonal matrices have been precomputed and stored:

$$\boldsymbol{L_{11}} = \boldsymbol{W} \left(\boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_1} \right) \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_1} \right) + \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_2} \right) \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_2} \right) \right) \tag{4.38}$$

$$\boldsymbol{L_{12}} = \boldsymbol{W} \left(\boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_1} \right) \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_1} \right) + \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_2} \right) \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_2} \right) \right) \tag{4.39}$$

$$\boldsymbol{L_{21}} = \boldsymbol{L_{12}} \tag{4.40}$$

$$\boldsymbol{L_{22}} = \boldsymbol{W} \left(\boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_1} \right) \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_1} \right) + \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_2} \right) \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_2} \right) \right) . \tag{4.41}$$

By breaking down the discrete elemental Helmholtz operator in the described fashion, the construction of the elemental matrix operator $\boldsymbol{H}^e$ is avoided and thus memory bandwidth is reduced and algorithmic intensity is increased.

Where possible, we additionally exploit the tensor-product construction of the C^0 basis to apply the sum-factorisation technique [93]. This is a widely used strategy for attaining substantial reductions in operator count for tensor-product operations.

In our implementation, we therefore consider an application of the Helmholtz operator in a matrix-free fashion that utilises these sum-factorisation kernels for each of the components of the Helmholtz equation above, rather than storing the elemental matrices $\boldsymbol{H}^e$. More details of this technique applied to the spectral element method on triangles and other higher dimensional shape types can be found in references [84, 63, 15].

4.3 Implementation and parallelisation strategy

We commence this section with a discussion of the employed portable programming models and their compiler support in Section 4.3.1.

In the following, we discuss the individual components of the method, as introduced in Section 4.2. For each one we develop an individual parallel kernel, that ideally performs well on both CPU and GPU architectures. As will be seen, this is achieved apart from the *gather*-type reduction kernels, see Section 4.3.3.

The first kernel implements the AXPY operations specified by Equation 4.18 to Equation 4.22. All three programming models, *Kokkos*, *OpenMP*, and *OpenACC* have a suitable syntax and implementation for these *parallel-for-loop*-type of equations. The same applies to the *scatter*-type operation, specified by Equation 4.28.

The *gather*-type operation specified by Equation 4.31 cannot readily be parallelised without causing a race-condition, as multiple local modes are added up to one global mode and hence need to write to the same memory space. How we still achieve reasonable parallel performance is discussed in Section 4.3.3.

The calculation of the conjugate gradient step sizes α and β and the residual ϵ, as given by Equation 4.25, Equation 4.24, and Equation 4.26 are classical reduction operations. These can be parallelised using the *parallel-reduce*-type syntax. Even though all three programming models support this operation, the *llvm-ykt* compiler branch for *OpenMP4.0* did not support it at the time of conducting this work. Instead, we wrote our own kernel for parallel-reductions. This kernel is based on a two level-implementation as follows. A first level that spawns multiple blocks to utilise the individual streaming multiprocessors of the GPU, so that each reduces a chunk of the overall array down to one scalar. The second level then takes all these intermediate results and performs another reduction on a single SMX to yield the final reduced scalar. We use zero-padding of the arrays on both levels to improve the performance. Although this custom kernel does not achieve the optimised performance of an optimal BLAS library implementation, it does not compromise the overall performance of the method. This will be shown in the results section (see Section 4.4).

The computationally most demanding part is the evaluation of the elemental Helmholtz operator, as given by Equation 4.33. A careful evaluation of the parallel algorithm is hence performed in Section 4.3.2.

4.3.1 *Kokkos* versus *OpenMP* versus *OpenACC*

For this work we have identified three programming models that can be added to our existing *Nektar++* framework. As a requirement, they needed to support both CPU and GPU architectures to expose their parallel capabilities, have open-access compiler support and a syntax that allows seamless implementation in an existing *C++* codebase. Other portable models, that were presented in Section 2.2, like *OpenCL* [67], *SYCL* [69], *RAJA* [54], or *OCCA* [79] were not considered due to their less mature ecosystems or poor

early performance results.

Kokkos [17] is a library consisting of a performance portable abstraction layer in *C++*, with *OpenMP* [92] and *Pthreads* [111] backends for multi-core CPUs and a *CUDA* [88] backend for Nvidia GPUs. At the time of conducting our study it only supported data- and no task-parallelism, which prevents load sharing between host and device. The compiler support is very wide, spanning the *gcc*, *clang-llvm*, and *pgi* open-access compilers. The *Kokkos* syntax is based on *Kokkos* data arrays, that map the data between CPU and device memory. Its unique quality is the polymorphic layout of the 2D arrays, that can automatically switch between row- and column-major layout, to achieve coalesced memory access on different architectures. We are going to exploit this feature within this chapter.

OpenMP [92] is a programming model based on compiler directives or so called *pragmas*, that can be added to legacy codebases and specify parallel constructs. Initially it only supported parallel executions on multi-core CPUs, but offloading to target devices is supported from the *OpenMP4.0* specification onwards. The specification supports both task- and data parallelism, which would allow for a real heterogeneous execution and load sharing between both host and device. The compiler support for the target offloading, however, is very sparse. The *gcc-7* compiler supports only the most basic *parallel-for* construct. The most mature open-access compiler has been the experimental *clang-ykt* compiler branch [41]. We will show that, whilst it was possible to implement, compile and execute our benchmark problem using this compiler, it is still lacking crucial features that prevent its enrolment within a complete framework like *Nektar++*.

OpenACC [90] is a similar programming model to *OpenMP* that is based on compiler directives, but primarily targeted at offloading to devices. It supports both task and data parallelism. The *gcc-6* compiler support very simple offloading directives and is hence not complete enough for our purpose. The *pgi* compiler, however, that supports Nvidia GPUs has a very mature ecosystem and will be employed in this work. More recently not only a host fallback is supported by the *pgi*-compiler, but also a proper multi-core CPU target. Without using a combined programming model of *OpenACC* for the device and *OpenMP* for the host parallelism, no heterogeneous execution on host and device is possible.

4.3.2 Interleaved data structures and kernels

Each of the elemental Helmholtz operations (see Equation 4.33) can be performed independently and thus in parallel. On a CPU without vectorisation, we would assign one thread per element or set of data, but on the GPU, there are multiple options. These are primarily explained by the constraint of GPUs and the *CUDA* programming model, which forces all 32 *CUDA* threads in one warp to perform the same mathematical or logical operation in any one cycle. Otherwise the warp will diverge so that the different operations are executed sequentially over multiple cycles, decreasing the performance. Three different work distributions can be distinguished here:

(a) Each *CUDA* kernel or GPU processes one set of data.

(b) Each *CUDA* block or streaming multiprocessor (SMX) processes one set of data.

(c) Each *CUDA* thread or core processes one set of data.

Option (a) is only suitable for a single, very large operation and hence can be discarded for our purpose here. On the Nvidia P100 GPU for example, a matrix size of 128×16384 on a `sgemm` kernel is necessary to get more than 50% of the peak performance of *CuBLAS* `sgemm` operations. Option (b) is mostly suited for operations on rather few, but large sets of data. The individual sets might vary in size and might undergo different operations, without inducing performance breakdowns. Option (c) is only suited for operations on many, small sets of data. For good performance, however, the individual sets should all have the same size and undergo the exact same operations.

In option (b), vectors or matrices need to be sufficiently large so that the overall operation can be efficiently split into multiple warps, in which all threads perform the same elemental mathematical operation. This is because in order to be fully occupied, each SMX of a Nvidia GPU needs to schedule 32 or 64 warps at the same time. Operations over multiple *CUDA* blocks are not synchronised, so that different operations can be executed without performance penalty, as long as a good load balancing between the different *CUDA* blocks is ensured. In terms of data structures, option (b) can operate efficiently on data that is arranged one set after the other. For matrix–matrix multiplication, techniques like *tiling* are further employed to split the overall operation efficiently into elemental operations on which each thread in a warp operates on coalesced data. This option is commonly implemented as *batched* or *strided* BLAS operations. Both these operations are collections or batches of BLAS operations that are performed concurrently, so that the batches will be automatically distributed over the individual SMX. As a special case, strided operations assume the same lengths of data arrays across all batches. The *MAGMA* library [22], the Intel *MKL* library, and the *CuBLAS* library [104] all support batched `gemm` operations, the latter also strided batched operations. Again on the Nvidia P100, 128 batched single precision matrix multiplications with matrix size of 128×128 will result in about 50% peak performance. This is already an improvement from individual `sgemm` kernels, but lower polynomial order elemental matrix sizes are still far smaller than 128×128. E.g. a `gemm` operation occuring in the sum-factorisation kernel of a triangular element with polynomial order 10 has matrix sizes of 11×12.

In option (c), all data sets need to have the same size and undergo the same overall operation, so that 32 threads can always be efficiently combined into a warp. Here a sufficiently large number of the small data sets is required to fully occupy the number of concurrent warps that need to be scheduled. For this approach however, the naive memory layout of having data sets consecutively in memory would violate any coalesced data access. This is because the specific data entries of each set are separated by the whole length of each data set in this case. Yet, all these specific data entries have to be loaded for the warp operations at the same time. For an efficient memory access, it follows that all specific data entries of a group of sets need to be coalesced. The group of sets

Listing 4.1: Custom interleaved `dgemm` syntax and algorithm

```
int interleavedDgemm(char transa, char transb,
        int M, int N, int K,
        const double alpha,
        const double *A, int lda, int strideA, int instA,
        const double *B, int ldb, int strideB, int instB,
        const double beta,
              double *C, int ldc, int strideC, int instC)
{
    // one of four cases shown
    if (transa == 'T',transb == 'N') {
        for (int m = 0; m < M; ++m) {
            for (int n = 0; n < N; ++n) {
                double c_mnp = 0;
                for (int k = 0; k < K, ++k)
                    c_mnp += A[(m + k*ldA) * strideA + instA]
                    * B[(k + n*ldB) * strideB + instB];
                C[(m + n*ldC) * strideC + instC] =
                alpha * c_mnp +
                beta * C[(m + n*ldC) * strideC + instC];
            }
        }
    }
}
```

is chosen as a multiple of the warp-size of 32 threads. As a result, the data sets of one such group become interleaved. The distance or stride between following entries of one data set is equal to the number of data sets that are combined in one group. Interpreting the simple data layout in which one data set is stored after each other, corresponds to a stride of one. For CPUs, this concept of explicit vectorisation over multiple elements using interleaved memory layouts has also been implemented in the *DUNE* library [9], and the *deal.II* library [72, 1] to benefit from the wider vector lanes.

For level-1 and level-2 BLAS operations, stride can be specified as a variable, so that no major code adjustment is necessary. Standard level-3 BLAS operations like `dgemm` which are used for our sum-factorisation kernels, however, do not possess a stride variable, so that we decided to implement a custom strided `dgemm` function, that can operate on the interleaved memory layout. Following the function call syntax of other strided batched `gemm`, we introduced two additional variables for each of the three data arrays representing the 3 sets of matrices. The two variables are the stride between the following entries, which is equal to the size of the group, and a variable denoting the number of the group a specific thread operates on. The function call syntax and the operations the algorithms of our custom `dgemm` kernel is given in Listing 4.1.

Having developed an efficient and working implementation for option (c), and because the performance of option (b) would be severely limited due to the small matrix sizes considered here, we proceed with option (c) throughout this work.

Algorithm 4.1 Initial local-to-global reduction algorithm

```
1: procedure LocalToGlobalInitial(ŝ_g, ŝ_g, v_sign, v_map)
2:     for all global coefficients j do in parallel
3:         ŝ_g[j] ← 0
4:     end for
5:     for all elements e do in serial
6:         for all coefficients c do in parallel
7:             i ← e * n_coeffs + c
8:             ŝ_g[v_map[i]] ← ŝ_g[v_map[i]] + v_sign[i] * ŝ_l[i]
9:         end for
10:    end for
11: end procedure
```

There is another limitation of using a batched function as in option (b), because a new kernel needs to be scheduled for each of these BLAS functions. This is not ideal in conjunction with our framework, in which the code is structured in a way that describes the operations to be executed for each element or data set. Ideally, the data will be loaded into the cache at the beginning of such extended kernels and then operated on by a sequence of BLAS functions and, at the end, stored back into global memory. Using *CuBLAS* functions that start a kernel for each BLAS function will require multiple memory copies between global memory and cache, compromising efficiency. Further, it would require to re-factor the code so that the extended kernel is broken up into chunks corresponding to one BLAS function each.

We further found that explicitly using the interleaved memory layout and strided kernel has only been strictly necessary for kernels in which dynamic memory allocation within the kernel is specified. This is the natural way to write with the *Kokkos* syntax and the natural extension from our initial CPU code. The variable length of arrays depending on the element polynomial order can more readily been accounted for using dynamic memory allocation. However, the *OpenMP*-target compiler did not cover this functionality and the *OpenACC* implementation was poorly performing. Here we had to write our kernels with static memory allocation and use templating to account for the varying array lengths of varying polynomial orders. We found that in this case, the compiler optimisation can implicitly deal with this situation and create an interleaved memory layout.

4.3.3 Parallel element colouring and reduction operations

The local-to-global mapping, as given in Equation 4.31 is an operation that is not trivial to parallelise. This *gather*-type or reduction operation can be expressed as a double for-loop, as in Algorithm 4.1. The inner loop over all local coefficients or modes can be implemented in parallel, and the outer loop over all elements needs to be in serial to avoid a race-condition.

Although we found this algorithm to perform very well on the CPU, its performance was limited on the GPU since not even a single SMX could be fully utilised even at

Algorithm 4.2 Parallel local-to-global reduction algorithm using element colouring

```
1: procedure LocalToGlobalParallel(ŝ_g, ŝ_g, v_sign, v_map, C)
2:     for all global coefficients j do in parallel
3:         ŝ_g[j] ← 0
4:     end for
5:     for all coloursets c ∈ C do in serial
6:         for all elements e ∈ c do in parallel
7:             for all coefficients c do in parallel
8:                 i ← e * n_coeffs + c
9:                 ŝ_g[v_map[i]] ← ŝ_g[v_map[i]] + v_sign[i] * ŝ_l[i]
10:            end for
11:        end for
12:    end for
13: end procedure
```

reasonable element polynomial orders. To enable more parallelism we introduced an initial elemental colouring of the mesh. This ensures that no neighbouring elements are in the same colourgroup, and hence no write conflict of multiple local modes to global modes can occur. With this colouring in place, it is now safe to parallelise the for-loop over all elements within one colourgroup and introduce a third outermost serial loop over all colourgroups, as in Algorithm 4.2. Our colouring algorithm uses a re-balancing step so we yield colourgroups that are typically within 25% of the average group size.

The algorithm requires a global synchronisation between the different colourgroups, which can be realised using two options on the GPU. This is either launching a new kernel per colourgroup, or launching a single kernel for all colourgroups, but only utilise one SMX, since it is not possible to synchronise between SMX in a single kernel. The first option comes with a greater overhead, whereas the second utilises only a fraction of the GPU. For reasonably large mesh sizes, each colourgroup kernel receives enough work so that the first option will be more performant. Another challenge occurs as the inner parallel-loop is relatively skinny and the outer parallel loop is relatively fat, which does not map well to the GPU hardware. However, since the two loops are perfectly nested, a good compiler implementation will be able to collapse the two loops, resulting in better performance.

For all CPU executions we utilise the initial version of Algorithm 4.1, for all GPU executions we utilise the mesh-colouring approach and the parallel version as in Algorithm 4.2.

4.3.4 Usability and maintainability of the programming models

All three tested programming models, *Kokkos*, *OpenACC*, and *OpenACC* advertise their capability to incrementally parallelise and port legacy applications written in *C++* to multicore CPUs and GPUs. While this is certainly the case, we noted a few obstacles that indicate this process is not as seamless. In fact, care has to be taken when the initial code-base features many object-oriented programming structures. While they can

be parallelised over the cores of a CPU, they do not map to the *CUDA* programming model, that underlies all GPU backends of the considered programming models.

For example, with *CUDA* it is not possible to parallelise over an array of element objects, in which each object holds its data as member-variables and calls its overloaded member-functions, depending on which type of element it is. In this case, the functions and hierarchies of objects need to be de-tangled and implemented in a more straightforward *C*-syntax, in which the corresponding variables and functions for each element are taken care of explicitly. This can result in a complete re-writing of the involved compute-heavy kernels.

Similarly, encapsulated data-structures cannot be mapped directly to the GPU memory. Instead the raw pointers would need to be passed, which in turn requires all functions further down the call-tree to be adapted for the plain data-type. *Kokkos* uses its own encapsulated data-arrays, that map between different memory spaces. It follows that, if in the case of *Nektar++*, encapsulated data-arrays are employed, a large initial implementation effort is necessary for all three programming models. If the legacy application already uses plain *C* arrays, *OpenMP* and *OpenACC* would be favourable.

The limitation of using dynamic memory allocation within the kernels also requires the introduction of templating for variable array lengths. This is a further step away from initial OOP-oriented *C++* code-bases.

When it comes to the actual syntax of specifying parallel loops, the three programming models are comparable, but the *Kokkos* syntax is slightly more complex than that of the *OpenMP* and *OpenACC* compiler directives. *Kokkos* allows for the possibility to explicitly specifying an array to be placed into shared memory, which can be rather cumbersome to optimise when dealing with different element polynomial orders and attempting to align this placement with the shared memory size restriction. *OpenACC* can place arrays into shared memory implicitly, whereas the tested *OpenMP-clang-ykt* implementation does not cover this functionality. In addition, the *OpenMP* and *OpenACC* syntaxes are so similar that a change between them should be straightforward.

4.4 Performance evaluation

Our test case involves the solution of the Helmholtz equation

$$\nabla^2 u - \lambda u = -(\lambda + 2\pi^2)\cos(\pi x)\cos(\pi y)\,, \tag{4.42}$$

with $\lambda = 2.5$, on a square domain $\Omega = \{(x, y) : -1 \leq x \leq 1, -1 \leq y \leq 1\}$. On the sides $x = -1$ and $x = 1$ we apply the Dirichlet boundary condition

$$u = \cos(\pi x)\cos(\pi y)\,, \tag{4.43}$$

and the Neumann boundary condition

$$\frac{\partial u}{\partial n} = \pi \cos(\pi x) \sin(\pi y) \tag{4.44}$$

is imposed on the sides $y = -1$ and $y = 1$, with n being the boundary normal vector. We note that the solution to this system is an eigenfunction. It follows that the conjugate gradient method, being a Krylov subspace method, should find the exact solution within the first iteration. However, since we are applying a preconditioner, this property does not manifest itself here.

We compare the same algorithms as presented in Section 4.2 that solve this equation implemented with three different programming models: *Kokkos*, *OpenMP* and *OpenACC*. We use the same codebase for both CPU and GPU applications and only at compile time specify the relevant compile flags for each architecture. The only algorithmic difference between architectures is the parallelism in the local-to-global reduction operation discussed in Section 4.3.3.

The individual meshes for the spatial discretisation consist of triangular straights-sided elements, but different element polynomial orders between 2nd and 11th order. The number of degrees of freedom in each mesh is fixed at 4 million, so that the meshes consist of 66k to 1975k elements from highest to lowest order respectively. We iterate the conjugate gradient loop until the residual is converged to an absolute tolerance of $\epsilon < 10^{-9}$. The polynomial order, number of elements, degrees of freedom, and iteration counts until convergence are provided in Table 4.1 for all ten considered meshes. It can be observed that meshes with higher polynomial order lead to a faster convergence, as expected for high-order spectral-*hp* element methods.

Element order	Elements	DOF	Iterations
2^{nd}	1,975,328	3,954,113	3,577
3^{rd}	875,778	3,944,458	2,432
4^{th}	493,212	3,949,153	1,906
5^{th}	315,840	3,951,461	1,491
6^{th}	220,138	3,965,941	1,488
7^{th}	161,028	3,965,941	1,224
8^{th}	122,580	3,926,017	1,182
9^{th}	97,452	3,950,263	1,066
10^{th}	78,940	3,950,461	970
11^{th}	66,220	4,009,787	874

Table 4.1: Mesh metrics for the triangular meshes of different polynomial order, including degrees of freedom (DOF) and number of iterations within the conjugate gradient solve to converge to a residual of $\epsilon < 10^{-9}$.

In terms of compiler toolchains, for our *Kokkos* implementation we use the GNU compiler *gcc-5.4.0* with optimisation flag -O3. For the *OpenACC* implementation we use the PGI compiler version *pgc++-18.4* with optimisation flag -O3. The *OpenMP*

implementation is compiled with the experimental *llvm-clang* compiler branch *clang-ykt* and optimisation flag -O3.

At the time of conducting this work the *clang-ykt* branch has not been feature complete. Even though the *OpenMP4.0* specifications support *parallel-reduce* operations, it has not been implemented, yet. As a workaround, we wrote an own kernel for parallel reductions, that is based on a two level-implementation.

The *llvm-ykt* compiler branch also did not support linking to external libraries at the time of conducting this work. Thus, our *OpenMP*-GPU implementation had to be completely decoupled from the *Nektar++* framework. For a better performance on the CPU we set the environmental variable *OMP_PROC_BIND=true*.

The correctness of our parallel implementations for CPUs and GPUs is ensured by comparing the residual and the number of iterations of the converged solution with a serial CPU benchmark implementation taken from the *Nektar++* framework.

4.4.1 Comparison of *Kokkos*, *OpenMP* and *OpenACC* on multi-core CPUs

For the multi-core system, we use an Intel E5-2690 v0 CPU with 16 cores and a base frequency of 2.9GHz and 64GB RAM with 1600Hz. The peak FP64 performance considering FMA and AVX is 371.2GFLOPs, the maximum memory bandwidth is 51.2GB/s, the thermal design point (TDP) is 135W.

As a benchmark, we also present the results obtained with our basic *MPI* implementation within *Nektar++*. It uses the same algorithms, but realises its parallel execution using domain decomposition.

Strong scaling exercise

As is customary for multi-core CPU applications, we conduct a strong scaling exercise between 1 and 16 threads on a mesh consisting of 7th-order triangles with 161k elements. This allows us to evaluate the performance of each model and identify potential performance overheads. The runtime results for the conjugate gradient solve are given in Figure 4.1. Here we see that the *OpenMP* and *OpenACC* implementations perform better than the *Kokkos* implementation; however their relative performance varies with polynomial order and will be clearer to evaluate in this context in Section 4.4.1. More important is the observation that all three implementations scale very similarly with increasing numbers of threads and do not reach the almost ideal scaling of our *MPI* implementation. This is probably due to the reduction operations that do not allow us to fully utilise all threads continuously.

Efficiency of different polynomial orders

For this test series we utilise all 16 threads on our multi-core CPU. We use meshes with elements of varying polynomial orders where the degree of freedom count is fixed across

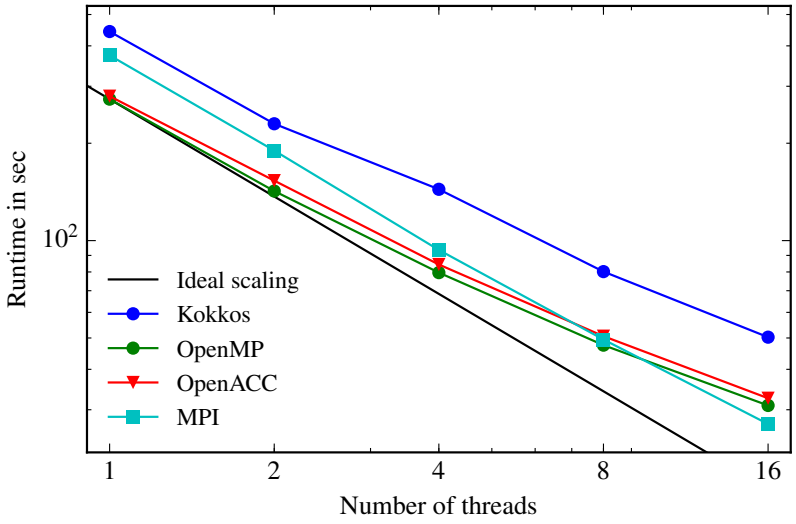

Figure 4.1: Strong scaling of different parallel implementations on a 16-core CPU, based on 7th-order elements.

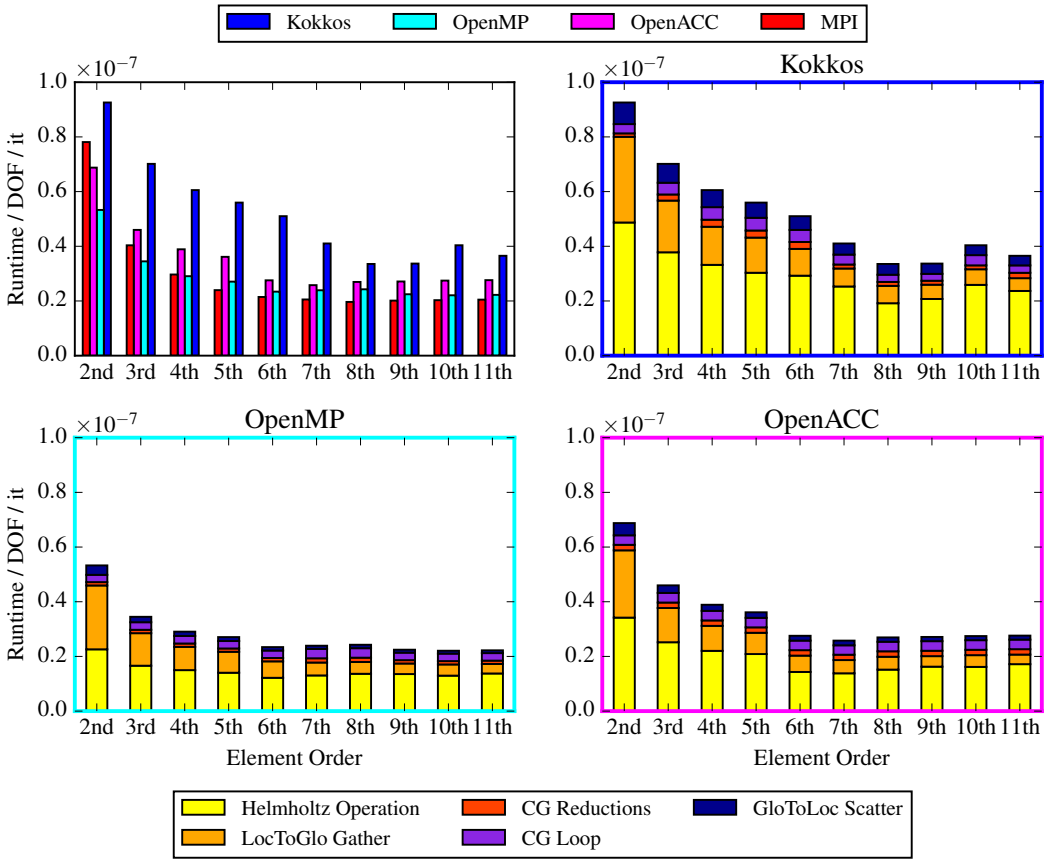

Figure 4.2: Overall runtime comparison and runtime splits between kernels of different parallel implementations over a range of elemental orders on a 16-core CPU.

the polynomial orders. For the runtime measurements we consider the time that the solver spends within the conjugate gradient iteration loop and scale it with the number of iterations and the exact number of DOFs of the underlying mesh, which are presented in Table 4.1. The scaled runtimes are given in the upper left subplot of Figure 4.2 for all four applications. The results improve with increasing element order up to about 6th order and then plateau. The low element orders have a lower FLOP count per DOF, but also a lower FLOP-per-byte ratio or arithmetic intensity. As our applications are memory-bound, they do not benefit from a lower FLOP count *per se*. Overall, the *MPI* implementation is the fastest across the range of polynomial orders, but closely matched by the *OpenMP* and the *OpenACC* versions. The *Kokkos* version gives the worst performance generally, with around twice the runtime.

The remaining three subplots of Figure 4.2 present the runtime-splits between the five different kernels of the application, as discussed in Section 4.3. The elemental Helmholtz operation consumes about 50% of the overall runtimes. The local-to-global reduction also shows bad performance for the lower elemental orders, as the involved arrays are too short in this case. Overall, the splitting of runtime between the *Kokkos*, *OpenMP*, and *OpenACC* is comparable.

Detailed performance evaluation

In order to evaluate how well the underlying CPU hardware is utilised we consider two metrics, the achieved double precision (FP64) FLOPs and the DRAM memory bandwidth. The FLOPs that the different kernels have achieved are given in Figure 4.3. The gather and scatter operations, as well as the conjugate gradient reductions and for-loops perform very poorly, because they are strongly memory bound. Only very few operations are performed on each data element, which is read and written back to DRAM memory. Only for the parallel reduction kernel data can be re-used from higher cache levels, leading to faster loading to registers. The gather and scatter operations perform especially poorly, since the data is usually not read and written in continuous cache blocks. The Helmholtz operation is the critical kernel and achieves much better performance of up to 38GFLOP/s for the *OpenMP* version, 31GFLOP/s for the *OpenACC* version, but only 23GFLOP/s for the *Kokkos* version. For the *OpenMP* version, this translates to utilising more than 10% of peak FLOPs. The trend for higher FLOPs with higher element orders is very clear, as it requires more compute operations per loaded byte, i.e. a higher arithmetic intensity. The overall FLOPs performance is dominated by the Helmholtz operation kernel, but pulled down by about one third by the less performant kernels.

The DRAM memory bandwidth is given in Figure 4.4. Generally it can be seen from the data that especially the *OpenMP* and to a slightly lower degree also the *OpenACC* implementation utilise the memory more efficiently than the *Kokkos* implementation. As a trend, higher polynomial orders are processed more efficiently, as can be expected due to the larger arrays. The *OpenMP* implementation achieves a combined *read* and *write*

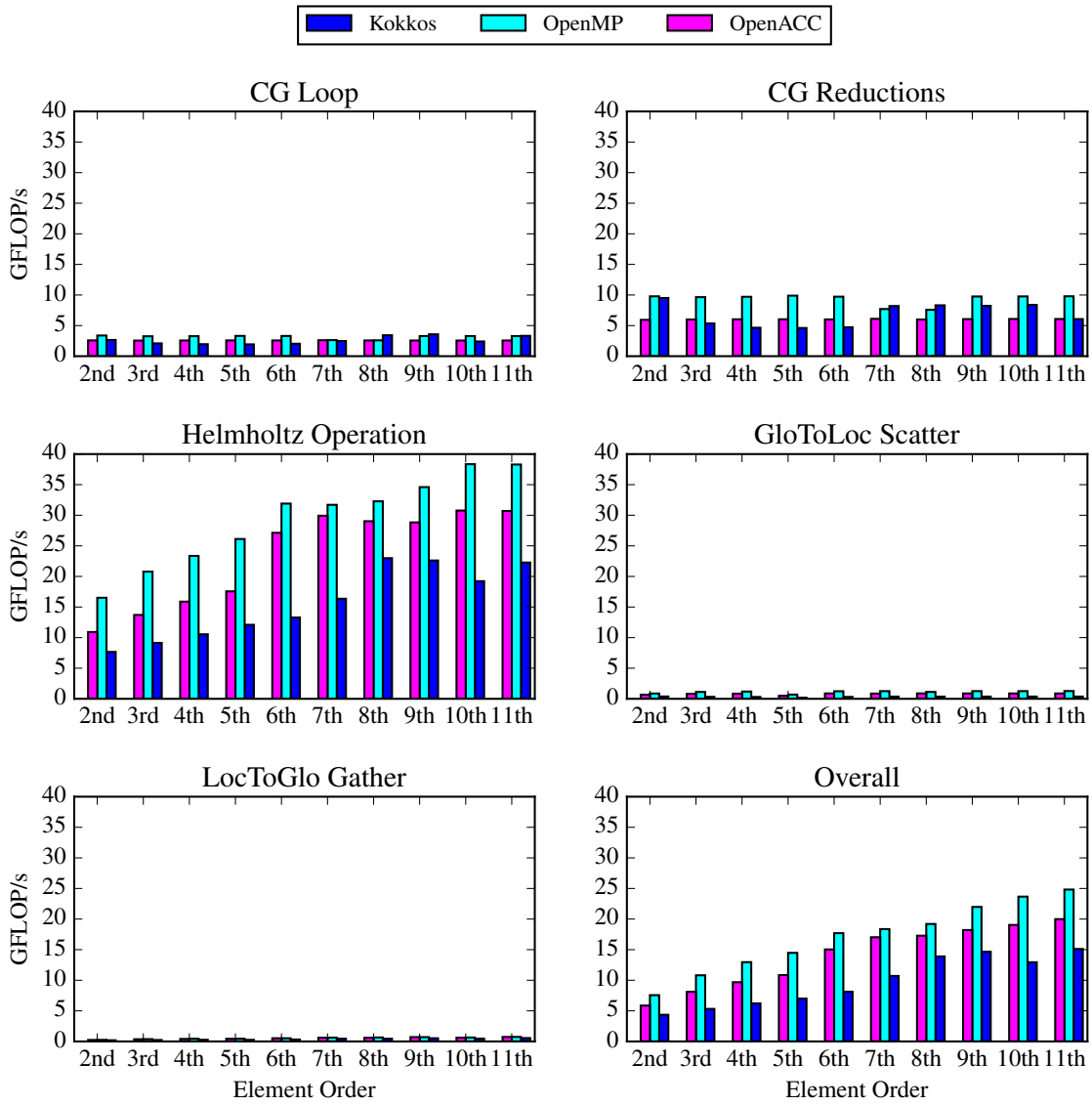

Figure 4.3: Achieved double precision (FP64) FLOPs for the main kernels of three parallel implementations over a range of elemental orders on a 16-core CPU.

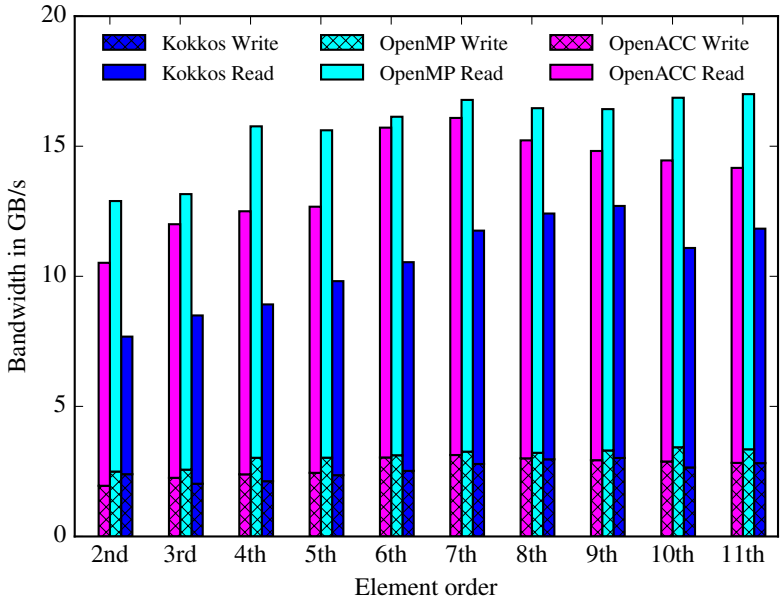

Figure 4.4: Overall achieved DRAM memory bandwidth of three parallel implementations over a range of elemental orders on a 16-core CPU.

memory bandwidth of 17GB/s, which amounts to about 33% of the theoretical bandwidth.

4.4.2 Comparison of *Kokkos*, *OpenMP* and *OpenACC* on GPUs

As the GPU system we use a Nvidia Tesla P100 with 60 streaming multiprocessors (SMX), a peak FP64 performance of 5304GFLOPS, a maximum bandwidth of 549GB/s, and a TDP of 300W.

Efficiency of different polynomial orders

We employ the same test methodology as in the multi-core CPU case in Section 4.4.1. The runtimes of the conjugate gradient solve scaled by the number of iterations and by the exact number of DOFs are given in Figure 4.5. Across the whole range of elemental orders, the *OpenACC* implementation is the most performant, while the variation between elemental orders is low. The runtime splits show a much larger proportion of time spent in the elemental Helmholtz operation, that varies between about 65% for 2nd order and 90% for 11th order. The performance for the *OpenMP* implementation is only about 25% worse for high polynomial orders, but suffers greatly for low polynomial orders. It becomes clear that primarily the bad performance of the local-to-global reduction is responsible and only secondarily the lower performance of our custom-written CG-reduction kernel. The embarrassingly parallel elemental Helmholtz kernel is *on par* with the *OpenACC* version. The performance of the *Kokkos* implementation is about 40% worse than the *OpenACC* implementation across the range of elemental orders. Examining the runtime splits of

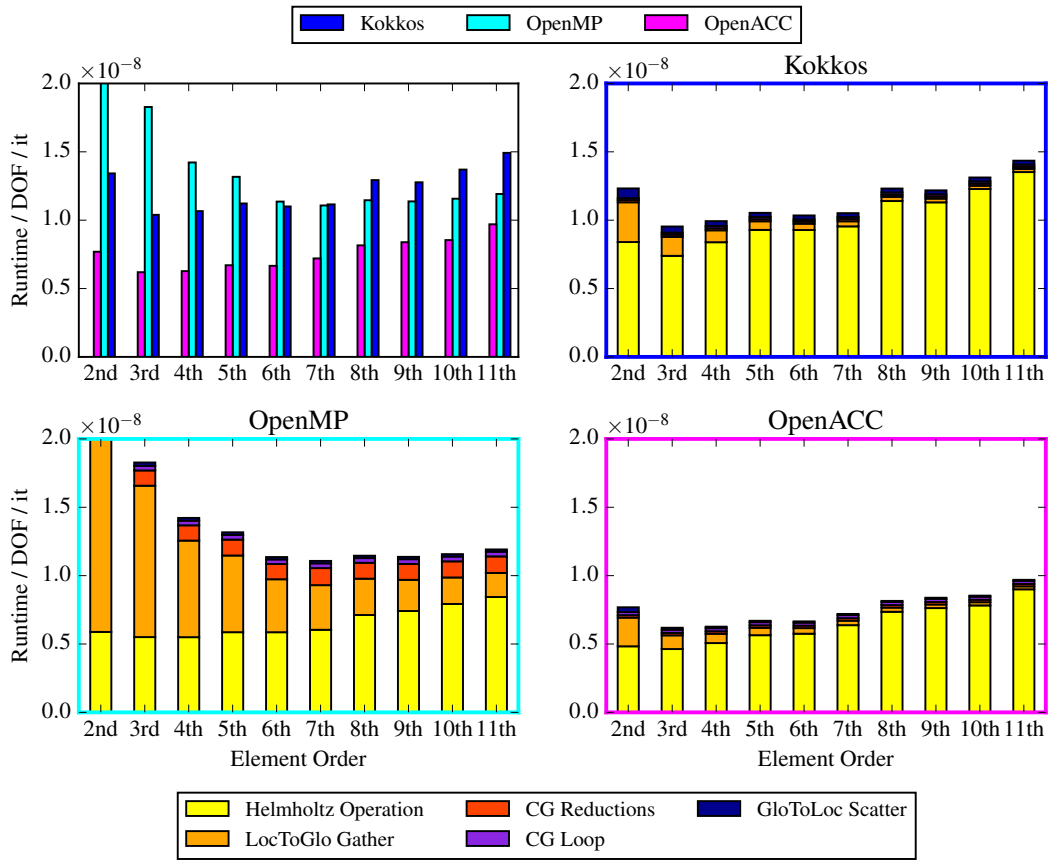

Figure 4.5: Overall runtime comparison and runtime splits between kernels of different implementations over a range of elemental orders on a P100 GPU.

the *Kokkos* version, no individual kernel can be pointed out that would be responsible, indicating a general performance issue with this version.

Detailed performance evaluation

Runtimes alone do not tell us how well the implementation is utilising the underlying hardware. To this end we first provide the achieved double precision (FP64) FLOP/s in Figure 4.6. Comparing the three different programming models, it can be seen again that *Kokkos* does not achieve the same performance as *OpenACC* and *OpenMP* for the Helmholtz operation kernel and hence for the overall application. Furthermore, the poor performance of the *OpenMP* reduction/gather kernels becomes obvious. In general, the achieved FLOP/s translate to utilising actually less than 2% of the theoretical FP64 performance, which indicates that the memory bandwidth is actually the performance bottleneck.

To prove this, we provide the DRAM memory bandwidth in Figure 4.7. Here it can be seen that our implementations operate very close to the theoretical maximum across all elemental orders for the CG-Loop, the CG-reductions and the global-to-local scatter kernels. The most important kernel, the Helmholtz operation, is not fully utilising the

DRAM bandwidth indicating that either the bandwidth of a higher level memory or cache level is limiting the overall performance or the memory latency results in a stall. The three programming models perform very similar, just the *OpenMP* application is falling off slightly.

Comparing the utilisation or bandwidth of the global (Figure 4.8) against the local and shared memory (Figure 4.9) for the Helmholtz operation kernel, we can see that the *Kokkos* kernel is utilising more of the slower global memory (residing on DRAM, L2 and L1 cache) than the faster local and shared memory that resides only on the L1 cache. This could explain the performance deficit of the *Kokkos* implementation. It has to be acknowledged, that a better performance could be achieved if tuning the Helmholtz operation kernel for each elemental order by explicitly specifying the utilisation of L1 cache for certain arrays. This tuning approach however defeats the idea of a performance portable programming model.

It can be concluded that our applications are heavily memory bound. It shows that just porting a legacy CPU-optimised algorithm cannot fully utilise modern GPUs. More effort to reduce the memory dependency of the algorithms and hence increase the arithmetic intensity are necessary.

4.4.3 Cost comparison between architectures

We use system acquisition costs to identify the most efficient architectures in terms of run-time and operational costs.

The acquisition costs are the sum of the CPU or GPU by itself plus a hosting system. We estimate the current price of the Nvidia Tesla P100 GPU as $6000, the Intel E5-2690 v0 CPU with 2 sockets or 16 cores as $800, and the hosting system as $2000. We use a time frame of 3 years for complete linear depreciation, to obtain hourly prices for both the complete CPU and the GPU system. The costs of code execution are then the product of run-time with hourly system costs. This follows the approach presented in Section 3.4.7. The result is given in Figure 4.10. It illustrates very well the benefit of employing GPU systems for the presented type of algorithms, as both the runtime and costs are lower on the P100, the costs by 20%, the runtimes by 72%, considering the most performant cases. Considering only the two *OpenMP* implementations, though, the cost of utilising the P100 GPU is higher than for the CPU, although it is faster. In terms of both cost and time performance metrics, the *OpenACC* version for the GPU is the most beneficial. This result would only become more pronounced, when optimising the algorithms further and replacing pre-computations and loading from memory with on-the-fly calculations to reduce the required memory bandwidth that so far slows down the GPU implementations to a higher degree than the CPU implementations.

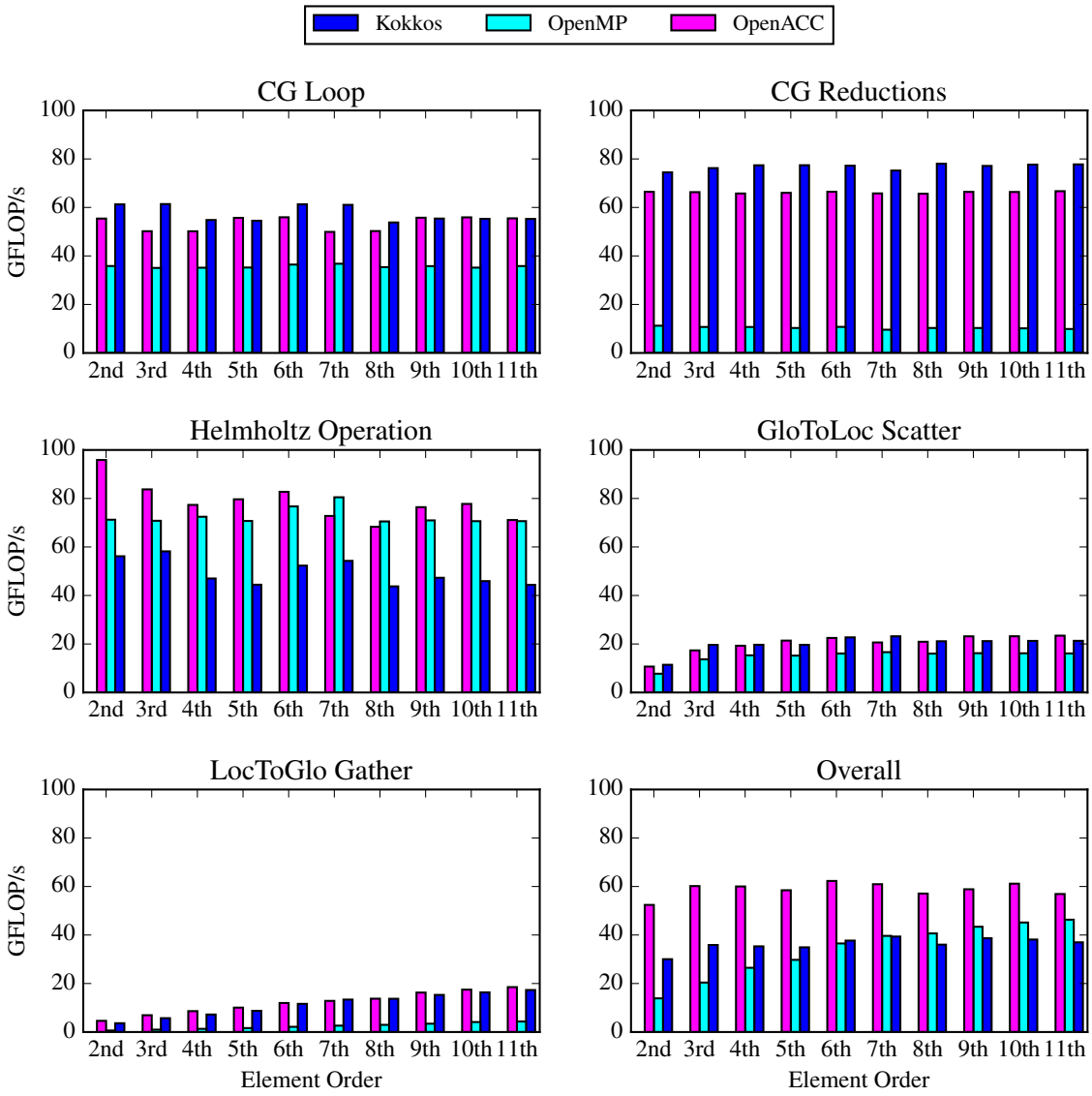

Figure 4.6: Achieved double precision (FP64) flops for the main kernels of three parallel implementations over a range of elemental orders on a P100 GPU.

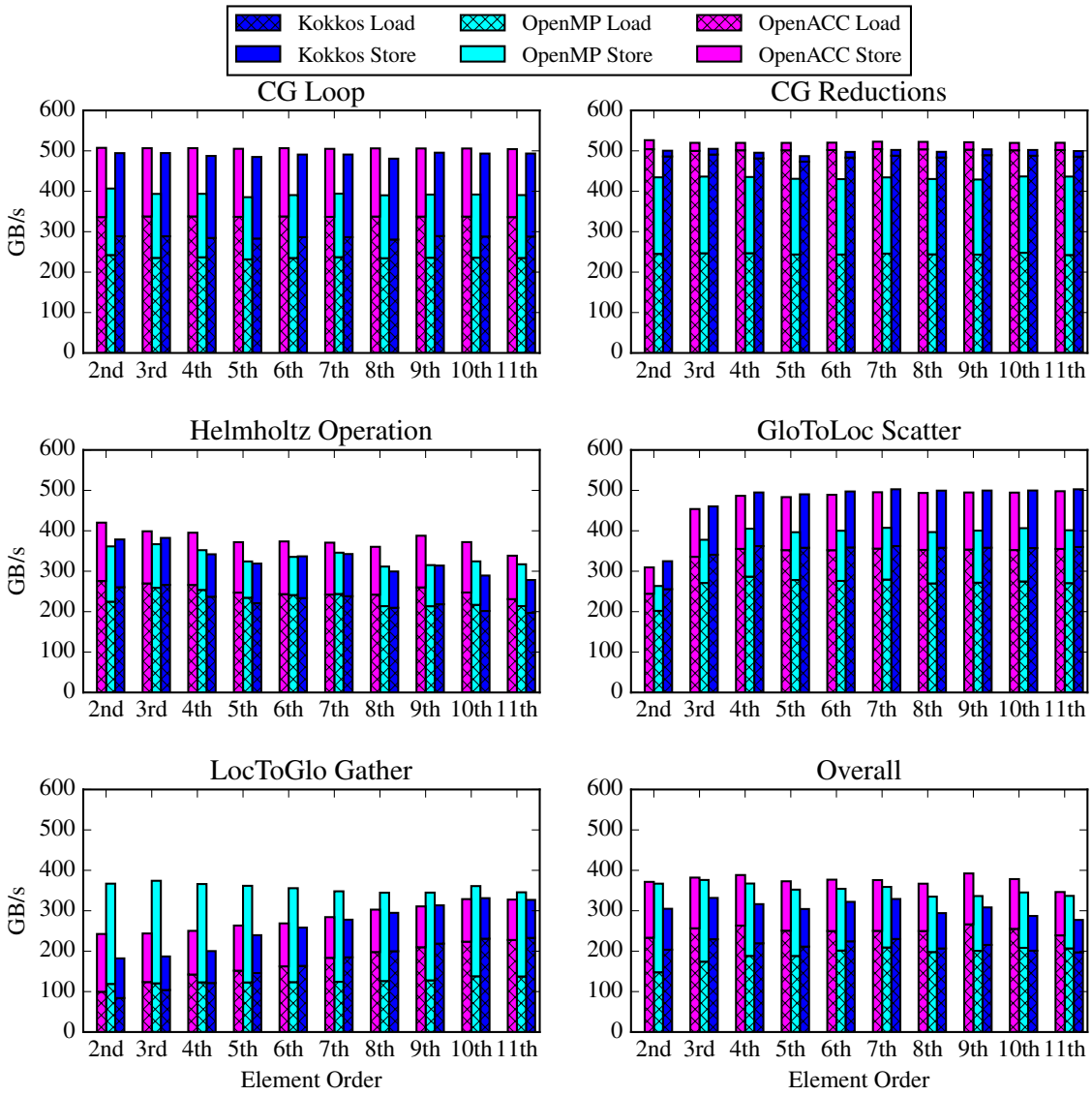

Figure 4.7: DRAM memory bandwidth for the main kernels of three parallel implementations over a range of elemental orders on a P100 GPU.

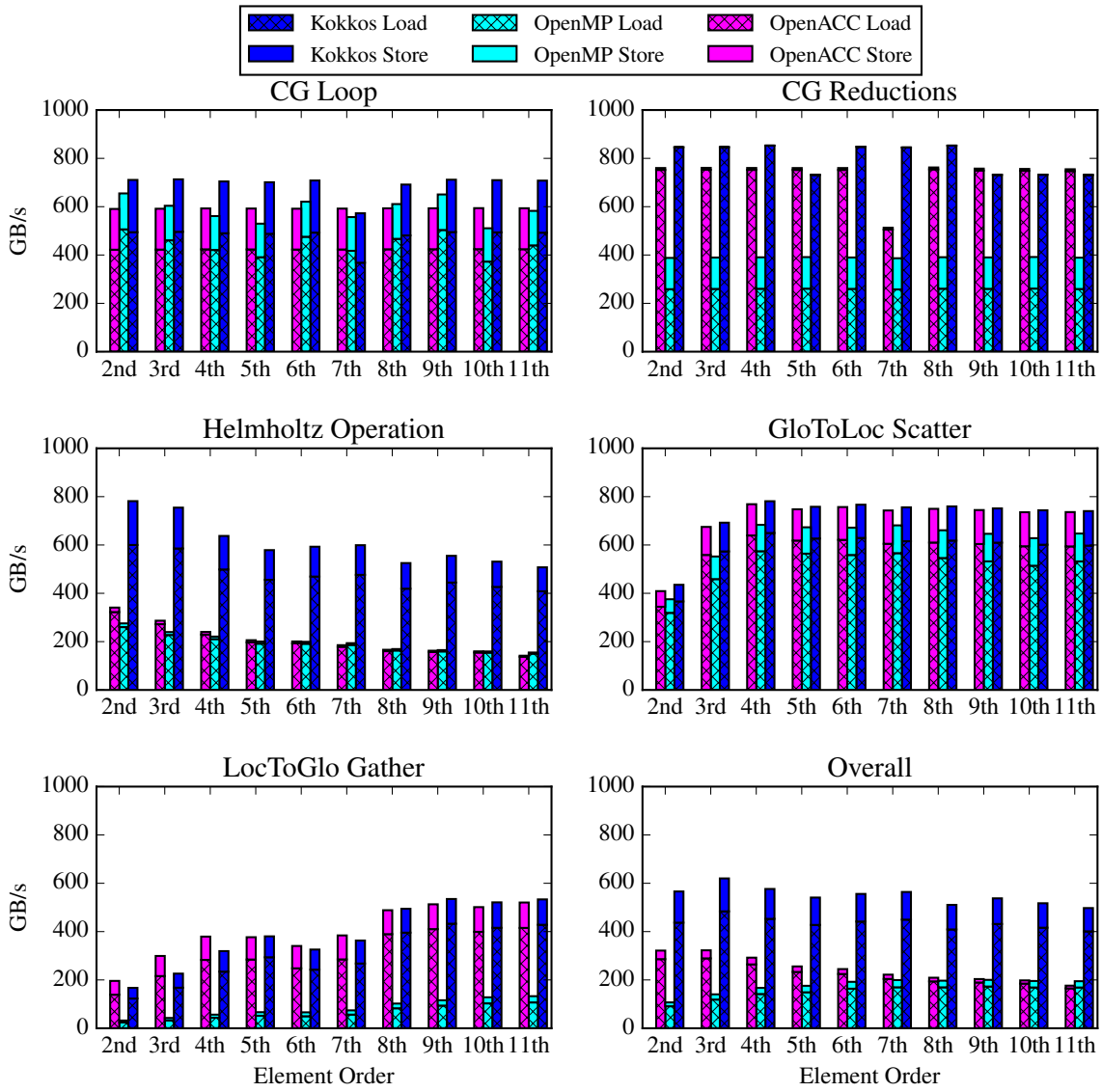

Figure 4.8: Achieved global memory bandwidth for the main kernels of three parallel implementations over a range of elemental orders on a P100 GPU.

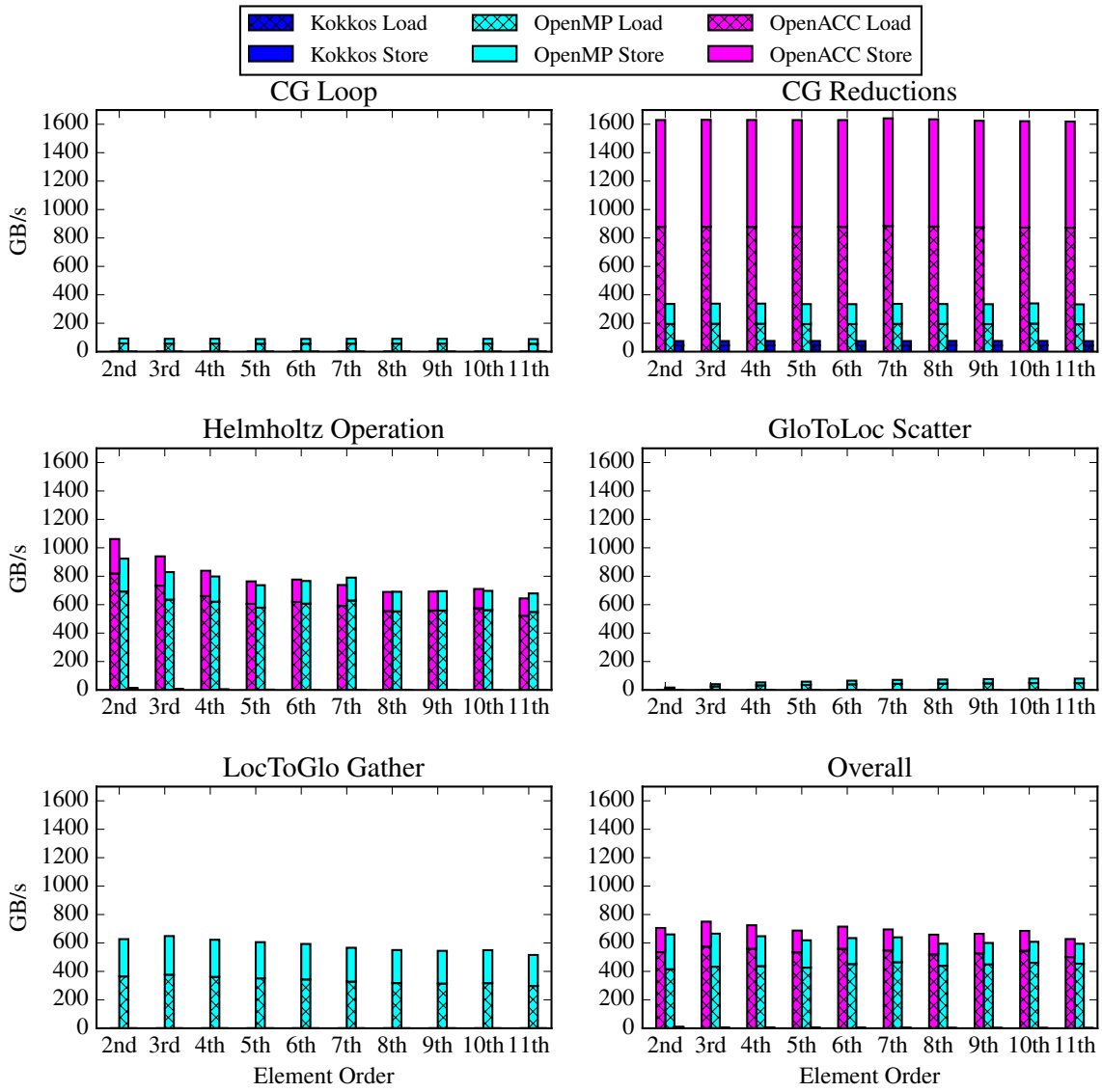

Figure 4.9: Achieved sum of local and shared memory bandwidth of three parallel implementations over a range of elemental orders for the main kernels on a P100 GPU.

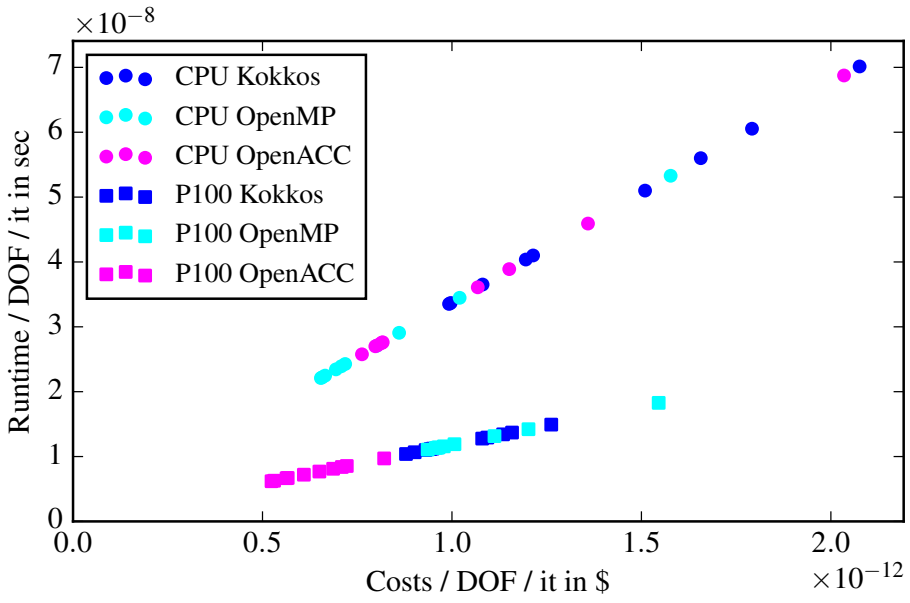

Figure 4.10: Scaled execution costs versus scaled runtimes of three different parallel implementations, measured on a 16-core CPU and a P100 GPU. Each dot corresponds to a different underlying triangular mesh between 2nd and 11th order.

4.5 Conclusions

There are two main conclusions to be drawn from this study.

Firstly, by comparing three different performance-portable programming models, namely *Kokkos*, *OpenACC* and *OpenMP* using open-access compilers in a high-order FEM setting, we gathered performance and usability aspects that help to choose the most suitable programming model:

All three programming models require very similar implementational efforts. There are syntactical differences, which are minor between *OpenMP* and *OpenACC* (and would enable a seamless switch between the two) and slightly deeper compared with *Kokkos*, mostly due to the requirement of using *Kokkos*-specific arrays. *Kokkos* further allows addressing many aspects of performance tuning which are connected with native *CUDA* applications, allowing more individual trade-offs between portability and performance. The main current drawback of the *OpenMP* model is the lack of robust open-access compiler implementations. The best compiler currently available, which is the *llvm-ykt* branch, still lacks a parallel reduction functionality, has no linking support for external libraries, and showed bad performance of the gather-type operation, probably due to the implementation of the *collapsing*-clause and the implicit compiler choice for block-sizes.

The *OpenACC* model has shown the best overall performance results, yielding the fastest runtimes on Nvidia GPUs, and only slightly slower runtimes on Intel CPUs than *OpenMP*.

Secondly, this study has helped us to identify the steps to be undertaken in order to

port a legacy CPU code to modern multi-core CPUs and many-core GPUs and achieve good performance. The mapping of the algorithmic parallelisation to the hardware parallelisation, together with the right data layout, plays a fundamental role in achieving reasonable performance on modern hardware. Here we have employed a very fine grained parallelism in which we process a single mesh element on each thread. This requires the processing of elements to be vectorised either on the AVX vector lanes or over the *CUDA* threads of a warp. This vectorisation in turn necessitates the use of interleaved data structures and custom BLAS functions to operate on them. Reduction operations can become a major bottleneck in massively parallel applications, so we employ mesh colouring to counteract this aspect and regain more parallelism.

The performance of our three applications is very reasonable on multi-core CPUs, but lacks the ability to fully leverage the arithmetic potential of GPUs. To do so, it is not enough to translate the traditional algorithms, that often just minimise FLOP count, as we did in this study. Instead of pre-computing data and loading it when required, more data should be recomputed on the fly, as it can often be faster than loading a large array from DRAM memory, when latencies are of the order of 10^4 cycles. For example, the framework *OpenSBLI* [60] offers the option to variably execute either or a mix of these two approaches.

These observations show that performance portability might not be realisable in practice just by employing a performance portable programming model. The algorithmic choices between current CPUs and GPUs that have to be made now are fundamentally different. To deal with this increasing code complexity, auto-tuning frameworks and just-in-time compilation, that can make algorithmic choices based on the underlying architecture at runtime, should be considered.

Chapter 5

On the development of efficient *CUDA* kernels for the Helmholtz operator

In this chapter, we develop efficient kernels for elemental operators of matrix-free solvers of the Helmholtz equation, which are the core operations for incompressible Navier-Stokes solvers. We consider regular and curved triangular and quadrilateral elements from unstructured high-order meshes. We investigate two types of efficient *CUDA* kernels for a range of polynomial orders and thus varying arithmetic intensities; a first type which maps each elemental operation to a *CUDA*-thread for a completely vectorised kernel, and a second that maps each element to a *CUDA*-block for nested parallelism. Our results show that the first option is beneficial for elements with low polynomial order, whereas the second option is beneficial for elements of higher order. For both options we show the importance of the right layout of data structures, which necessitates the development of *interleaved* elemental data for vectorised kernels, and analyse the effect of utilising different memory spaces on the GPU. As the considered kernels are foremost memory bandwidths bound, we develop kernels for curved elements that trade memory bandwidths against additional arithmetic operations, and demonstrate improved throughput in selected cases. We further compare our optimised *CUDA* kernels against optimised *OpenACC* kernels, to contrast the performance between a native and a portable programming model for GPUs.

Initial results of the work associated with this chapter have been published in reference [28].

This chapter is organised as follows: We start with an introduction in Section 5.1, and continue to describe the underlying method and mathematical formulations in Section 5.2. The different parallelisation strategies and data layouts are described in Section 5.3. The performance results for triangular and quadrilateral elements are then presented in Section 5.4, alongside a comparison of *CUDA* and *OpenACC*, before we draw conclusions in Section 5.5.

5.1 Introduction

This chapter investigates not a performance portable model, but a native programming model for GPUs: *CUDA* [88]. As already discussed in Chapter 2, the current and future hardware landscape is rather heterogeneous, however at this moment dominated by multi-core CPUs and Nvidia GPUs. This makes the utilisation of a portable programming model an attractive choice, as it is deemed to reduce the efforts to maintain a code base. We have followed this approach in the context of high-order finite element methods (FEM) in Chapter 4. However, we have experienced that this approach also has its limitations: in order to comply with GPU paradigms, data-structures have to be flattened, so that only *plain old data* (POD) types are used. To achieve good performance on the GPU, dynamic memory allocation should be replaced with static memory allocation as much as possible, requiring changes in the way functions or kernels are called, as more templating will be required in the context of high-order FEM to account for different elemental polynomial orders. Moreover, GPUs favour either very large and compute-intense operations, or alternatively a large number of batched operations, that are smaller and homogeneous. This has also been recognised in the context of modern CPUs in reference [84]. Finally, we found that the *gather* operation, which reduces local modes to global modes required a different algorithm for GPUs than for CPUs. Using the non-optimal algorithm on either architecture resulted in an at least an order of magnitude longer runtimes for this kernel. All these points demonstrate that using a portable programming model to target GPU architectures requires a large initial porting effort, that could also be invested developing a native GPU implementation in the first place.

Reference [82] also investigates elemental evaluations for high-order FEM for CPU architectures and finds that only by using assembler intrinsic commands good performance can be achieved on CPUs with larger vector lanes, that are becoming ever more prevalent. Here we therefore propose a strategy to address CPU and GPU hardware by maintaining two different code-bases for the computationally intense kernels and use a common code-base for the pre- and post-processing routines within the method, and potentially the higher levels of time-stepping routines. To accompany a CPU code-base as in reference [82], we develop a bespoke GPU code-base using *CUDA*, but note that the *HIP* programming model [40] would also be worth investigating, as it promises to be portable between AMD and Nvidia GPUs.

Developing and optimising a native GPU implementation also creates a benchmark to establish the maximally attainable performance of our algorithms on a GPU system, and use it to evaluate the performance penalty that occurs when alternatively using a portable model such as *OpenACC*. It allows to make a more informed trade-off choice between the strategy of using either a performance portable programming model or a selection of native programming models.

In terms of algorithms to be investigated, we follow the rationale that also led us in Chapter 4. Considering the incompressible Navier-Stokes equation, an operator splitting

scheme as introduced in reference [64] and in Section 4.2.1 leads to very efficient implementations. In this scheme, the advection terms are solved using explicit time-stepping, while the diffusion and pressure terms are solved using implicit time-stepping. The latter forms the computationally most expensive part and can be expressed in terms of solving the Helmholtz equation.

In a continuous Galerkin approximation, the global Helmholtz operator can be efficiently solved with a low memory footprint using a *matrix-free* approach, which is performed in three steps: first the global modes are scattered to their local element-wise modes, then the Helmholtz operation is performed on each element independently, and finally all elemental modes are gathered back to their global modes. This approach avoids the assembly of the complete global matrix, which although being rather sparse would lead to a huge memory footprint of the application, which would be detrimental for good performance especially on GPUs.

This chapter focusses on the central part of this scheme, the parallel execution of the elemental Helmholtz operator. We note that the scatter and gather operations were already implemented in our work in Chapter 4. Focussing only on the elemental operators, we do not need to consider element connectivity, and also allow our work to be transferred to discontinuous Galerkin schemes.

Here we restrict our work to two-dimensional tensor product based elements, namely triangles and quadrilaterals, but note that our investigations can be easily transferred to three-dimensional elements. Although quadrilateral elements can be computed more efficiently, triangular elements are often required for robust meshing methods for complex geometries. The elemental evaluations are executed in a *matrix-free* approach, too, in which they are broken down into a series of operations, that again reduce the memory footprint. The use of tensor products of 1D basis functions allows the use of the *sum-factorisation* technique, which reduces the number of arithmetic operations, making the matrix-free approach more attractive. The evaluation of the elemental Helmholtz operator is described in more detail in Section 5.2.

Similar work has been conducted by other finite element groups, see for example *deal.II* [72] and *Dune* [9] focusing mostly on efficient vectorised operations for quadrilateral and hexahedral elements on CPUs. These types of elements, however limit the applicability of these methods, as the generation of high-order meshes for complex geometries with all-quadrilateral or all-hexahedral elements is still an open problem. Related work has been performed with a focus on vectorisation for CPU architectures covering all type of elements, including triangular prisms and tetrahedra, in reference [82]. Here we therefore aim to investigate how the techniques presented in these works can be extended to GPU architectures.

High-order finite element methods additionally posses the desirable feature of varying the polynomial order p of the shape functions that are used to perform the elemental evaluations. The choice of polynomial order is influenced by multiple factors and is subject of significant ongoing research efforts. Here we consider the computational aspect, thus

our target measure is maximising a throughput expressed in degrees of freedom (DOF) per time-unit. With increasing polynomial order the number of operations per DOF increases faster than the memory footprint. This results in higher arithmetic intensities of the algorithms for higher polynomial orders p, which on GPUs is often required to achieve good performance and utilisation of the hardware.

It is crucial to design our algorithms with the GPU hardware in mind, especially with regards to the hierarchical parallel structure and also with regards to the different memory spaces, that can be addressed explicitly. The parallelism on the GPU comes on two levels, on the outer level over the streaming multiprocessors, and on the inner level over the *CUDA* cores, where groups of 32 effectively act as a vector lane. As we will show, the decision how to map the parallelism of the algorithm to the hardware parallelism is crucial for good performance. Furthermore GPUs have a complex memory concept, that was introduced in Section 2.1. As we are dealing mostly with memory bandwidth bound algorithms in high-order FEM, the exploitation of good data placement is necessary for maximum performance. We discuss these aspects in more detail in Section 5.3.

Manipulating the arithmetic intensity of the method by varying the algorithms is another factor that we are investigating. Trading a reduced memory bandwidth for more arithmetic operations can lead to overall performance improvements in certain cases. Using roofline models that visualise how close the implementation is to arithmetic or memory bandwidth limits facilitates and serves to explain in which cases algorithmic changes can be beneficial. To our knowledge we present the first case where a reduced bandwidth algorithm achieves better performance than an algorithm with minimum count of arithmetic operations in the context of high-order spectral element methods.

5.2 Method

We implement the following arithmetic operations in our *CUDA* kernels. For more details on the method and the notation, the reader is referred to the book by Karniadakis and Sherwin [63].

We already developed the weak form of the Helmholtz equation using a Galerkin formulation in Chapter 4, which with a condensed right-hand side reads as

$$(\nabla u, \nabla v)_\Omega + \lambda (u, v)_\Omega = -(f, v)_\Omega \,, \tag{5.1}$$

with

$$(u, v)_\Omega = \int_\Omega u(x) v(x) \, \mathrm{d}x \,. \tag{5.2}$$

Substituting expansions for u and v with the same expansion bases ϕ of the form

$$u(x) = \sum_n \hat{u}_n \phi_n(x) \tag{5.3}$$

and

$$v(x) = \sum_m \hat{v}_m \phi_m(x) \tag{5.4}$$

and considering a single element of the mesh, the equation can be written as

$$\int_{\Omega^e} \lambda \hat{u}_n^e \phi_n^e(x) \phi_m^e(x) + \hat{u}_n^e \nabla \phi_n^e(x) \cdot \nabla \phi_m^e(x) \, \mathrm{d}x = \int_{\Omega^e} f(x) \phi_m^e(x) \, \mathrm{d}x, \quad \forall n, m \,. \tag{5.5}$$

As the coefficients $\hat{u}_n^e$ are independent of x, we can factor them out and rewrite the equations as a matrix system

$$\boldsymbol{H}^e \, \hat{u} = \hat{f} \,, \tag{5.6}$$

with the elemental Helmholtz operator $\boldsymbol{H}^e$, the vector of solution coefficients $\hat{u}$, and the right-hand side vector $\hat{f}$.

Considering only the elemental Helmholtz operator, we can now conduct a coordinate transformation from global Cartesian coordinates x to local Cartesian coordinates ξ using the one-to-one mapping $x = \chi^e(\xi)$

$$\boldsymbol{H}^e = \int_{\Omega_{\mathrm{st}}} \left[\lambda \phi_n(\xi) \phi_m(\xi) + (\boldsymbol{J}^e)^{-T} \nabla \phi_n(\xi) \cdot (\boldsymbol{J}^e)^{-1} \nabla \phi_m(\xi) \right] |\boldsymbol{J}^e| \, \mathrm{d}\xi \,, \tag{5.7}$$

with the view to solve the integral on the standard element Ω_{st}. Here we have applied the chain rule

$$\nabla \phi(x) = \frac{\partial \xi}{\partial x} \frac{\partial \phi}{\partial \xi} = \boldsymbol{J}^{-1} \nabla \phi(\xi) \,, \tag{5.8}$$

with the Jacobian matrix of the transformation $\boldsymbol{J} = \nabla \chi(\xi)$.

Representing the integral using Gaussian quadrature, we can write the discrete Helmholtz operator as

$$\boldsymbol{H}_\delta^e = \sum_l \left[\lambda w_l \phi_n(\xi_l) \phi_m(\xi_l) + w_l (\boldsymbol{J}_l^e)^{-T} \nabla \phi_n(\xi_l) \cdot (\boldsymbol{J}_l^e)^{-1} \nabla \phi_m(\xi_l) \right] |\boldsymbol{J}_l^e| \,, \tag{5.9}$$

with a summation over all quadrature points l.

For a two-dimensional element the indices l, m, and n need to be expanded to $l = l(i, j)$, $m = m(r, s)$ and $n = n(p, q)$. Further splitting the discrete Helmholtz operator into the mass operator $\boldsymbol{M}_\delta^e$ and the Laplacian operator $\boldsymbol{L}_\delta^e$ yields:

$$\begin{aligned} \boldsymbol{H}_\delta^e &= \lambda \boldsymbol{M}_\delta^e + \boldsymbol{L}_\delta^e \\ &= \lambda \left[\sum_i \sum_j w_i w_j \phi_{pq}(\xi_{1i}, \xi_{2j}) \phi_{rs}(\xi_{1i}, \xi_{2j}) |\boldsymbol{J}_{ij}^e| \right] \\ &+ \left[\sum_i \sum_j w_i w_j (\boldsymbol{J}_{ij}^e)^{-1} \nabla \phi_{pq}(\xi_{1i}, \xi_{2j}) \cdot (\boldsymbol{J}_{ij}^e)^{-1} \nabla \phi_{rs}(\xi_{1i}, \xi_{2j}) |\boldsymbol{J}_{ij}^e| \right] . \end{aligned} \tag{5.10}$$

5.2.1 Helmholtz operator in matrix form

We can then rewrite the mass operator in matrix form as

$$\boldsymbol{M}_{\delta}^{e} = \boldsymbol{B}^{T}\boldsymbol{W}\boldsymbol{B}\,, \tag{5.11}$$

with the dense basis matrix

$$\boldsymbol{B}[l(i,j)][n(p,q)] = \phi_{pq}(\xi_{1i}, \xi_{2j})\,, \tag{5.12}$$

and the diagonal weight matrix

$$\boldsymbol{W}[l(i,j)][n(p,q)] = w_i w_j |\boldsymbol{J}_{ij}^{e}| \delta_{ln}\,, \tag{5.13}$$

where δ_{ln} is the Dirac delta function.

Noting that for two-dimensional expansions the derivative of the basis functions with respect to the local coordinates is

$$\nabla\phi(\xi_{1i}, \xi_{2j}) = \begin{pmatrix} \dfrac{\partial\phi}{\partial\xi_{1i}} \\ \dfrac{\partial\phi}{\partial\xi_{2j}} \end{pmatrix}, \tag{5.14}$$

and the inverted Jacobian matrix $(\boldsymbol{J})^{-1}$ is

$$(\boldsymbol{J})_{ij}^{-1} = \begin{pmatrix} \dfrac{\partial\xi_1}{\partial x_1} & \dfrac{\partial\xi_2}{\partial x_1} \\ \dfrac{\partial\xi_1}{\partial x_2} & \dfrac{\partial\xi_2}{\partial x_2} \end{pmatrix}_{ij}, \tag{5.15}$$

the Laplacian operator can be expanded as

$$\begin{aligned} \boldsymbol{L}_{\delta}^{e} = \sum_i \sum_j w_i w_j \\ \Bigg[\left(\frac{\partial\xi_1}{\partial x_1}\frac{\partial\phi_{pq}}{\partial\xi_1} + \frac{\partial\xi_2}{\partial x_1}\frac{\partial\phi_{pq}}{\partial\xi_2} \right) \left(\frac{\partial\xi_1}{\partial x_1}\frac{\partial\phi_{rs}}{\partial\xi_1} + \frac{\partial\xi_2}{\partial x_1}\frac{\partial\phi_{rs}}{\partial\xi_2} \right) \\ + \left(\frac{\partial\xi_1}{\partial x_2}\frac{\partial\phi_{pq}}{\partial\xi_1} + \frac{\partial\xi_2}{\partial x_2}\frac{\partial\phi_{pq}}{\partial\xi_2} \right) \left(\frac{\partial\xi_1}{\partial x_2}\frac{\partial\phi_{rs}}{\partial\xi_1} + \frac{\partial\xi_2}{\partial x_2}\frac{\partial\phi_{rs}}{\partial\xi_2} \right) \Bigg]_{ij} |\boldsymbol{J}_{ij}^{e}|\,. \end{aligned} \tag{5.16}$$

We can now rewrite the Laplacian operator in matrix form as

$$\boldsymbol{L}_{\delta}^{e} = \boldsymbol{U}^{T}\boldsymbol{W}\boldsymbol{U} + \boldsymbol{V}^{T}\boldsymbol{W}\boldsymbol{V}, \tag{5.17}$$

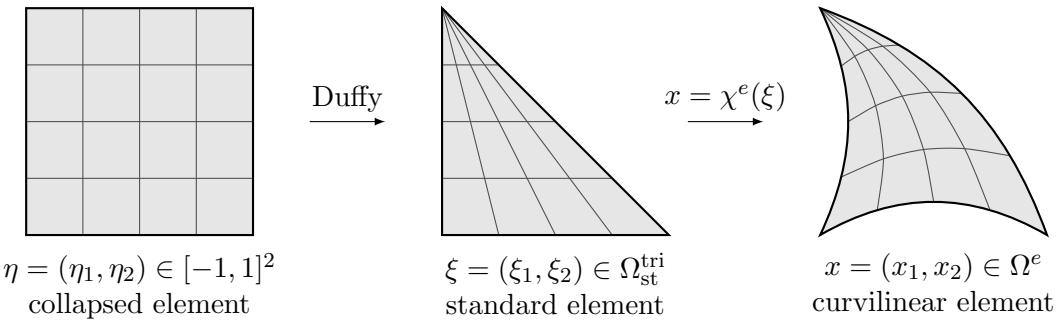

Figure 5.1: Representation of a high-order triangular element with three different coordinate systems. The distribution of quadrature points is illustrated with equispaced lines.

with $\boldsymbol{U}$ and $\boldsymbol{V}$ being

$$\boldsymbol{U} = \left(\boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_1}\right)\boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_1}\right)\boldsymbol{D}_{\xi_2}\right)\boldsymbol{B} \tag{5.18}$$

$$\boldsymbol{V} = \left(\boldsymbol{\Lambda}\left(\frac{\partial \xi_1}{\partial x_2}\right)\boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda}\left(\frac{\partial \xi_2}{\partial x_2}\right)\boldsymbol{D}_{\xi_2}\right)\boldsymbol{B}\,. \tag{5.19}$$

The diagonal coefficient matrix $\boldsymbol{\Lambda}(f)$ is defined as

$$\boldsymbol{\Lambda}(f(\xi_1, \xi_2))[m(i,j)][n(p,q)] = f(\xi_{1i}, \xi_{2j})\delta_{mn}\,, \tag{5.20}$$

while the block-diagonal differentiation matrix $\boldsymbol{D}_{\xi_i}$ is

$$\boldsymbol{D}_{\xi_1}[m(i,j)][m(p,q)] = \left.\frac{\mathrm{d}h_p(\xi_1)}{\mathrm{d}\xi_1}\right|_{\xi_{1i}} \delta_{jq}\,, \tag{5.21}$$

with $h_p(\xi)$ being the one-dimensional Lagrange polynomial.

5.2.2 Dealing with triangular elements

To yield a tensor product formulation for triangles, we employ the standard square-to-triangle Duffy transformation [25] to define two independent coordinate directions. This transformation is illustrated in Figure 5.1. Analytically, the Duffy mapping is defined as

$$\eta_1 = 2\frac{1+\xi_1}{1-\xi_2} - 1, \quad \eta_2 = \xi_2\,. \tag{5.22}$$

For triangles, Equation (5.10) is solved on the collapsed coordinate space, instead of the Cartesian coordinate space. We hence require a partial derivative with respect to the

Cartesian space, which is determined by the chain rule as

$$\begin{pmatrix} \dfrac{\partial}{\partial \xi_1} \\ \dfrac{\partial}{\partial \xi_2} \end{pmatrix} = \begin{pmatrix} \dfrac{2}{1-\eta_2} & 0 \\ 2\dfrac{1+\eta_1}{1-\eta_2} & 1 \end{pmatrix} \begin{pmatrix} \dfrac{\partial}{\partial \eta_1} \\ \dfrac{\partial}{\partial \eta_2} \end{pmatrix}, \tag{5.23}$$

or in matrix notation as

$$\nabla \xi = \boldsymbol{G}\, \nabla \eta\,. \tag{5.24}$$

5.2.3 Basis functions and quadratures

In this work we aim to explore elemental expansions that are based on tensor-products of one-dimensional basis functions. In order to construct a discrete representation of a variable u^δ in *physical* space, these elemental expansions of the transformed variable $\hat{u}$ in *coefficient* space take the form

$$u^\delta_{\text{quad}}(\xi) = \sum_{p=0}^{P} \sum_{q=0}^{Q} \hat{u}_{pq} \psi^a_p(\xi_1) \psi^a_q(\xi_2) \tag{5.25}$$

for a quadrilateral element.

We can define two common choices of basis functions that permit tensor-product decompositions for collapsed elements, like triangles, by additionally employing the collapsed coordinates (η_1, η_2). The first are Dubiner-type basis functions [24], which form an orthogonal *modal* basis on each element. For example, the triangular Dubiner basis can be expressed as

$$\tilde{\psi}_{pq}(\eta) = \underbrace{\sqrt{2} P^{(0,0)}_p(\eta_1)}_{\psi^a_p(\eta_1)} \underbrace{P^{(2p+1,0)}_q(\eta_2)(1-\eta_2)^p}_{\psi^b_{pq}(\eta_2)}, \tag{5.26}$$

where $P^{(\alpha,\beta)}$ denotes the standard Jacobi polynomial and the indices p and q lie in the triangular indexing set. Under a suitable indexing strategy, where the limits on inner summations now rely on outer summations, this basis can still be decomposed into two one-dimensional functions for η_1 and η_2, for example as

$$u^\delta_{\text{tri}}(\eta) = \sum_{p=0}^{P} \psi^a_p(\eta_1) \left[\sum_{q=0}^{Q-p} \hat{u}_{pq} \psi^b_{pq}(\eta_2) \right]. \tag{5.27}$$

The second choice of tensor product basis is the modified C^0-basis presented in reference [63] and normally used in *Nektar++*. These consist of interior and boundary modes: a set of orthogonal interior high-order polynomials which are zero along all boundaries, in addition to a set of linear vertex modes, high-order edge boundary modes, and – in case of three-dimensional elements – high-order face boundary modes. This results in a basis that is not strictly orthogonal for collapsed elements, but enables straightforward C^0 connectivity and separation into boundary and interior modes for static condensation

approaches. As we are interested in using the Helmholtz kernel as a basis for a continuous Galerkin discretisation to solve the incompressible Navier-Stokes equation, we will employ the C^0-basis here.

The modified C^0 basis for the elements we consider in this work are defined using two tensor product bases:

$$\psi_p^a(\xi) = \begin{cases} \frac{1-\xi}{2} & p = 0 \\ \frac{1+\xi}{2} & p = 1 \\ \left(\frac{1-\xi}{2}\right)\left(\frac{1+\xi}{2}\right) P_{p-1}^{1,1}(\xi) & 2 \leq p < P \end{cases} \tag{5.28}$$

$$\psi_{pq}^b(\xi) = \begin{cases} \psi_q(\xi) & p = 0 & 0 \leq q < P \\ \left(\frac{1-\xi}{2}\right)^p & 1 \leq p < P & q = 0 \\ \left(\frac{1-\xi}{2}\right)^p \left(\frac{1+\xi}{2}\right) P_{q-1}^{2p-1,1}(\xi) & 1 \leq p < P, & 1 \leq q < P - p \end{cases}, \tag{5.29}$$

where P and Q denote the polynomial order in each coordinate direction, and $P_p^{\alpha,\beta}(\xi)$ is the standard Jacobi polynomial.

In order to exactly integrate the mass matrix, we select $Q = P + 2$ Gauss-Lobatto-Legendre points in both coordinate directions for quadrilateral elements, with P being the polynomial order of the element.

For the collapsed coordinate direction of triangular elements, the singularity at the collapsed vertex needs to be treated with care. Here we employ the customary option of using $Q = P + 1$ Gauss-Radau points, which exclude the endpoint at the collapsed vertex. Despite having one fewer integration point, the same accuracy as for Gauss-Lobatto-Legendre points is achieved.

5.3 Parallel implementation

The matrix-free Helmholtz operator is implemented in a standalone benchmarking mini-application. Again, it is coupled to the *Nektar++* framework [14, 83] to link to its libraries that construct utility data like expansion basis functions, their derivatives, quadrature points, and weights, as well as mesh connectivity and geometric mappings.

5.3.1 Mapping of parallelism

Each elemental operation within the global Helmholtz solution can be performed independently and hence in parallel. Additionally, we can realise a second nested level of parallelism within each element over the different quadrature points or modes. Therefore, we investigate two different mappings of the parallel algorithms to the parallel GPU hardware. We recall the hardware layout of current GPUs, that is decomposed into a few dozen streaming multiprocessors (SMX), which in turn consists of groups of 32 *CUDA*-cores, that can be considered the equivalent of a AVX vector lane on CPUs. As opposed to the compiler intrinsics used for vectorised CPU instructions in reference [82] that would

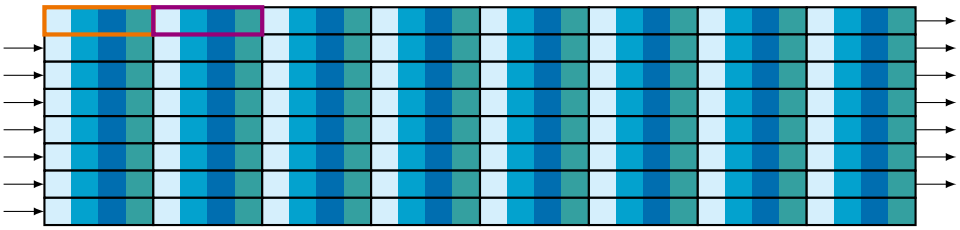

Figure 5.2: Default memory layout with stride = 1. The data corresponding to the first two elements are framed in orange and violet, with each element having 4 entries.

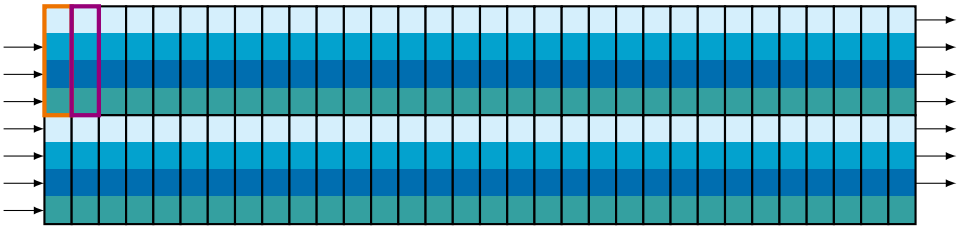

Figure 5.3: Interleaved memory layout with stride = 32 (size of a GPU warp). The data corresponding to the first two elements are framed in orange and violet, with each element having 4 entries.

be basically impossible to achieve with compiler auto-vectorisation, dense vectorisation is naturally achieved using the *CUDA* programming model for GPUs. Along the same lines, we do not need to specify fused-multiply-add (FMA) operations explicitly, but let the compiler use them when beneficial.

Element *per-thread* parallelism

As the first option, each elemental operation is mapped to a single *CUDA*-thread. For executions on CPUs, this approach has been explored for example in references [32, 72, 71, 8].

This mapping is illustrated in Figure 5.4, where each elemental operation, depicted with a circle, is assigned to one *CUDA*-core on which the *CUDA*-thread operates. For this option, it is paramount to place special emphasis on efficient data-structures by interleaving the data of a group of 32 elements for the vectorised operations within a *CUDA*-warp. This is illustrated in Figure 5.2 for the default memory layout, where each elemental data set is placed consecutively into the linear memory array. Here all the first data units of each elemental data set are scattered over the memory, and need to be loaded individually. However, when the data is interleaved with a stride equal to the vectorisation length, as illustrated in Figure 5.3, a contiguous memory access can be achieved so that the bandwidth of loading operations is increased significantly. This option does not require any padding within the elemental operation. Only if the total number of elements is not divisible by the vector width, the last warp will include some idle threads. We run all cases with a block-size of 128 threads, that has shown the most performant results in preliminary investigations.

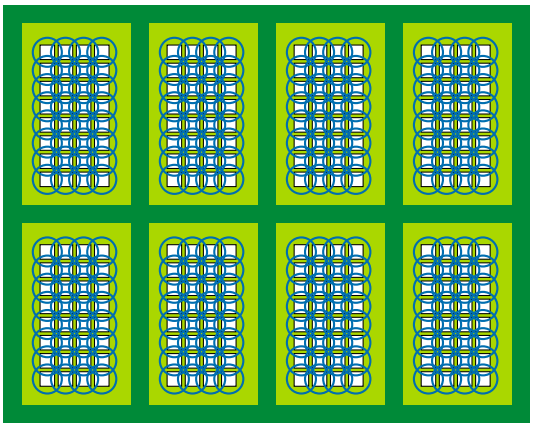

Figure 5.4: Element *per thread* parallelism mapped to the GPU hardware, each operation is executed by one *CUDA* core.

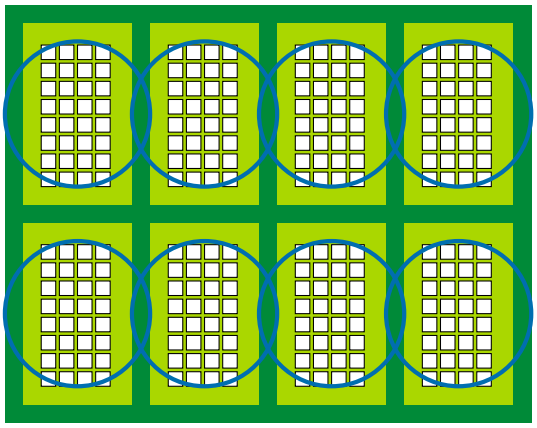

Figure 5.5: Element *per block* parallelism mapped to the GPU hardware, each operation is executed by one SMX.

In terms of the *CUDA* syntax, we refer to Listing 5.1. It illustrates the employed concepts in a simplified version of the Helmholtz kernel for a quadrilateral element. It shows the template parameters relating to the polynomial order (number of modes and quadrature points), a few important input parameters (the pointers to the utility data arrays are not shown for brevity), and the employed vectorisation width of 32. The indexing for the variable `elmt` ensures each thread is assigned a single element until all elements have been processed. As the data is interleaved with the vector width of 32 as illustrated in Figure 5.3, a sophisticated indexing to receive the element local pointer to the input and output array is employed. Finally, loops i and j over the quadrature points for the elemental operation are executed in serial.

Element *per-block* parallelism

As the second option, each elemental operation is mapped to a single *CUDA*-block. This approach has been applied in the *EXA-DUNE* project [9] in the context of CPUs and in references [127, 62] in the context of GPUs.

This option is illustrated in Figure 5.5, where each elemental operation, depicted with a circle, is assigned to a streaming multiprocessor (SMX) on which the *CUDA*-block operates. Here it is critical to harness the parallelism within the elemental operations by assigning one or more quadrature points or modes (depending if the operation is in physical or transformed space) to one *CUDA*-thread. In this case the default memory layout achieves contiguous memory access and is therefore employed. One drawback of this method is that the number of quadrature points or modes is rarely divisible by the block size, and so a few threads will run idle. To minimise the number of idle threads, we choose a different block size for each polynomial order being the next natural multiple of 32, the warp size. Additionally, special care needs to be taken to assign each thread the correct indices within the kernels. As we employ the sum-factorisation approach, these assign-

Listing 5.1: *Per-thread*-parallel *CUDA* kernel indexing

```
template<int nm0, int nm1,        // number of modes
         int nq0, int nq1>        // number of quadrature points
__global__ void CUDAHelmholtzQuad_Thread(
    const double *inptr,          // global input array
          double *outptr,         // global output array
    const double lambda,
    const int nElmt,              // total number of elements
    const int nmTot,              // number of modes on each element
    [utility data])
{
    constexpr int VW = 32;        // vector-width equal to warp size

    // perform operations on each element in parallel
    // the default CUDA variables for thread and block indexing are
        used: threadIdx.x, blockIdx.x, blockDim.x, gridDim.x
    for (int elmt = blockIdx.x * blockDim.x + threadIdx.x;
         elmt < nElmt;
         elmt += blockDim.x * gridDim.x)
    {
        // get element local pointer
        const int data_block = blockIdx.x * (blockDim.x / VW) +
           threadIdx.x / VW;
        const int data_elmt = threadIdx.x % VW;
        const double *in = inptr + data_block * VW * nmTot +
           data_elmt;
        double *out = outptr + data_block * VW * nmTot + data_elmt;

        for (int j = 0; j < nq1; ++j)
        {
            for (int i = 0; i < nq0; ++i)
            {
                // perform operations for each quadrature point in
                   serial
            }
        }
    }
}
```

Listing 5.2: *Per-block*-parallel *CUDA* kernel indexing

```
template<int nm0, int nm1,        // number of modes
         int nq0, int nq1>        // number of quadrature points
__global__ void CUDAHelmholtzQuad_Block(
    const double *inptr,          // global input array
          double *outptr,         // global output array
    const double lambda,
    const int nElmt,              // total number of elements
    const int nmTot,              // number of modes on each element
    [utility data])
{
    constexpr int VW = 1;         // vector-width

    // perform operations on each element in parallel
    // the default CUDA variables for thread and block indexing are
        used: threadIdx.x, blockIdx.x, blockDim.x
    const int elmt = blockIdx.x;

    // get element local pointer
    const double *in  = inptr  + elmt * nmTot;
          double *out = outptr + elmt * nmTot;

    for (int cnt_ij = threadIdx.x; cnt_ij < nq0 * nq1; cnt_ij +=
       blockDim.x)
    {
        // [cnt_ij = j * nq0 + i]
        const int j = cnt_ij / nq0;
        const int i = cnt_ij - j * nq0;
        // perform operations for each quadrature point in parallel
    }
    __syncthreads();
}
```

ments often change within one operation. For the storage of temporary results, we use shared memory arrays to create a workspace that can be suitably addressed by all threads within a block.

In terms of the *CUDA* syntax, we refer to Listing 5.2, in order to highlight the differences with the first option. The indexing of the variable `elmt` ensures that each element is assigned one *CUDA*-block. Here the pointers to the input and output arrays are much simpler, as these arrays have a vector-width of 1 and are thus not interleaved. To introduce the parallelism over the quadrature points, the loop indices i, and j need to be assigned to the individual *CUDA*-threads `threadIdx.x`. Using a for-loop ensures that all quadrature points are computed, even if the number of threads scheduled in this block would be smaller. To yield the correct and contiguous index for each thread, we utilise the technique of integer division, even though it incurs the cost of around a dozen cycles per integer division operation. After each loop-block, all threads need to be synchronised.

5.3.2 Computing the derivative factors

Another parameter that needs to be considered for the kernels is the degree to which repeatedly needed data arrays are precomputed and subsequently loaded from global memory or could alternatively be re-computed *on the fly*. Re-computing them can potentially be more efficient for already memory-bound implementations and can shift some of the memory traffic to otherwise unused arithmetic compute resources.

To address this question, we investigate the computation of the determinant of the Jacobian matrix and the derivative factors $\mathbf{\Lambda}\left(\frac{\partial \xi_i}{\partial x_j}\right)$ via the elemental deformation mapping χ^e. For the default implementation these data have been pre-computed and stored, as has been customary within *Nektar++* to minimise the operation count. This resulted in the most efficient option on older hardware.

Now, when re-computing the Jacobian matrix and the derivative factors for two-dimensional elements, only two arrays with the length of the number of geometric modes, instead of five arrays with the length of the number of quadrature points need to be loaded. For an iso-parametric mapping (where the polynomial degree of the mesh geometry and of the elemental evaluation is equal) this already constitutes a saving of 60% memory bandwidth. For a sub-parametric mapping (where the mesh geometry is of a lower order than the elemental evaluation), the memory savings are even more substantial. For three-dimensional elements only three arrays containing the mapping need to be loaded instead of 10 arrays containing the Jacobian and the derivative factors, resulting in a bandwidth saving of at least 70%.

To derive the calculation instructions, we recall the geometric mapping $\chi^e : \Omega_{st} \to \Omega^e$ from coordinates on the standard element $\xi \in \Omega_{st}$ to the global coordinates $x \in \Omega^e$. We therefore have the relation

$$x_i = \chi_i^e(\xi)\,. \tag{5.30}$$

The mapping χ^e is stored in the form of its coefficients in transformed space $\hat{\chi}^e$. As a first step, we therefore need to apply the backward transformation of $\hat{\chi}^e$ in each coordinate direction as

$$x_i = \chi_i^e(\xi) = \sum_p \sum_q (\hat{\chi}_i^e)_{pq} \phi_{pq}(\xi)\,, \tag{5.31}$$

or in matrix form:

$$x_i = \boldsymbol{B}\hat{\chi}_i^e\,. \tag{5.32}$$

As a second step to recover the Jacobian $\boldsymbol{J} = \nabla\chi^e$, the spatial derivatives of the global coordinates need to be applied as $\frac{\partial x_i}{\partial \xi_j}$, or in matrix form

$$\frac{\partial x_i}{\partial \xi_j} = \boldsymbol{D}_{\xi_j} x_i\,. \tag{5.33}$$

Alternatively, we can apply the backward transformation using the derivative bases,

and not the bases itself as

$$\frac{\partial x_i}{\partial \xi_j} = \frac{\partial \chi_i^e(\xi)}{\partial \xi_j} = \sum_p \sum_q (\hat{\chi}_i)_{pq} \frac{\partial \phi_{pq}(\xi)}{\partial \xi_j} \,. \tag{5.34}$$

Again, this can be expressed in matrix form as

$$\frac{\partial x_i}{\partial \xi_j} = (\boldsymbol{D}_{\xi_j} \boldsymbol{B}) \hat{\chi}_i \,. \tag{5.35}$$

Whereas the second option requires slightly less operations, is requires the loading of the basis function and its derivative. Both options executed in almost the same time, however the first option was slightly faster for triangular elements, whereas the second option was slightly faster for quadrilateral elements. To recover the correct global coordinates, we use the modified C^0 basis for these operations. Also note that for triangular elements, these operations are performed on the collapsed coordinates η. The transformation from collapsed space to standard space need to be performed using the matrix $\boldsymbol{G}$ as follows:

$$\nabla_\xi x = \boldsymbol{G} \nabla_\eta x \,. \tag{5.36}$$

After having calculated the entries of the Jacobian matrix $J_{ij}^e = \frac{\partial x_i}{\partial \xi_j}$, we can compute its determinant $|\boldsymbol{J}^e|$ and the derivative factors $\frac{\partial \xi_i}{\partial x_j}$. For a two-dimensional element, we have the relations:

$$\frac{\partial \xi_1}{\partial x_1} = \frac{1}{|\boldsymbol{J}^e|} \frac{\partial x_2}{\partial \xi_2}, \qquad \frac{\partial \xi_1}{\partial x_2} = -\frac{1}{|\boldsymbol{J}^e|} \frac{\partial x_1}{\partial \xi_2}, \tag{5.37}$$

$$\frac{\partial \xi_2}{\partial x_1} = -\frac{1}{|\boldsymbol{J}^e|} \frac{\partial x_2}{\partial \xi_1}, \qquad \frac{\partial \xi_2}{\partial x_2} = \frac{1}{|\boldsymbol{J}^e|} \frac{\partial x_1}{\partial \xi_1}. \tag{5.38}$$

5.3.3 Templating parameters

Apart from the two major parameters introduced above – the parallel mapping and the calculation of derivative factors – there are a number of additional parameters to explore in terms of improving the efficiency of the Helmholtz kernels.

Placement of utility data in memory

To address another design consideration, we analyse the effect of placing utility data like the basis matrix $\boldsymbol{B}$, quadrature weights w, and derivative matrices $\boldsymbol{D}_{\xi_i}$ into either *global* memory, *constant* memory, or *shared* memory of the GPU. The different GPU memory spaces have been discussed in Section 2.1.3. The default option of employing *global* memory results in the lowest potential memory bandwidth, whereas *shared* memory is physically located on a higher cache level with increased bandwidth, but smaller size. From here each thread can fetch data without a penalty for non-contiguous access. Finally, the *constant* memory allows very fast read access, but only if all threads within one warp are accessing the same memory address. This makes this memory space a promising candidate

for utility data of the completely vectorised kernels. On all three memory spaces, data that is constant and required by all threads in a warp at the same time is automatically vectorised.

Curved versus regular elements

Curved high-order elements allow to represent curved CAD geometries more accurately, leading to more accurate flow solutions. They however require more involved computations, as the Jacobian matrix of these elements needs to be evaluated at each quadrature point. This is caused by the nonlinear standard-to-global mapping χ^e for this case. If this mapping is affine, however, which is the case for straight-sided triangles or parallelogram quadrilaterals, the Jacobian matrix is constant. This results in a small saving of number of operations, as the constant values for the Jacobian matrix can be factored out of the summation over the quadrature points. More significantly, in a regime of memory-bound implementations, storing only one Jacobian matrix per element results in a significant saving of memory. Comparing with the memory requirements for each degree of freedom, that is an input and an output vector of the solution variable with the length of the degrees of freedoms, the full Jacobian matrix requires d^2 entries per quadrature point. Because of these large performance implications, we consider these two cases separately through the use of template programming, so the compiler auto-optimisation can realise the full performance saving potential. In the following these two cases are referred to as *curved* for the curved elements versus *regular* for elements with an affine mapping.

Elemental polynomial order

Another very important parameter is the polynomial order of the elemental evaluations, since it changes the arithmetic intensity of the kernels and therefore its performance profile, which is discussed in more detail in Section 5.3.5.

The polynomial order is a crucial templating parameter, as only a known value at compile time can enable the compiler's ability to do automatic code optimisations, like loop unrolling, index re-ordering and optimising data transfers. Specifically for this purpose, we include the related template parameters which are the number of modes and quadrature points in each direction. To enhance usability and avoid recompilation due to a change in polynomial order, we use a jump table to specify precompiled kernels from a range of common polynomial orders in the range $1 \leq P \leq 10$. We found that using templating to allow for compiler auto-optimisation is a significant performance factor, often reducing the execution times by 20-50%, as can also be seen for the results of the *OpenACC* implementation in Section 5.4.3.

5.3.4 Structuring the Helmholtz operator

To simplify the implementation and avoid code-duplication, the Helmholtz operator can be decomposed into a combination of three sub-operators, as already noted in reference [84].

These are:

- `BwdTrans`: Calculates a backwards transformation or polynomial interpolation from transformed space onto physical space. This operation is implemented by the matrix-vector product of the basis function matrix $\boldsymbol{B}$ with the elemental coefficients $\hat{\boldsymbol{u}}$ as

$$\boldsymbol{u} = \boldsymbol{B}\hat{\boldsymbol{u}}\,. \tag{5.39}$$

- `InnerProduct`: Calculates the L^2 inner product $(\cdot,\cdot)_\Omega$ in physical space implemented with Gaussian quadrature. Here the arguments of the inner product are always the expansion modes ϕ_{pq} or its derivatives $\frac{\partial \phi_{pq}}{\partial \xi}$ and a function in physical space u_{ij} further using the quadrature weights w_i, w_j and the Jacobian determinant $|\boldsymbol{J}_{ij}|$. In matrix notation this operation is

$$\hat{\boldsymbol{u}} = \boldsymbol{B}^T[\boldsymbol{W}\boldsymbol{u}]\,. \tag{5.40}$$

- `TensorDerivative`: Calculates the partial derivatives on the standard element of a function in physical space. For each direction a matrix-vector product of the derivative matrix $\boldsymbol{D}_\xi$ with the function $\boldsymbol{u}$ is performed as

$$\frac{\partial \boldsymbol{u}}{\partial \xi_1} = \boldsymbol{D}_{\xi_1}\boldsymbol{u}, \qquad \frac{\partial \boldsymbol{u}}{\partial \xi_2} = \boldsymbol{D}_{\xi_2}\boldsymbol{u}\,. \tag{5.41}$$

The combination of these sub-operations that collectively form the Helmholtz-operator is illustrated in Algorithm 5.1. We note that we do not specify these operators as individual kernels, but as inlined functions to achieve better performance. To minimise the operator count, the operator involves the precomputed matrix products $\nabla\boldsymbol{B}_i = \boldsymbol{D}_{\xi_i}\boldsymbol{B}$.

5.3.5 Scaling of elemental evaluations

In order to better understand the scaling of the arithmetic intensity with the polynomial order of the elemental expansion, a numerical example of a structure very similar to common operators in the spectral-*hp* element method shall be discussed.

Consider an interpolation on a two-dimensional standard quadrilateral element from a set of points (ξ_{1k}, ξ_{2l}) to another set of points (ξ_{1i}, ξ_{2j}) using Lagrange polynomials h, namely

$$U_{ij} = U(\xi_{1i}, \xi_{2j}) = \sum_{k=1}^{P}\sum_{l=1}^{P} u(\xi_{1k}, \xi_{2l})h_k(\xi_{1i})h_l(\xi_{2j}), \qquad 1 \leq i, j \leq P\,. \tag{5.42}$$

Since all indices i, j, k, and l are of $\mathcal{O}(P)$, then for each point i and j operations of $\mathcal{O}(P^2)$ need to be performed. The number of points i and j are of $\mathcal{O}(P^2)$ themselves, so the overall number of operations will be of $\mathcal{O}(P^4)$.

Algorithm 5.1 Overview of the matrix-free evaluation of the Helmholtz operator

procedure HELMHOLTZ($\hat{\boldsymbol{u}}, w, \boldsymbol{B}, \boldsymbol{\nabla B}, \boldsymbol{D}_\xi, (\boldsymbol{J}^e)^{-1}, |\boldsymbol{J}^e|$)
  $\boldsymbol{u} \leftarrow$BWDTRANS($\hat{\boldsymbol{u}}, \boldsymbol{B}$)
  $\boldsymbol{out} \leftarrow \lambda \cdot$INNERPRODUCT($\boldsymbol{u}, \boldsymbol{B}, w, |\boldsymbol{J}^e|$)
  $\boldsymbol{D}_\xi \boldsymbol{u} \leftarrow$TENSORDERIVATIVE($\boldsymbol{u}, \boldsymbol{D}_\xi$)
  if element is not curved **then**
    calculate element constant metric $\boldsymbol{M} \leftarrow (\boldsymbol{J}^e)^{-1}(\boldsymbol{J}^e)^{-T}$
  end if
  for each quadrature point ξ_i **do**
    if element is curved **then**
      calculate metric $\boldsymbol{M} \leftarrow (\boldsymbol{J}^e)^{-1}(\boldsymbol{J}^e)^{-T}$
    end if
    if triangular element **then**
      apply collapsed coordinate factors $\boldsymbol{M} \leftarrow \boldsymbol{GM}$
    end if
    map from standard region to global region $\boldsymbol{D}_x \boldsymbol{u}[i] \leftarrow \boldsymbol{M}\boldsymbol{D}_\xi \boldsymbol{u}[i]$
  end for
  for each dimension d **do**
    $\boldsymbol{out} \leftarrow \boldsymbol{out} +$ INNERPRODUCT($(\boldsymbol{D}_x \boldsymbol{u})_d, \nabla \boldsymbol{B}_d, w, |\boldsymbol{J}^e|$)
  end for
end procedure

In case this expansion is a tensor product of one-dimensional expansions, we can use the sum-factorisation approach and factor out the term $h_k(\xi_{1i})$ as

$$U_{ij} = \sum_{k=1}^{P} h_k(\xi_{1i}) \left(\sum_{l=1}^{P} u(\xi_{1k}, \xi_{2l}) h_l(\xi_{2j}) \right), \qquad 1 \leq i, j \leq P\,. \tag{5.43}$$

It is now possible to calculate the terms in brackets first as

$$\bar{U}_{jk} = \sum_{l=1}^{P} u(\xi_{1k}, \xi_{2l}) h_l(\xi_{2j}), \qquad 1 \leq j, k \leq P\,, \tag{5.44}$$

which is an $\mathcal{O}(P^3)$ operation. The second step, which yields the overall summation is then

$$U_{ij} = \sum_{k=1}^{P} h_k(\xi_{1i}) \bar{U}_{jk}, \qquad 1 \leq i, j \leq P, \tag{5.45}$$

which is again an $\mathcal{O}(P^3)$ operation. The additional cost of this sum-factorisation is the requirement for an $\mathcal{O}(P^2)$-sized scratch memory space.

For a three-dimensional hexahedral element we would equivalently reduce an $\mathcal{O}(P^6)$ operation to an $\mathcal{O}(P^4)$ operation, and in general a d-dimensional shape complexity reduces from $\mathcal{O}(P^{2d})$ to $\mathcal{O}(P^{d+1})$.

The memory bandwidths for these operations scale with the number of degrees of freedom and hence with $\mathcal{O}(P^2)$ and $\mathcal{O}(P^3)$ for two- and three-dimensional elements, re-

spectively. It becomes clear that polynomial expansions of higher order lead to higher arithmetic intensities of the operators, with the strength of the scaling depending on the utilisation of the sum-factorisation approach.

5.4 Performance results

We investigate the Helmholtz kernels for polynomial orders in the range $1 \leq P \leq 10$ on a Nvidia Titan V GPU on a single workstation. The Titan V GPU possesses a maximum of 7450GFLOPS, is composed of 80 streaming multiprocessors (SMX), has an overall peak DRAM bandwidth of 651.3GB/s, and a peak bandwidth of the L1 cache of 14899.2GB/s. We utilise the *nvcc* compiler version 9.1 and set the *-O3* optimisation flag. No register limits are set. We emphasise again that we have set up specific kernels for each polynomial order and each combination of execution parameter with a templating approach. These parameters are the *per-thread* versus *per-block* parallelism, curved versus regular elements, the memory space for utility data, or the option of re-calculating the Jacobian and derivative factors. The reported performance metrics are averaged over 100 consecutive kernel executions. The correctness of the CUDA implementation is compared in all cases against a serial CPU version, that is based on the default *Nektar++* library. This is done by computing the infinity norm of the error between the two complete solutions and ensuring it is not larger than typical machine rounding errors. We investigate quadrilateral elements in Section 5.4.1, and triangular elements in Section 5.4.2. We then compare the performance for triangles between our *CUDA* implementations against *OpenACC* implementations in Section 5.4.3.

5.4.1 Quadrilateral elements

In Figure 5.6, we report the average FLOPS achieved by five different kernels for regular quadrilateral elements using different memory spaces. Looking at the completely vectorised operations, it shows that placing constant utility data into the *constant* memory space, compared with the default of loading this data from *global* memory space, can lead to performance improvements of around 20% for lower polynomial orders where this type of parallelisation is superior. Improvements of more than 100% are achieved for higher polynomial orders, where the vectorised versions are inferior to the block-parallel versions and would not be employed, however they serve as an example of the large effect of correct data placement. Loading the utility data to *shared* memory for the vectorised operation leads to worse performance than the default option for $P < 9$. In these cases the compiler excells at loading the utility data to higher cache levels, compared with manual *shared* memory load instructions to L1 cache. Looking at the *CUDA*-block parallel operations, improvements of up to 20% for elements of $P > 6$ can be realised by loading the utility data to *shared* and *constant* memory, instead of using *global* memory. Only the quadrature weights w_1 and w_2 are loaded to *constant* memory and all other utility data to *shared*

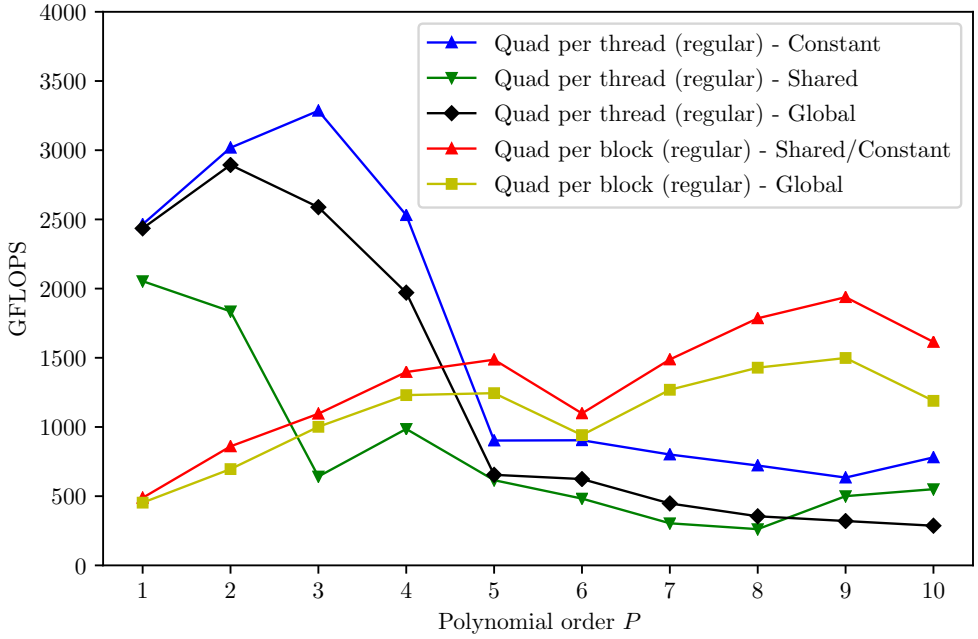

Figure 5.6: Performance of thread-parallel and block-parallel kernels for regular quadrilateral elements. The usage of *global*, *constant*, and *shared* memory space for utility data is compared.

memory, as the algorithm allows only the entries of the quadrature weights to be called at the same time by all threads.

Figure 5.7 shows the equivalent results for curved quadrilateral elements. Again, it can be seen that using *constant* memory for the completely vectorised kernels and a combination of *shared* and *constant* memory for the element-per-block kernels is beneficial.

Figure 5.8 compares the optimum kernels in terms of memory placement, measured in processed degrees of freedom (DoF) per second. This measure looks not only at how well the hardware is utilised, but also factors in the suitability of the algorithm to effectively process the mathematical operations. We can derive that for the specific GPU employed here, the mapping of elemental operations per thread is beneficial up to a polynomial order of $P = 4$, after which register spilling to the slower *local* memory occurs, as we will see later. For higher polynomial orders, the mapping per block is superior, while its throughput is limited for low polynomial orders because the smaller number of modes do not fully occupy all 32 threads of a warp. Higher arithmetic intensities are also shown to generally favour the performance of higher polynomial orders. Slightly breaking that trend are the performance dips for polynomial orders of $P = 6$ and $P = 10$ for the *per-block* parallel version. We investigated multiple possible causes and found that the varying numbers of busy and idle threads and thus the warp-efficiency between polynomial orders alone can not explain this effect. However, we found a significant drop in shared memory efficiency for $P = 6$ and $P = 10$. Yet, as the kernels for all polynomial orders stem from

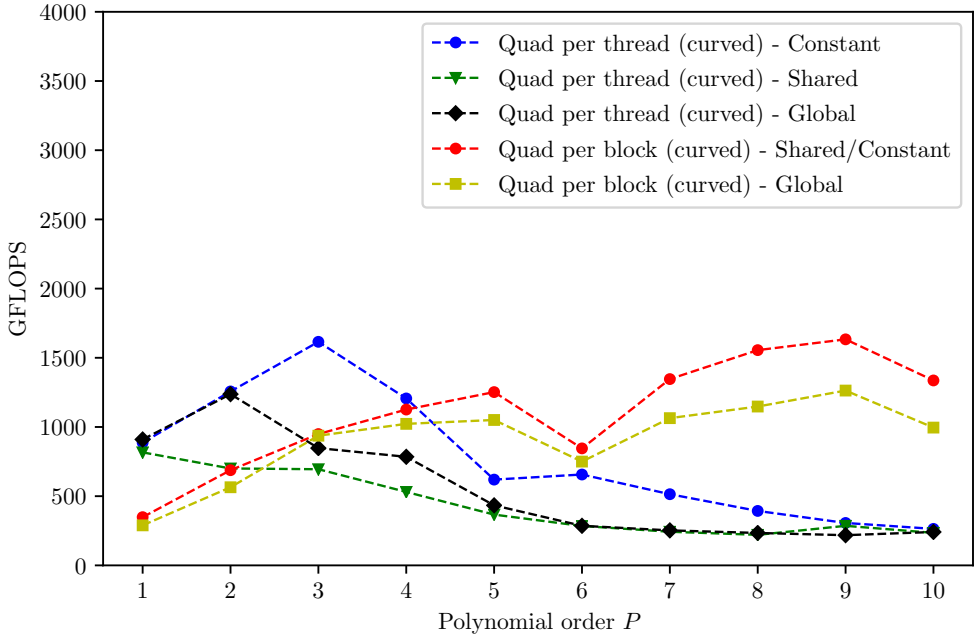

Figure 5.7: Performance of thread-parallel and block-parallel kernels for curved quadrilateral elements. The usage of *global*, *constant*, and *shared* memory space for utility data is compared.

the same source code, it is not entirely clear why the compiled code would show such pronounced performance differences.

The figure further shows the difference in throughputs achieved depending on the deformation of the elements. Especially for the vectorised kernels, it is very beneficial to have a separate kernel for regular elements. Additionally, we present three kernels which calculate the Jacobian and the derivative factors *on the fly*, which are denoted *reduced bandwidth* kernel for curved elements. The first one with per-thread parallelism, which is based on an iso-parametric mapping of first to fifth order achieves increased throughput for polynomial order of less than three, compared with the default version. The second version, which is based on a second-order geometric mapping χ for all evaluation polynomial orders, performs almost identically to the first version, meaning it cannot capitalise on having a lower memory footprint and requiring less arithmetic operations for polynomial orders P from 3 to 5. The third version for per-block parallelism performs worse than the default version across all polynomial orders. This will be explained in the following using *roofline* plots.

To assess how efficiently we employ the GPU hardware, we present the achieved FLOPS of the four optimal kernels plus two reduced bandwidth kernels in two different so called *roofline* plots. The roofline performance model has first been introduced in reference [121]. The horizontal lines of a roofline plot depict the limits in terms of FLOPS, whereas the diagonal line depicts the memory bandwidth limit of the specific memory level. The FLOPS achieved are visualised against the arithmetic intensity achieved, that is FLOPS

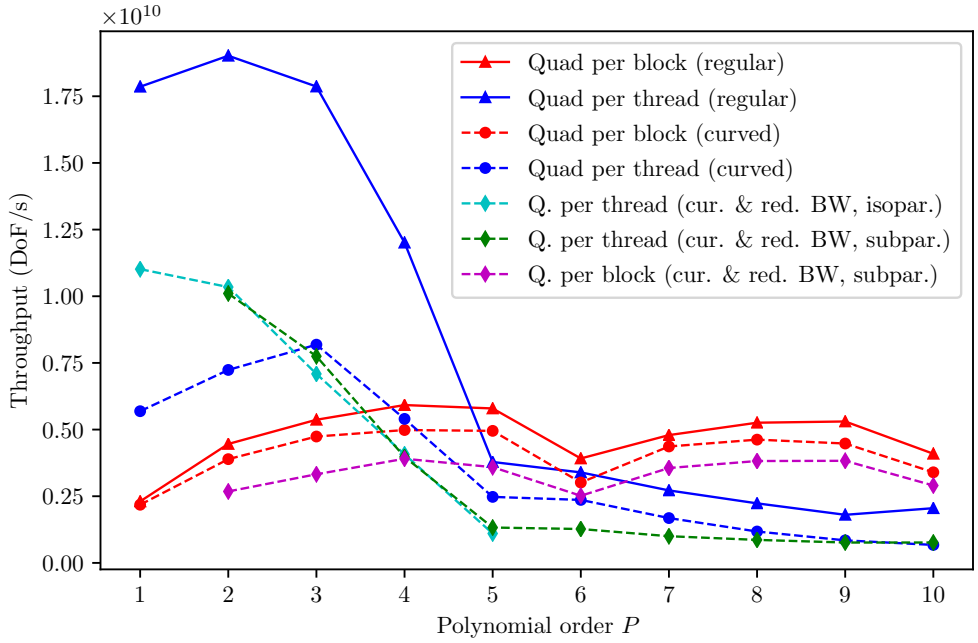

Figure 5.8: Throughput of thread-parallel and block-parallel kernels for regular and curved quadrilateral elements, as well as *reduced bandwidth* kernels for curved elements based on isoparametric or subparametric mappings. All kernels are employing the optimal memory space for utility data.

divided by memory bandwidth in bytes. From Figure 5.9, we can deduce that the kernels that map per-thread are memory bound at the DRAM level, whereas the kernels that map per-block are not close to either the DRAM, nor the FLOP limit. The reduced bandwidth kernels with per-thread parallelism have a higher arithmetic intensity than their respective default kernels for curved elements. This is due to both more arithmetic operations and less required DRAM bandwidth. Accordingly, the performance of the kernels moves to the right of the plot, while staying close to the memory limit, to achieve a higher FLOP rate. While all five kernels in the range $1 \leq P \leq 5$ perform better than their respective default kernel in terms of hardware utilisation, only for $P < 3$, they outperform their default kernel sufficiently to yield a higher throughput despite a higher operation count. The reduced bandwidth kernel for per-block parallelism achieve a higher arithmetic intensity than their respective default kernel, as can be expected by the modified algorithm, however the achieved FLOP rate is still about the same.

To evaluate the potential limit for the per-block parallel kernels, we consider a roofline plot based on L1 cache. Looking at Figure 5.10, we can indeed deduce that these kernels are memory bound at the L1 cache level, whereas the completely vectorised kernels are not. It can also be observed that the achieved arithmetic intensity of the reduced bandwidth kernels is comparable to their respective default kernel, and hence they achieve similar FLOP rates and ultimately a lower throughput, as seen in Figure 5.8.

Despite the memory bottlenecks, the best kernels nevertheless achieve up to 42% of the

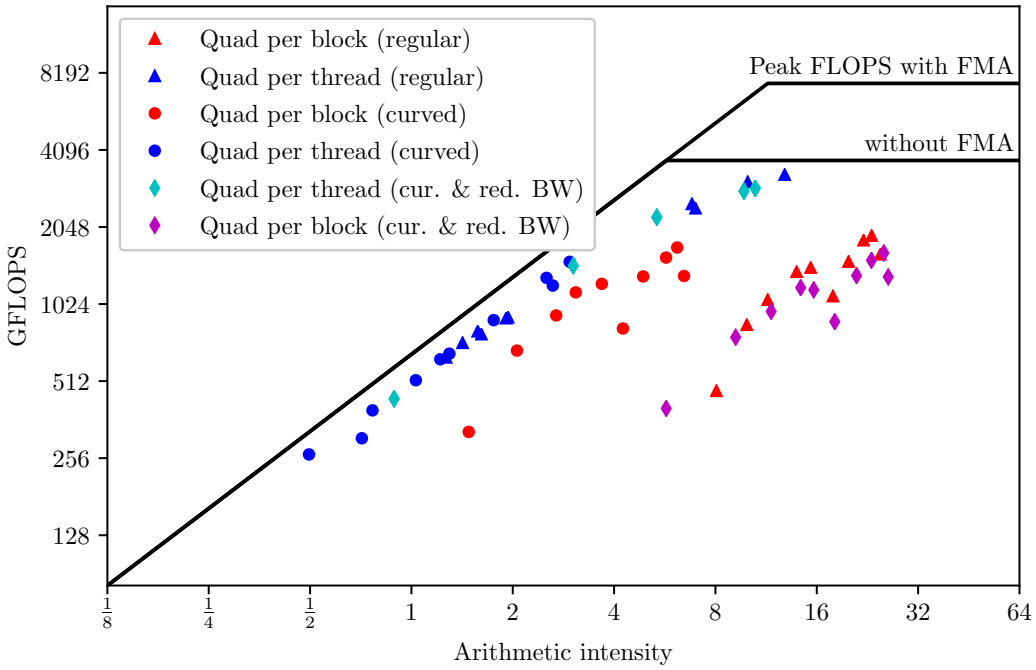

Figure 5.9: Roofline plot based on DRAM memory utilisation of thread-parallel and block-parallel kernels for regular and curved quadrilateral elements, as well as *reduced bandwidth* kernels for curved elements. Each dot corresponds to a different polynomial order P, with increasing arithmetic intensity from $P = 1$ to $P = 10$, within each group.

maximum theoretical FLOPS, indicating a very good utilisation of the GPU hardware. When comparing the arithmetic intensity of the same kernels between DRAM and L1 bandwidths, it can be observed that it is almost the same for the element-per-thread kernels. This indicates a streaming of the data through the GPU and very little data reusing. This can be expected, because we do not employ the shared memory as a scratch-space memory. For the element-per-block kernels we can observe a different picture, with the arithmetic intensity being considerable smaller for the L1 cache, indicating a higher level of data re-usage. Indeed, this is what we have specified by creating scratch spaces for each element on shared memory to store intermediate results.

To investigate closer why the performance of the completely vectorised kernels degrades so abruptly for polynomial orders over $P = 4$, we investigate a metric based on the usage of *local* memory. This memory space is placed on L1 cache and solely acts as a spillover memory bank for the registers. Indeed Figure 5.11 shows that the default kernels for curved and regular elements with $P < 4$ do not utilise *local* memory at all, as the registers are sufficiently sized for all variables. The higher the local memory overhead is, the more the FLOP rate of the kernels is degraded, which can be deduced when comparing the rates alongside the achieved FLOPS in Figure 5.6 and Figure 5.7.

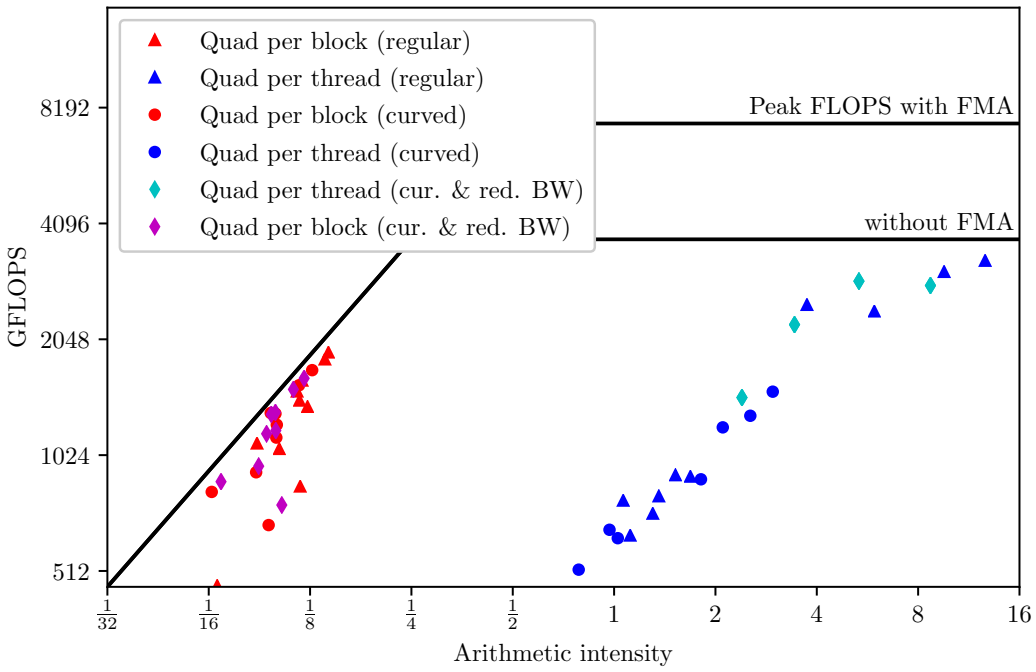

Figure 5.10: Roofline plot based on L1 cache utilisation of thread-parallel and block-parallel kernels for regular and curved quadrilateral elements, as well as *reduced bandwidth* kernels for curved elements. Each dot corresponds to a different polynomial order P, with increasing arithmetic intensity from $P = 1$ to $P = 10$, within each group.

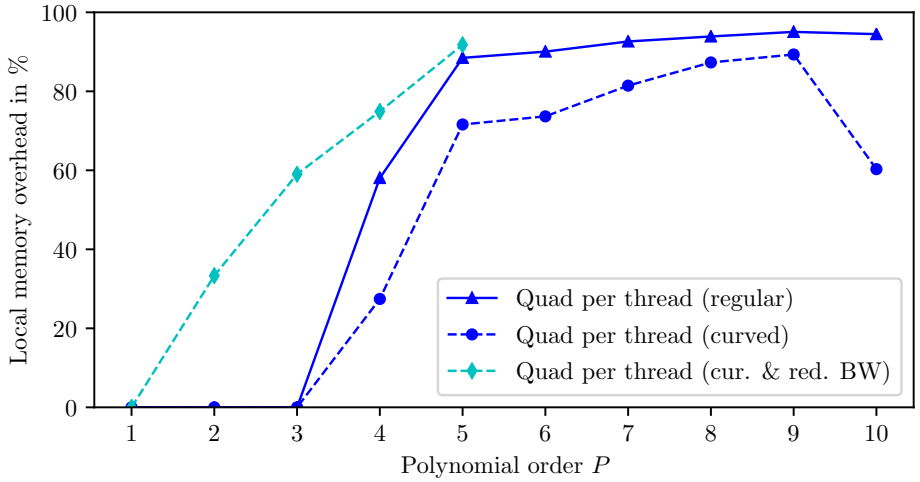

Figure 5.11: *Local* memory overhead of thread-parallel kernels for quadrilateral elements. All kernels are employing the optimal memory space for utility data.

5.4.2 Triangular elements

We now repeat the equivalent performance evaluation for triangular elements and present their results. We expect to see similar, but different results as the balance between memory footprint and number of operations is shifted compared with quadrilateral elements. Each triangle has fewer modes or DoFs for the same polynomial order: a quadrilateral element of order P has $M_{quad} = (P+1)^2$ modes, whereas a triangular element of order P has only $M_{tri} = (P+1)(P+2)/2$ modes. The number of modes roughly scales with the required DRAM bandwidth. However, as we employ tensor based expansion bases, we have almost as many quadrature points to process for triangles as for quadrilaterals, the difference just stemming from the use of the Gauss-Radau quadrature with $Q_{GR} = P+1$ points for the collapsed dimension of the triangles instead of the Gauss-Lobatto-Legendre quadrature with $Q_{GLL} = P+2$ points. The number of operations per DoF for each triangular element is therefore higher and is further increased by the need to do additional coordinate transformations between the standard element and the collapsed element, as explained in Section 5.2.2. The basis expansions and their derivatives for triangles however require more memory, as for a quadrilateral only two bases functions of the form $\psi_p^a(\xi)$ with $0 \leq p < P$ need to be loaded, whereas the triangular bases requires one of the two bases functions to be of the form $\psi_{pq}^b(\xi)$ with $0 \leq p < P, 0 \leq q < P - p$, which has M_{tri} entries. This will increase the required memory bandwidth on L1 cache, unless the *constant* memory space is used.

Overall in Figure 5.12, we can observe the same performance trends for regular triangles as for regular quadrilaterals. The vectorised kernels with per-thread parallelism is superior for lower polynomial orders, whereas for higher polynomial orders the per-block parallelism is faster. Again it is beneficial to employ the *constant* memory space in case of the vectorised kernels and the *shared* memory space in case of the block-parallel kernels to load the utility data from. As before, the best kernels achieve more than 42% of the theoretical FLOP maximum. However, the vectorised kernels' performance only drops significantly for polynomial order $P > 5$ and not $P > 4$, suggesting the register pressure for triangles is less than for quadrilaterals. Additionally, the performance difference between the thread-parallel and the block-parallel kernels for higher polynomial orders is not as significant for triangles (being almost identical for $P = 6$ and $P = 8$) as for quadrilaterals, due to a combination of the factors discussed above.

For curved triangular elements, the same trend as for curved quadrilateral elements are observed, as shown in Figure 5.13. However, the achieved FLOP rate for the block-parallel kernels of higher polynomial orders is significantly reduced compared with quadrilaterals. This could be explained by the triangular C^0-basis that contains additional boundary modes that need to be processed individually, so that at these instances many threads are running idle.

Looking at the triangular element kernels' performance in terms of throughput (DoF/s) in Figure 5.14, the same trend as for quadrilateral elements can be observed. The mag-

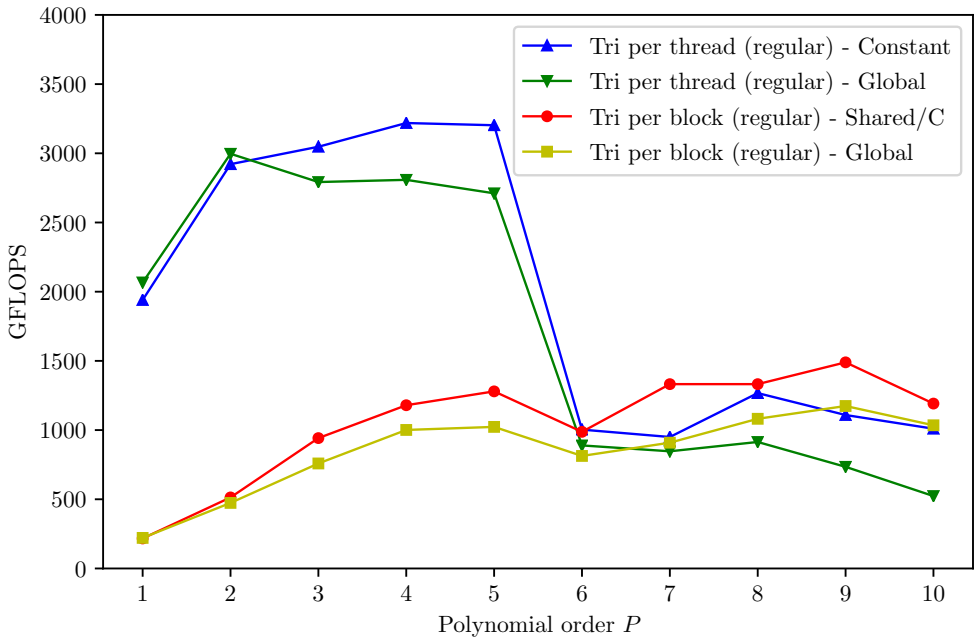

Figure 5.12: Performance of thread-parallel and block-parallel kernels for regular triangular elements. The usage of *global*, *constant*, and *shared* memory space for utility data is compared.

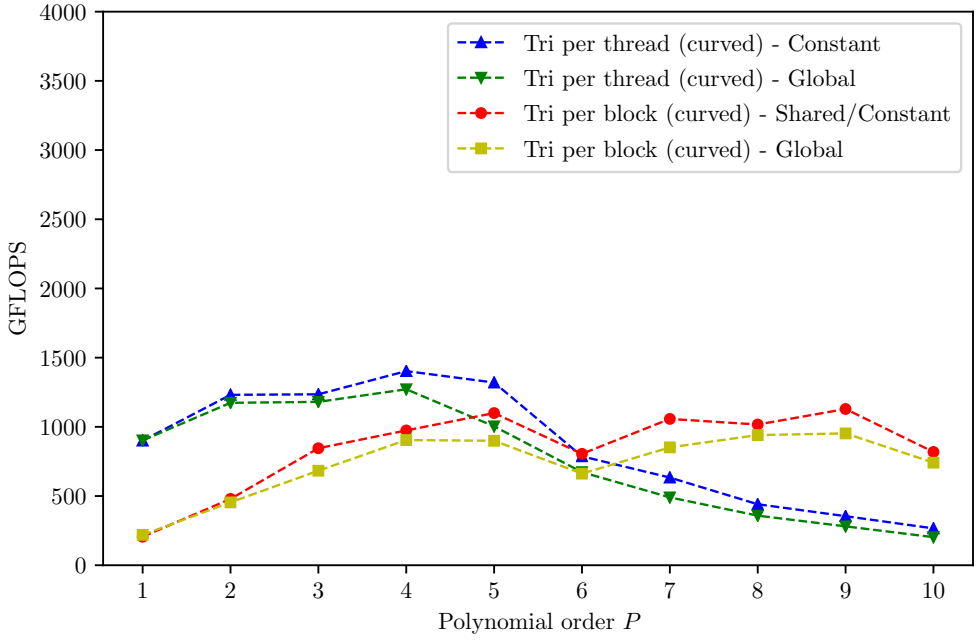

Figure 5.13: Performance of thread-parallel and block-parallel kernels for curved triangular elements. The usage of *global*, *constant*, and *shared* memory space for utility data is compared.

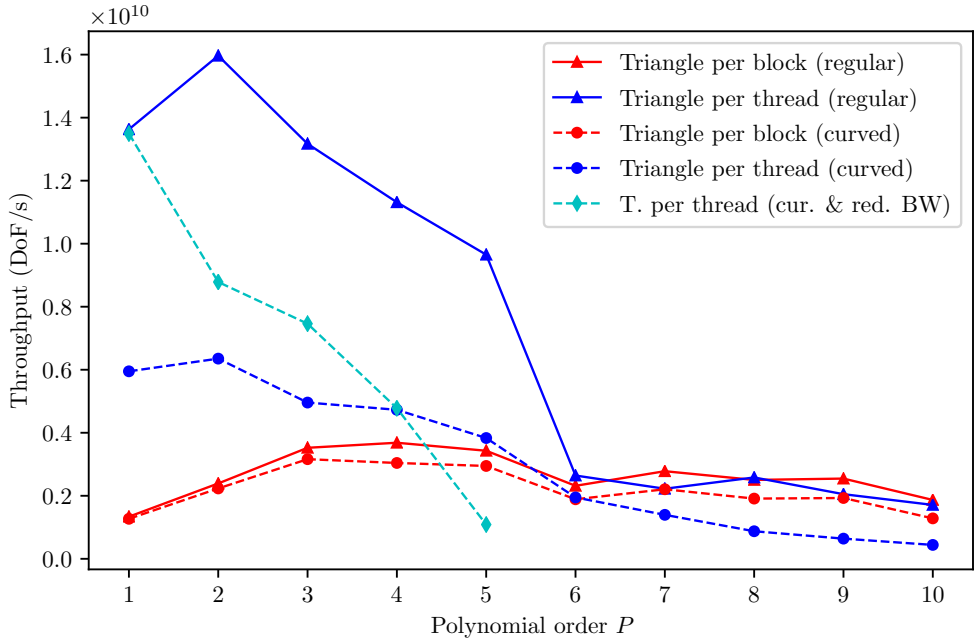

Figure 5.14: Throughput of thread-parallel and block-parallel kernels for regular and curved triangular elements, as well as *reduced bandwidth* kernels for curved elements based on isoparametric or subparametric mappings. All kernels are employing the optimal memory space for utility data.

nitude of the throughput however is about 20% lower than for quadrilaterals. The main factor towards this result will be the higher number of operations required for each DoF. Here the reduced bandwidth kernel for curved triangles, that computes the Jacobian $|\boldsymbol{J}^e|$ and the derivative factors $\Lambda\left(\frac{\partial\xi}{\partial x}\right)$ from the geometric mapping χ^e, achieves a higher throughput than its default version for $P < 4$, by around 25-30%.

On the two roofline plots in Figure 5.15 and Figure 5.16, these reduced bandwidth kernels show very high FLOP rates of 40-50% of peak FLOP on the Titan V GPU. These kernels are also not only bandwidth-bound anymore, but show a mix of stall reasons in the profiling results. Generally the same performance results as for quadrilaterals are displayed: the per-thread parallel kernels are bandwidth bound at DRAM level, as they stream most of the data through the kernel and do little data-reusing, whereas the per-block parallel kernels are memory bound at L1 cache, and do more data reusing on higher cache levels due to using *shared* memory arrays used as workspaces.

As before for quadrilaterals, we consider the *local* memory overhead in Figure 5.17. Comparing the data with the achieved FLOPS in Figure 5.12, we can again recognise the two regimes: low *local* memory utilisation and register pressure resulting in high FLOP rates for $P \leq 5$, as opposed to high *local* memory usage and lower FLOP rates for $P > 5$.

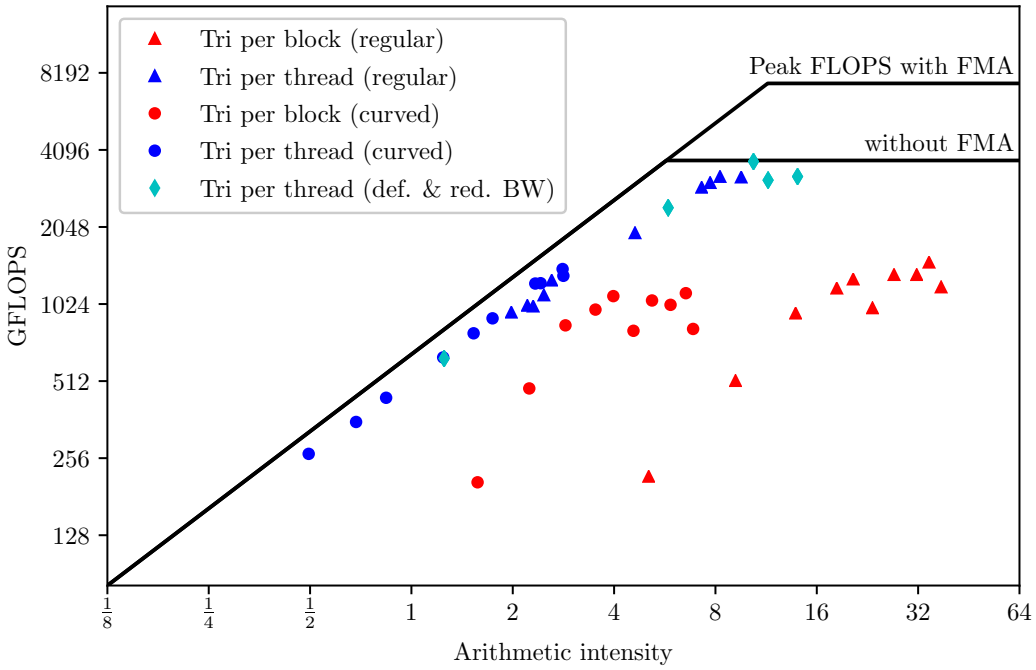

Figure 5.15: Roofline plot based on DRAM memory utilisation of thread-parallel and block-parallel kernels for regular and curved triangular elements, as well as *reduced bandwidth* kernels for curved elements. Each dot corresponds to a different polynomial order P, with increasing arithmetic intensity from $P = 1$ to $P = 10$, within each group.

5.4.3 Comparison of *CUDA* versus *OpenACC*

A fundamental consideration when judging a performance portable programming model against an architecture specific model is the tradeoff between maintainability and performance. As explained in Chapter 4, the performance comparison needs to be based on a representative case, and not just on a BLAS kernel benchmark. To address this question, we therefore compare the performance of the Helmholtz operator for triangular curved elements between two of our existing implementations. The first one is the portable *OpenACC* implementation taken from Chapter 4. The second one is one of the *CUDA* kernels developed in this chapter. We take implementations that load the upper triangular matrix of the matrix product of the derivative factors $(\boldsymbol{J}^e)^{-1}(\boldsymbol{J}^e)^{-T}$ (three arrays) and the Jacobian determinant $|\boldsymbol{J}^e|$ from global memory. We compare both the vectorised evaluation, in which each elemental operation is assigned to one *CUDA*-thread, as well as a block-parallel evaluation, in which each element is assigned to one *CUDA*-block. For the *OpenACC* versions, we employ the *pgi*-compiler version 19.4 with the optimisation flag *-O3*. All benchmarks are run on a Nvidia Titan V GPU.

The results for the completely vectorised versions are presented in Figure 5.18. The *OpenACC* baseline implementation is directly taken from Chapter 4. However, in order to investigate the performance using syntaxes and code constructs that are close to *Nektar++* and thus allow code maintainability, not all performance optimising measures have

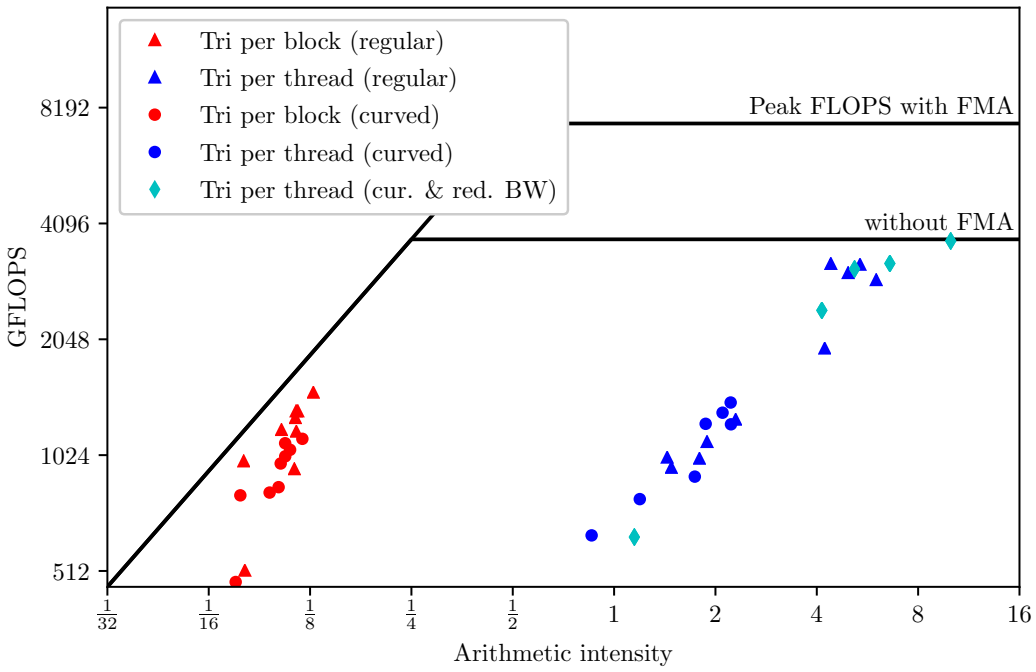

Figure 5.16: Roofline plot based on L1 cache utilisation of thread-parallel and block-parallel kernels for regular and curved triangular elements, as well as *reduced bandwidth* kernels for curved elements. Each dot corresponds to a different polynomial order P, with increasing arithmetic intensity from $P = 1$ to $P = 10$, within each group.

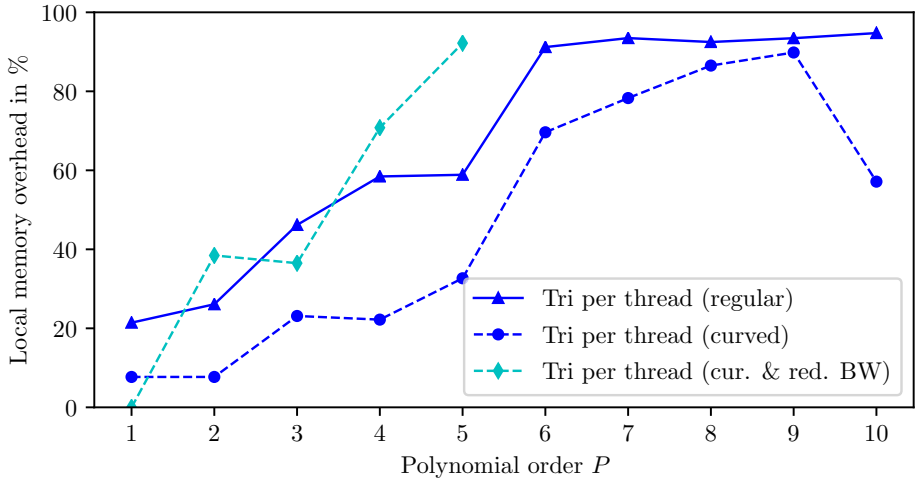

Figure 5.17: *Local* memory overhead of thread-parallel kernels for triangular elements. All kernels are employing the optimal memory space for utility data.

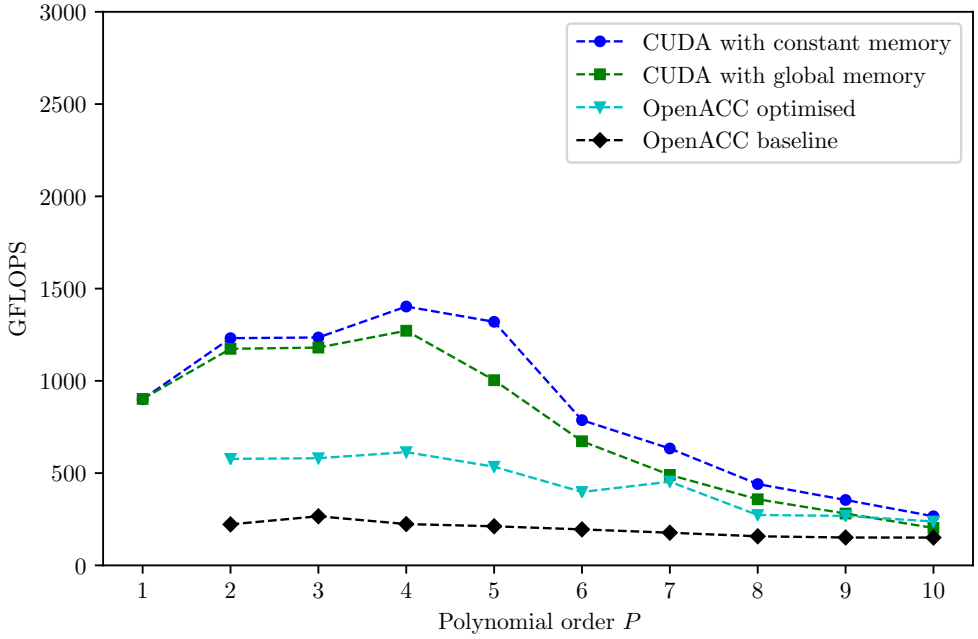

Figure 5.18: Performance of thread-parallel kernels for curved triangular elements, comparing baseline and optimised *OpenACC* and *CUDA* implementations.

been taken. Since in this chapter we prioritise the performance aspect above maintainability and portability, an optimised *OpenACC* implementation is developed, to allow for a fair comparison with the *CUDA* implementation. With this in mind, all loop sizes and array length have been set to static allocations using templating, workspace sizes have been reduced as much as possible, and calls to BLAS functions have been replaced with the custom operations. This way the compiler can employ its code optimisation for each individual kernel depending on its polynomial order. This optimisation improves the performance measured in FLOPS more than twofold between the baseline and the optimised *OpenACC* versions for all polynomial orders. The optimised *OpenACC* version can now be fairly compared against the *CUDA* version, that loads utility data from the global memory space. Hence, these two versions just differ in the utilised programming language and the compiler. For the lower polynomial orders $P \leq 5$, a significant performance improvement of the *CUDA* version of more than factor two compared with *OpenACC* can be observed. For the higher polynomial order $P > 6$, the differences become negligible. However in this range the chosen parallel mapping of element-per-thread is not as performant as the mapping element-per-block, which is investigated in more detail later. Using the *CUDA* programming model further allows to utilise the constant memory space to load the utility data that is constant across all elements. Especially for polynomial order $P = 5$, valuable register space is saved and higher FLOPS counts are achieved.

The results for the block-parallel implementations are presented in Figure 5.19. Here we only developed an optimised *OpenACC* version, and its performance is shown to increase almost linearly with polynomial order P. Our *CUDA* kernels however are perform-

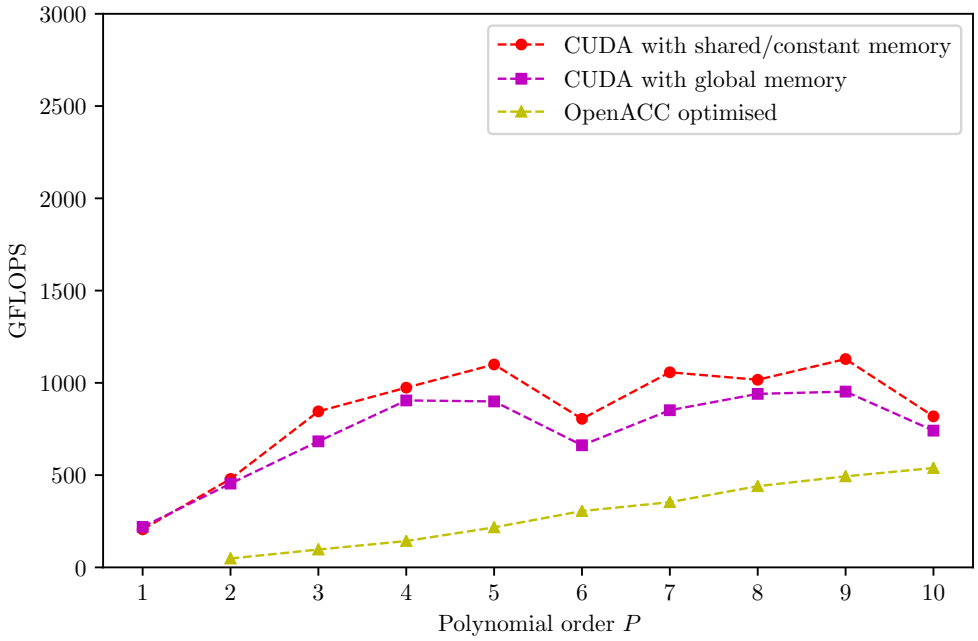

Figure 5.19: Performance of block-parallel kernels for curved triangular elements, comparing baseline and optimised *OpenACC* and *CUDA* implementations.

ing two to five times better for $P < 10$, indicating that the *OpenACC* specifications, or compilers, are not able to deal with these kind of algorithmic structures efficiently.

The same results are observed when considering the throughput of all our optimised implementations in Figure 5.20. For all polynomial orders in the range $1 < P \leq 10$, the most performing *CUDA* implementation outperforms the most performing *OpenACC* implementation by about a factor of two.

In addition to the performance results presented, we would also like to comment on the usability of *OpenACC* in the context of this work. Here, we were employing templating for various parameters and especially for the polynomial order of the element, which is a large performance factor. However, placing the *OpenACC* compiler directives (or *pragmas*) inside the code-base often resulted in ambiguity with respect to the *CUDA* code that the compiler developed under the hood and which we checked using the compiler flag *-Minfo=accel*. Only after merging some functions from different levels of the call tree, all arrays were placed in the intended memory spaces (i.e. workspace arrays in *shared* memory). As a consequence, our resulting *OpenACC* code showed a very different structure and syntax than the original *Nektar++* code, severely compromising the effectiveness of *OpenACC* to incrementally accelerate a legacy code-base.

5.5 Conclusions

We have investigated two options of mapping the parallel execution of the two-dimensional elemental Helmholtz operator in a high-order FEM setting to the parallel GPU hardware

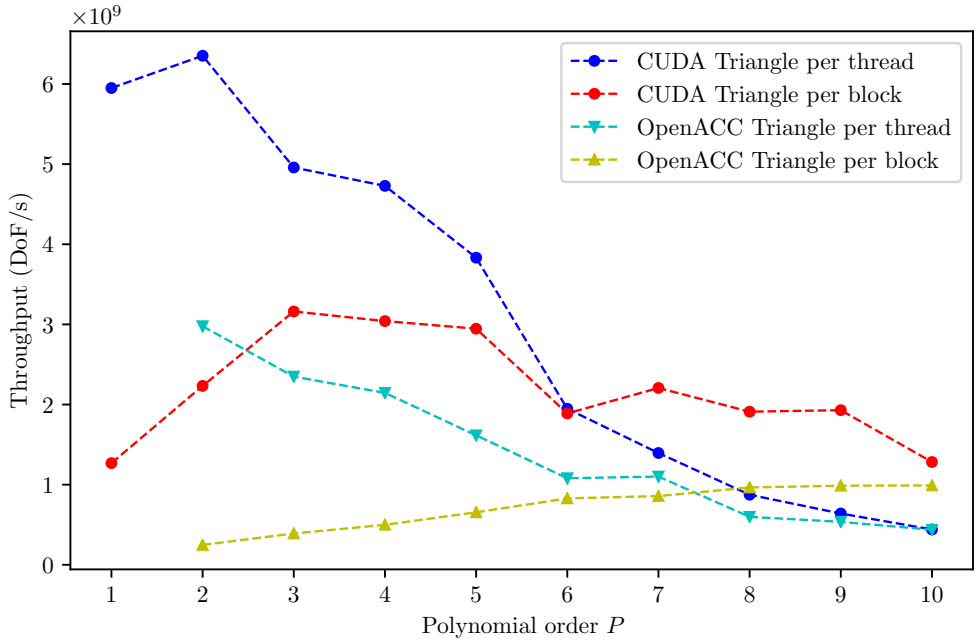

Figure 5.20: Throughput of thread-parallel and block-parallel kernels for curved triangular elements, comparing optimised *CUDA* against optimised *OpenACC* implementations.

using the *CUDA* programming model. The first option maps each elemental operation to one *CUDA*-thread for a fully vectorised kernel, whereas the second options maps each elemental operation to a *CUDA*-block, where we employ nested parallelism by assigning each quadrature point or mode to a *CUDA*-thread. While both options of parallel mapping of high-order operators have been considered before, on both CPU and GPU systems, no systematic comparison has been conducted before to the best of our knowledge.

We evaluated a number of kernel parameters with a templating approach in order to enable the most effective compiler auto-optimisation. The most prominent parameter is the polynomial order of the elemental evaluation, because its choice scales the arithmetic intensity of the algorithms and is thus a crucial performance factor. Overall our results show that the first option is beneficial for small quadrilateral elements with low polynomial orders $P < 5$, and for triangular elements of orders $P < 6$. For higher polynomial orders, the block-parallel option achieves better performance.

As a second template parameter, we developed bespoke kernels for straight-sided and for curved elements. Since the straight-sided elements have a constant Jacobian matrix, these kernels benefit from a substantially decreased memory bandwidth compared with curved elements. Indeed the kernels for straight-sided elements achieved a significantly higher throughput. Outside the scope of our work, this is a finding that should be taken into account when devising mesh-optimisation algorithms, as for only slightly curved elements, the slightly increased mesh quality might not outweigh the higher arithmetic costs of the solver compared with a straight-sided element.

Our implementations rely on a number of constant utility data, that could potentially

be loaded into the registers from a variety of memory spaces on the GPU. Our analysis of this third template parameter show that placing such data into the *constant* and *shared* memory space leads to performance gains compared with only using the *global* memory space.

Ultimately, the vectorised thread-parallel option achieves near-optimal performance of more than 40% peak FLOPS, but quickly approaches the register limits with increasing polynomial order, after which the performance sharply degrades. The block-parallel option cannot fully fill all the threads of a warp for low polynomial orders, but for higher orders with a higher arithmetic intensity good performances of up to 25% peak FLOPS for quadrilaterals and up to 20% for triangles are achieved. Further analysis of our performance benchmarks using so called *roofline* models show the thread-parallel mapping is DRAM memory-bandwidth bound, whereas the block-parallel mapping is L1-cache bandwidth bound.

Another main motivation for our work is to study strategies to accelerate large legacy software frameworks. Often performance portable models like *OpenMP*, *OpenACC*, or *Kokkos* are considered, because they appear to be easier maintainable in an environment of heterogeneous hardware systems. However such a choice will induce a performance penalty compared with hardware-specific kernels like *CUDA* for GPUs. As far as we are aware, this penalty has not been quantified before in the context of high-order spectral element methods. Our results show that optimised *OpenACC* kernels achieve only about 50% of the performance of optimised *CUDA* kernels for curved elemental Helmholtz operator on triangular elements.

In terms of maintainability, the initial effort of implementing the *OpenACC* kernels has been as involved as implementing the *CUDA* kernels. Considering that each hardware type will have a different performance profile, different parallel mappings like the ones studied in this work will have different optima. If performance is a consideration, this further emphasises the requirement for maintaining specific kernels for different hardware systems. We would argue that because of these reasons performance portable programming models are less suitable in practise in the investigated context.

Having developed our optimised elemental evaluation kernels, these also serve as benchmarks to investigate alternative algorithms for the same mathematical problem. Since the default algorithms studied here are memory bound, we investigated ways to reduce the required bandwidth by performing additional arithmetic operations. These options have not been considered in detail previously, because algorithms with the lowest operating count have most often been the fastest on traditional hardware. For high-order elemental evaluations, a path can be established by not pre-computing and storing the complete Jacobian matrix (that is the geometric mapping between the standard and the curved element) and its determinant. Instead we can load only the geometric mapping function itself, which saves 60% of the memory bandwidth for two-dimensional elements and recompute the Jacobian matrix and its determinant during each execution of the elemental

operator. We present results that are indeed up to 25% faster than the default algorithms for the vectorised elemental evaluations of low polynomial order.

In order to yield optimal performance, each kernel should be implemented based on many templating parameters like deformation, polynomial order, or memory space, resulting in dozens or even hundreds of targets for each kernel that need to be compiled. In this situation, just-in-time compilation frameworks should be investigated, to keep the compilation time reasonable, while still benefiting from bespoke kernel implementations.

The elemental Helmholtz kernels evaluated in this work form the most central core part of an implicit solver for the Helmholtz equation. The next layer of the method will be the global Helmholtz operator, which for a continuous Galerkin discretisation consists of a scatter operator, the considered elemental Helmholtz operator and a gather operator. It will be worthwhile to investigate how these three operators can be combined the most efficient way, especially in terms of data and memory management on a GPU system.

The next steps would be the evaluation of three-dimensional elements in the same fashion as in this chapter. An open question to address will be if the complete vectorisation of the elemental operator can also be beneficial for improving the performance in the 3D case.

Chapter 6

Conclusions and future work

This thesis has investigated the question of how to implement high-order finite element codes for modern shared memory architectures, such as individual multi-core CPUs and GPUs, in a portable fashion. Three main themes are repeated throughout this thesis: the hardware on which to execute the codes, the algorithms that solve the physical or mathematical problems, and the programming language that expresses these algorithms for the underlying hardware. It became apparent, at least since implementing practical examples, that these three aspects of hardware, algorithms, and programming model need to be co-designed.

However, there is a hierarchy of these three aspects in terms of the potential to affect or influence them in order to improve the performance of implementations. The available hardware has to be taken more or less as a given for computational scientists or software developers. Currently, there are foremost the diverging interests of the 'classical' high-performance computing community and the emerging machine learning community in terms of hardware design, as laid out in Section 2.1.4. The programming languages can be influenced to a higher degree, but it still needs larger interest groups to push forward new developments. The paradigm *Kokkos* [17] for example, that we employed in this work, is developed by Sandia National Labs, which as a whole has huge financial and manpower leverage. The aspect which individual research groups or development teams can influence the most is adapting their software frameworks. This is what we have attempted in this work, and the lessons learnt are presented here. Influence is exerted by selecting suitable programming models and adapting the algorithms from the relevant scientific domain to suit the chosen hardware systems in the best possible way.

We conducted all our investigations within the context of high-order spectral element codes, but aimed at presenting our findings in a general fashion to be applicable to other areas. Our findings on algorithmic design of our methods are presented in Section 6.1 and our conclusions on selecting a suitable programming paradigm in Section 6.2. Naturally, not all aspects can be investigated to completion within a thesis, hence we propose some directions for future work in Section 6.3.

6.1 Algorithmic developments for high-order finite element codes

Section 2.1 reviewed the current hardware landscape, which is dominated by multi-core CPUs and GPUs. Both architectures are fundamentally different from CPUs that were released before 2005, when CPUs had no vector lane and only a single core. However, the implementations that we considered at the starting point of our investigations were designed when single-core CPUs were still ubiquitous. We have then explored paths to adapt our implementations towards both modern hardware systems in Chapters 3 and 4, but also targeted GPU systems specifically in Chapter 5.

The aim of the algorithmic adaptations was to use the employed hardware more efficiently, to achieve speed-ups in execution time and to reduce energy consumption, or more broadly speaking to save time and money. In order to monitor those savings, we developed a novel cost metric for hardware utilisation, that is either based on cloud computing prices of bare-metal systems in Section 3.4.7, or on purchase prices with a linear depreciation of hardware systems in Section 4.4.3. Alternatively the power consumption of operating the system could be factored into the equation. For all our investigations we realised both time and cost savings.

We considered multiple algorithms that are constituents of high-order finite element codes, from the domains of mesh optimisation and incompressible Navier-Stokes solvers. We purposely worked with more light-weight solvers that allow for a simpler development environment, to be able to compare multiple implementations and perform extensive benchmarking tests. However, for these benchmarks to be meaningful, the simplified solvers still needed to be representative of the complete solvers. To this end, we accelerated a mesh-optimisation method in Chapter 3, and a Helmholtz solver mini-application in Chapter 4.

It emerged that the porting exercise can be broken down into two steps. In the first step, the implementations need to be prepared for the two levels of shared-memory parallelism and the limitations this structure brings in terms of architecting a solver method. In the second step, the prepared algorithms can be expressed in a suitable parallel programming language. We will discuss our conclusions regarding this point in Section 6.2.

The first step became necessary because the new hardware does not only perform SISD instructions, but increasingly executes SIMD instructions. GPU hardware further has only reduced execution logic circuits, so executables need to be specified in a more straightforward way. This can pose a limit on object-oriented programming (OOP) features that can be applied in code-bases targeting GPU architectures. Our main algorithmic adaptations are described in the following.

6.1.1 Re-factoring of operations and data structures

Basically all OOP logic needs to be specified outside of GPU kernels, however, the granularity of these logic instructions was found to be too fine for the considered algorithms, so that individual kernels did not possess enough workload to be efficient. We followed approaches like amalgamating multiple elemental evaluations into collective operations to increase the kernel workloads, as first presented in reference [84]. The main effort for the mesh-optimiser in Chapter 3 was to re-factor the code structure from associative OOP data structures and function calls to plain-old-data (POD) structures that are required for the *CUDA* programming model that targets GPUs.

6.1.2 Reduction operations and element colouring

Algorithmic adaptations were also found to be critical for all kind of reduction operations, as these are inherently difficult to vectorise and write conflicts or so called race-conditions need to be avoided. To mitigate this performance bottleneck on GPUs, we employed element colouring that would allow to process multiple elements in parallel. In our investigations of a *gather* kernel in Chapter 4, we found that this adaptation reduced the execution time on the GPU by about two orders of magnitude and made it comparable to the multi-core CPU execution time.

6.1.3 Interleaved memory layout

For the vectorised elemental operations in Chapters 4 and 5, we developed an interleaved memory layout. This was imperative in order to achieve a contiguous memory access and thus fully utilise the GPU memory bandwidth, instead of only 1/8 of its maximum in case of current GPUs. To ensure flexibility between architectures, we specified the SIMD vector-width as a template parameter of our kernels. A number of syntactical changes were also necessary in order to call strided BLAS functions that can operate on the interleaved data instead of calling regular BLAS functions, and to use templating over the polynomial order of the elemental evaluation to create static workspace arrays, that are critical for efficient GPU executions.

6.1.4 Multi-level parallel mapping and reduced bandwidth kernels

In Chapter 5, we compared two options of mapping multi-level parallelism to GPU systems. Both resulting kernels were memory bound, so we developed an alternative kernel to reduce the memory bandwidth by performing additional arithmetic operations. We found that the best choice between these three versions depended on the arithmetic intensity of the kernel, that varies with the polynomial order. This result shows that different algorithmic options need to be considered and assessed anew for each modern parallel computing system. Moreover, algorithms with the smallest number of arithmetic operations are not automatically the fastest algorithms.

6.2 Performance portable programming models for shared memory systems

Section 2.2 discussed three main requirements for programming models: usability, portability, and performance. Considering these requirements, we reviewed a number of parallel programming languages for CPUs and GPUs, which can be classified either as portable or architecture-specific. While in an environment of multiple different hardware systems a portable programming model is beneficial to simplify the maintenance of a code-base, its downside is a potential performance penalty. We therefore outlined three different broad options to maintain parallel implementations of large scientific software frameworks:

A) Maintaining one code-base with a portable programming language, that is as *performance* portable as possible.
B) Maintaining two code-bases of computational hotspots for the two currently dominant classes of hardware: multi-core CPUs and GPUs, where each of them can achieve near optimal performance.
C) Designing a new framework from scratch, consisting of a high-level interface and a translation level to automatically compile multiple architecture-specific executables.

In this thesis we considered option A) in Chapters 3 and 4 and option B) in Chapter 5. We neglected option C) as we wanted to only explore strategies to accelerate already existing large legacy frameworks.

6.2.1 Preparation of code-structures for parallelisation

Without adapting the algorithms and the code-base, it might be possible to implement new parallel instructions, but we found that this could often lead to performance degradation. This was either due to unfavourable reduction operations or non-contiguous memory accesses. Similar steps need to be undertaken, even in the case of only targeting modern multi-core CPU systems.

While the evolution from single-core SISD CPUs to multi-core CPUs with SIMD vector-lanes has been gradual, the overall change of CPU systems since around the year 2005 has been significant. It can be argued that CPU hardware vendors preferred to market their products as an evolutionary development, to keep the barrier to entrance low. Compiler auto-optimisation options allowed to further mask any code-adaptations that might actually be necessary. Alternatively, the use of optimised BLAS kernels might provide a route to keep optimally performing implementations without changing the core code-base. For the methods we considered within high-order spectral element codes this route was blocked, as they cannot be efficiently mapped to standard BLAS calls. In conclusion, relying only on the compilers will not be sufficient to exploit the new parallel hardware properly for the following reasons that we found in our investigations.

When stepping up from single-core to multi-core CPUs, it is possible to extend the MPI model and map each MPI rank not to a single node, but to a single core. However

at the strong scaling limit, where communication costs dominate, no speed-up in time-to solution can be achieved, as still only one thread processes a single chunk of work. Only with a shared-memory programming model like *Pthreads* or *OpenMP*, multiple threads on the same shared-memory system are able to work on the same chunk of work, thus reducing the time to solution. A similar argument applies for the CPU vector-lanes. While compilers can attempt to do auto-vectorisation, ideally we want to specify the work on the vector lanes explicitly. In our work on the portable Helmholtz solver mini-application in Chapter 4, the compiler clearly did not achieve the ideal vectorisation pattern. As a remedy, another extension of the programming model is required, for example as investigated in reference [82] through the use of compiler intrinsics and operator overloading. Additionally, when we ran benchmarks of the mesh-optimiser on a CPU-type accelerator, the Intel Xeon Phi, we observed that efficient memory access of the type that is mandatory for GPU implementations enabled an order of magnitude faster runtimes compared with associative data structures.

6.2.2 Comparing three portable programming models

While our investigation on the mesh-optimisation method in Chapter 3 mostly gave answers on the first steps of preparing an implementation for GPU-acceleration, our investigations on the Helmholtz solver was aiming at providing answers on which portable programming model is most suitable when pursuing option A). To this end we compared the models *OpenMP*, *OpenACC*, and *Kokkos* with both multi-core CPUs and Nvidia GPUs compilation targets. A major limitation for the usability of these models is not their specifications, but actually their open-source compiler support, that is critical for an academic software framework.

We found that the differences in terms of performance between the three models are rather negligible, the *OpenMP* version with the *llvm* compiler being the fastest on CPUs and the *OpenACC* version with the *pgi* compiler being the fastest on Nvidia GPUs. The *Kokkos* version with the *gcc* compiler was about 30% slower against the faster option on both architectures. Syntactically and in terms of implementational efforts all three versions were comparable. However, the *OpenMP* support for GPUs was still very limited, so this model was not portable in practise at the time of conducting the study.

The compiler support is expected to be a general issue of portable programming models, as it will probably take a few years after the introduction of new hardware types and the development of new specifications to actually have open-source compiler support available that is robust enough to be used in practise. As a case in point, it also took years for the *pgi* compiler to implement a multi-core CPU backend for the *OpenACC* instructions. Similarly, none of the three models provided a backend that could target AMD GPUs at that time.

In summary, if we had to decide for one portable programming model, we would chose *OpenACC* with the *pgi* compiler for its robustness and performance.

6.2.3 Assessing a GPU-specific programming model

In Chapter 5, we worked with *CUDA*, a GPU-native programming language that is an extension to *C++*. This work allowed us to explore option B), that is maintaining hardware-specific kernels for computational hotspots within a broader framework.

We started off from a CPU implementation, that was already prepared for a two-level parallelisation and was setup for parameter templating of the kernels. Compared with the directive-based model *OpenACC*, we experienced *CUDA* to allow more direct control, especially of data placements. In our work with *OpenACC*, we often found ambiguity in terms of how the compiler would interpret the directives we specified in the code-base. The more direct control allowed us to specify the memory space of our method's utility data to be either *shared* memory or *constant* memory, thus realising performance gains of 10-20%.

We further compared the performance of the elemental Helmholtz operator between the *OpenACC* and the *CUDA* implementation and found that for the same algorithm and code-structure the *CUDA* version was performing about twice as fast. We note that the *OpenACC* implementation in this comparison was not a carefully accelerated initial CPU implementation with minimal adaptations, but a rigorously optimised GPU-code.

The developed GPU kernels could be accompanied by the vectorised CPU kernels presented in reference [82] in a common framework. This way kernels of the computational hotspots for multi-core CPUs with AVX vector lanes, as well as for GPUs would be available, that can achieve near optimal performance. Still, most of the code-base that provides functions for the creation and processing of auxiliary data, or higher level solver functions (i.e. for Newton-Krylov solvers or Runge-Kutta time-stepping) could be expressed in a single portable framework.

6.2.4 Weighing up between portable versus hardware-specific programming models

As we have shown, there is no simple way to achieve acceleration of a serial code base that was designed for single-core CPU systems with minimum effort by employing a portable programming model.

We acknowledge that using a portable model like *OpenACC*, *Kokkos* or *OpenMP* allows porting an application with the least effort and will result in the easiest maintainability of the code-base. However, we already showed in our work with the implicit Helmholtz solver in Chapter 4, that algorithmic divergence between CPU and GPU compilation targets is required in order to achieve performance portability. Further, even for multi-core CPUs with AVX vector lanes and Nvidia GPUs that have been on the market for almost a decade, the portable models could not address or exploit all performance relevant features, such as AVX vectorisation or memory placement. Only by using compiler intrinsics for CPUs with AVX vector lanes, or the *CUDA* model for Nvidia GPUs, near optimal performance could be achieved, which showed more than twice the performance of the best portable

implementations. We would summarise that while the low barrier to entry of directive-based programming models allows to quickly parallelise an application (but not necessarily achieve cost or time reductions), it was more user-friendly to use a model with specific instructions to yield near-optimal acceleration.

It is often argued that a portable model will additionally allow a simpler switch to newly emerging architectures, however under close examination this argument becomes very weak: each class of new hardware will by definition be sufficiently distinct from other existing classes. It follows that they will have different performance profiles in terms of memory models and arithmetic operations, or potentially special function units. Initially there will only be native compilers for the new architecture and it can be expected to take years for the open-source community to support a compilation target for the new system. Additionally, the new architectures will most likely favour sufficiently different algorithmic types, otherwise it would be hard to carve out an economically viable niche in the market. Currently the newly emerging systems are bespoke chips for machine learning applications that are often equipped with low-precision matrix-matrix-multiplication special function units. This however means that proper adaptations or redesigns of the traditional algorithms will be necessary in the first place. The simpler, faster, and more performant option in this case would be to write new native kernels for the computational hotspots and link them to the larger framework.

We would therefore propose to support portability only between different systems of the same class, i.e. between different CPU systems, and additionally between different GPU systems. In this case, the hardwares and their performance profile are most likely similar, so that a portable programming model can also be *performance*-portable with much less effort. To summarise, we would currently propose to maintain two types of kernels for the computational hotspots: a first one for multi-core CPUs using the robust and performant *OpenMP* model and compiler intrinsics in cases where explicit vectorised instructions are required, and a second one for GPUs, using *CUDA* for Nvidia GPUs or the new programming model *HIP*(*C++ Heterogeneous-Compute Interface for Portability* [40], that is portable between AMD and Nvidia GPUs.

6.3 Future work

Our grand vision would be to create a complete performance-portable Navier-Stokes solver framework. As a first aspect, there are still a number of developments necessary to ensure all required algorithms are adapted to current parallel shared-memory systems. Independently of the specific future development of the hardware landscape, algorithms with multi-level parallelism and reduced memory footprint are needed. As a second aspect, a number of software features need to be developed that allow to automatically select the most performant kernels depending on the underlying hardware. While we cannot rely on compiler auto-optimisation, the code-base itself must exhibit enormous flexibility to balance the processing of data across a heterogeneous system for an optimal hardware

utilisation.

We would propose to continue the work on the *CUDA* kernels and integrate those into a wider high-order spectral element framework. Existing frameworks such as *Nektar++* would be suitable due to their maturity and extensive range of implemented pre-processing, utility functions, and post-processing capabilities. This way many functionalities can be re-used, while only the code-base for the computational hotspots would need to be re-developed. The *CUDA* kernels could form the basis to address Nvidia GPU architectures. However, it should be considered to switch to the recently introduced *HIP* model [40], that follows the *CUDA* syntax and thus allows for a simple translation. The resulting code-base can be compiled with the *nvcc* compiler to target Nvidia GPUs, but also with the *hcc* compiler to target AMD GPUs. To address CPU architectures, the framework would be suitably paired with the optimised vectorised CPU kernels based on compiler intrinsics and variable vector-widths, as presented in reference [82]. Such a software architecture would require an interface layer, that manages the selection between kernels for different hardware systems. It would also serve as an interface for additional optimised kernels for future hardware architectures.

In terms of natural further steps, the GPU kernels should be extended to three-dimensional elements to investigate which type of parallel mapping between algorithm and hardware is beneficial. As three-dimensional elements have a different balance of memory bandwidths and arithmetic operations we might see a different performance balance.

For collapsed elements with a tensor product basis, a more suitable construction and storage of basis functions could be investigated. Considering triangles, so far the basis function is decomposed and stored as

$$\phi_{pq}(\xi_1, \xi_2) = \psi_p^a(\eta_1(\xi_1, \xi_2))\, \psi_{pq}^b(\xi_2(\eta_2)), \tag{6.1}$$

where ψ_{pq} is a two-dimensional tensor. We alternatively propose to decompose and store the basis as

$$\phi_{pq}(\xi_1, \xi_2) = \psi_p^a(\eta_1(\xi_1, \xi_2))\, f_p(\eta_2(\xi_2))\, \psi_q^b(\eta_2(\xi_2)), \tag{6.2}$$

with the necessary factor f_p to keep the expansions as polynomials in terms of the Cartesian coordinates, which is

$$f_p(\eta_2(\xi_2)) = \left(\frac{1-\xi_2}{2}\right)^p. \tag{6.3}$$

In terms of memory storage, the alternative formulation would reduce the requirement from $\mathcal{O}(P^2 + P)$ to $\mathcal{O}(2P)$. However, using the same Jacobi polynomial for each basis ψ_p^a and ψ_q^b impairs the conditioning of the numerical system, which might be counteracted by suitable preconditioning, though.

The proposed path to develop the complete incompressible Navier-Stokes solver would be to add more and more functions in a shell-like fashion to the CPU and GPU core-kernels. As the next layer, the integration of the elemental Helmholtz operations with the

scatter and *gather* operations, that map between local and global degrees of freedom in a continuous Galerkin setting, should be investigated. Combining these three operations opens up the opportunity to use the *scatter* and *gather* operation as a tailored way to load elemental data from global memory to higher cache levels or *shared* memory, where it is processed in the elemental operation. This would safe a significant amount of memory bandwidth, compared with the split into three kernels, as performed in Chapter 4.

Following that step, the implicit linear Helmholtz solver can be implemented. Here the opportunity arises to investigate different preconditioners, potentially with a block-diagonal structure, or even p-multigrid preconditioning techniques. The challenge for developing preconditioners for GPUs is the large memory requirement that quickly tends to bursts the DRAM limits. Evaluating the best convergence criteria for the iterative Newton-Krylov solvers also offers efficiency gains. It remains to implement the explicit time-stepping kernel for the advection terms, and kernels for the relevant pressure boundary conditions, to obtain the complete incompressible Navier Stoke solver that is based on the operator splitting scheme. Alternatively, the constituent functions of the Helmholtz kernel can be used to construct elemental operators for a compressible Navier-Stokes solver with a discontinuous Galerkin discretisation.

So far all investigations were based on homogeneous meshes of the same shape and polynomial order. To allow for more flexibility in mesh generation, for example using both structured prismatic boundary layer meshes and unstructured tetrahedral volume meshes, functionalities to deal with heterogeneous meshes have to be implemented. These have to be based on collecting groups of the same element types to allow for vectorised kernel executions. Another challenge will be to manage the larger variety of utility data, which might not entirely fit into the higher memory or cache levels on GPUs. These data could then be asynchronously swapped between kernel invocations for different element types.

Furthermore, recent hardware trends have led to the integration of more low-precision arithmetic units, which can offer an order of magnitude higher FLOP rates on certain systems. It might therefore be worth investigating which parts of a complete high-order spectral element solver could be executed with lower precision, while maintaining an acceptable overall precision. For iterative solvers, for example, this could be achieved by executing the first few iterations with lower precision until a to be defined convergence threshold is reached, before switching to higher precision for the last few iterations. Setting up the preconditioner in low-precision data types also provides a straightforward route to exploit.

Practical problems to be solved with the Navier-Stokes equation do not fit on a single shared-memory system. Hence the solver framework needs to integrate with distributed memory parallelisation. The most reasonable option is still to use the MPI interface and a domain decomposition approach. However, in an environment of heterogeneous clusters and nodes, there is huge potential to develop a new intelligent load balancing system. The domain decomposition would benefit from taking into account the different

elemental types within the mesh and attempt to cluster elements of the same type within one domain, to reduce the number of different kernel invocations for each sub-domain. This still needs to be balanced with the traditional rationale of minimising communication. On the algorithmic side, we would propose to develop a range of kernels for equivalent mathematical operations, but with different performance profiles and implemented for a range of architectures. On the hardware side, the performance profile of all constituents would need to be assessed with an automatic benchmarking using relevant kernels. The sub-domains, the individual kernels, and the hardware nodes could then be optimally mapped. This would ensure that all sub-domains and their element types are executed by the most suitable kernel on the most suitable computer architecture for optimal resource utilisation. Such a framework should further be outfitted with *just-in-time* compilation capability, so only the required optimised kernels need to be compiled, saving hundreds of compilation targets that would arise due to the large number of kernel parameters and potential hardware systems.

Bibliography

[1] G. Alzetta, D. Arndt, W. Bangerth, V. Boddu, B. Brands, D. Davydov, R. Gassmoeller, T. Heister, L. Heltai, K. Kormann, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The deal.II Library, Version 9.0. *Journal of Numerical Mathematics*, 2018.

[2] White Paper - Dissecting the Polaris Architecture. `https://www.amd.com/system/files/documents/polaris-whitepaper.pdf`, last visited on 03/06/2020, 2016.

[3] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini. MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*, 2020.

[4] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The deal.II finite element library: Design, features, and insights. *Computers & Mathematics with Applications*, 2020.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing*, pages 863–874, 2009.

[6] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. V. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, volume March, pages 138–149, 2016.

[7] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. In *WOLFHPC 2012 - 2nd Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 1–5, 2013.

[8] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software*, 38(2), January 2012.

[9] P. Bastian, C. Engwer, J. Fahlke, M. Geveler, D. Göddeke, O. Iliev, O. Ippisch, R. Milk, J. Mohring, S. Müthing, M. Ohlberger, D. Ribbrock, and S. Turek. Hardware-Based Efficiency Advances in the EXA-DUNE Project. In H.-J. Bungartz, P. Neumann, and W. E. Nagel, editors, *Software for Exascale Computing - SPPEXA 2013-2015*, pages 3–23, Cham, 2016. Springer International Publishing.

[10] T. Beattie. Investigation of the SYCL for OpenCL Programming Model. Master thesis, University of Edinburgh, 2015.

[11] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59: 65-98, 2017.

[12] P. Birken, G. Gassner, M. Haas, and C.-D. Munz. Preconditioning for modal discontinuous Galerkin methods for unsteady 3D Navier – Stokes equations. *Journal of Computational Physics*, 240:20–35, 2013.

[13] J. Bonet and R. D. Wood. *Nonlinear continuum mechanics for finite element analysis.* Cambridge Univeristy Press, second edition, 2008.

[14] C. D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J. E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R. M. Kirby, and S. J. Sherwin. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.

[15] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers and Fluids*, 43(1):23–28, 2011.

[16] C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clMAGMA: High Performance Dense Linear Algebra with OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, 2014.

[17] H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, pages 44–54. IEEE, 2009.

[19] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite Categories and Subject Descriptors. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, 2010.

[20] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel numerical linear algebra. *Acta Numerica*, 2:111–197, 1993.

[21] J. Diaz, A. Nin, and C. Mun. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[22] T. Dong, A. Haidar, P. Luszczek, S. Tomov, A. Abdelfattah, and J. Dongarra. MAGMA Batched: A Batched BLAS Approach for Small Matrix Factorizations and Applications on GPUs. ICL Tech Report 08/2016, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, 37996, 2016.

[23] J. J. Dongarra, P. Luszczek, and A. Petite. The LINPACK benchmark: Past, present and future. *Concurrency Computation Practice and Experience*, 15(9):803–820, 2003.

[24] M. Dubiner. Spectral methods on triangles and other domains. *Journal of Scientific Computing*, 6(4):345–390, 1991.

[25] M. G. Duffy. Quadrature over a pyramid or cube of integrands with a singularity at a vertex. *SIAM Journal on Numerical Analysis*, 19(6):1260–1262, 1982.

[26] J. Eichstädt, M. Green, M. Turner, J. Peiró, and D. Moxey. Accelerating high-order mesh optimisation with an architecture-independent programming model. *Computer Physics Communications*, 229:36–53, 2018.

[27] J. Eichstädt, D. Moxey, and J. Peiró. Towards a performance-portable high-order implicit flow solver. In *AIAA Science and Technology Forum 2019*, January 2019.

[28] J. Eichstädt, D. Moxey, and J. Peiró. Efficient vectorised CUDA kernels for high-order finite element flow solvers. In *UKACM Conference 2020*. figshare, April 2020.

[29] J. Eichstädt, M. Vymazal, D. Moxey, and J. Peiró. A comparison of the shared-memory parallel programming models OpenMP, OpenACC and Kokkos in the context of implicit solvers for high-order FEM. *Computer Physics Communications*, page 107245, 2020.

[30] J. Enmyren and C. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14, 2010.

[31] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Proceedings of the International Conference on Parallel Processing*, pages 216–225, 2011.

[32] N. Fehn, W. A. Wall, and M. Kronbichler. Efficiency of high-performance discontinuous Galerkin spectral element methods for under-resolved turbulent incompressible flows. *International Journal for Numerical Methods in Fluids*, 2018.

[33] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[34] M. Fortunato and P. O. Persson. High-order unstructured curved mesh generation using the Winslow equations. *Journal of Computational Physics*, 307:1–14, 2016.

[35] V. Garanzha and I. Kaporin. Regularization of the variational method of grid generation. *Computational Mathematics and Mathematical Physics*, 39:1428–1440, 1999.

[36] V. A. Garanzha. Variational principles in grid generation and geometric modelling: Theoretical justifications and open problems. *Numerical Linear Algebra with Applications*, 11(5-6):535–563, 2004.

[37] A. Gargallo-Peiró, X. Roca, and J. Sarrate. A surface mesh smoothing and untangling method independent of the CAD parameterization. *Computational Mechanics*, 53(4):587–609, 2014.

[38] C. Geuzaine. Gmsh : a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[39] clBLAS Repository. `https://github.com/clMathLibraries/clBLAS`, last visited on 03/06/2020, 2017.

[40] C++ Heterogeneous-Compute Interface for Portability (HIP) Repository. `https://github.com/ROCm-Developer-Tools/HIP`, last visited on 03/06/2020, 2020.

[41] clang-ykt Repository. `https://github.com/clang-ykt`, last visited on 03/06/2020, 2020.

[42] Rose-compiler Repository. `https://github.com/rose-compiler/rose`, last visited on 03/06/2020, 2020.

[43] P. N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Technical report, NVIDIA, 2009.

[44] D. Grewe, Z. Wang, and M. F. P. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[45] J. Guermond and J. Shen. Velocity-correction projection methods for incompressible flows. *SIAM Journal of Numerical Analysis*, 41:112–134, 2003.

[46] N. A. Gumerov and R. Duraiswami. *Fast multipole methods for the Helmholtz equation in three dimensions.* Elsevier, 2005.

[47] G. A. Hansen, P. G. Xavier, S. P. Mish, T. E. Voth, M. W. Heinstein, and M. W. Glass. An MPI+X implementation of contact global search using Kokkos. *Engineering with Computers*, 32(2):295–311, 2016.

[48] A. Hart, R. Ansaloni, and A. Gray. Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. *European Physical Journal: Special Topics*, 210(1):5–16, 2012.

[49] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, pages 465–471, 2012.

[50] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Labs, 2009.

[51] M. A. Heroux, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanley, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, and R. P. Pawlowski. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

[52] J. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods.* Springer, 2008.

[53] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur. MPI + MPI: A new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121–1136, 2013.

[54] R. Hornung, H. Jones, J. Keasler, R. Neely, A. Kunen, and O. Pearce. RAJA Overview. Technical Report LLNL-TR-677453, Lawrence Livermore National Laboratory, 2015.

[55] T. Hoshino, N. Maruyama, and S. Matsuoka. An OpenACC extension for data layout transformation. In *Proceedings of WACCPD 2014: 1st Workshop on Accelerator Programming Using Directives*, pages 12–18, 2014.

[56] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki. CUDA vs OpenACC: Performance case studies with Kernel benchmarks and a memory-bound CFD application. In *Proceedings - 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013*, pages 136–143, 2013.

[57] D. Ibanez and M. Shephard. Mesh adaptation for moving objects on shared memory hardware. In *25th International Meshing Roundtable (IMR25)*, 2016.

[58] Intel. Using AVX Without Writing AVX Code. `https://software.intel.com/en-us/articles/\using-avx-without-writing-avx-code`, last visited on 03/06/2020, 2012.

[59] Intel oneAPI Toolkits (Beta). `https://software.intel.com/en-us/oneapi`, last visited on 03/06/2020, 2020.

[60] C. T. Jacobs, S. P. Jammy, and N. D. Sandham. OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Journal of Computational Science*, 18:12–23, 2017.

[61] Z. Jia, M. Maggioni, B. Staiger, and D. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. `https://arxiv.org/pdf/1804.06826.pdf`, last visited on 06/09/2020, 2018.

[62] A. Karakus, N. Chalmers, K. Świrydowicz, and T. Warburton. A GPU accelerated discontinuous Galerkin incompressible flow solver. *Journal of Computational Physics*, 390:380 – 404, 2019.

[63] G. Karniadakis and S. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press, second edition, 2005.

[64] G. E. Karniadakis, M. Israeli, and S. A. Orszag. High-Order Splitting Methods for the Incompressible Navier-Stokes Equations. *Journal of Computational Physics*, 97(2):414–443, 1991.

[65] P. Kegel, M. Steuwer, and S. Gorlatch. DOpenCL: Towards uniform programming of distributed heterogeneous multi-/many-core systems. *Journal of Parallel and Distributed Computing*, 73(12):1639–1648, 2013.

[66] C. Kessler, U. Dastgeer, M. Majeed, N. Furmento, S. Thibault, R. Namyst, S. Benkner, S. Pllana, J. L. Traff, and M. Wimmer. Leveraging PEPPHER technology for performance portable supercomputing. In *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1395–1397, 2012.

[67] The OpenCL Specification. `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`, last visited on 03/06/2020, 2018.

[68] The SPIR Specification. https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf, last visited on 03/06/2020, 2018.

[69] SYCL Specification. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, last visited on 03/06/2020, 2020.

[70] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee. Bridging OpenCL and CUDA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[71] B. Krank, N. Fehn, W. A. Wall, and M. Kronbichler. A high-order semi-explicit discontinuous Galerkin solver for 3D incompressible flow with application to DNS and LES of turbulent channel flow. *Journal of Computational Physics*, 348:634–659, 2017.

[72] M. Kronbichler and K. Kormann. Fast Matrix-Free Evaluation of Discontinuous Galerkin Finite Element Operators. *ACM Transactions on Mathematical Software*, 45(3):1–40, 2019.

[73] J. H. Lee, N. Nigania, H. Kim, K. Patel, and H. Kim. OpenCL performance evaluation on modern multicore CPUs. *Scientific Programming*, 2015.

[74] S. Lee and J. S. Vetter. Early evaluation of directive-based GPU programming models for productive exascale computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[75] H. Luo, G. Chen, P. Li, C. Ding, and X. Shen. Data-centric Combinatorial Optimization of Parallel Code. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

[76] M. Mantor and M. Houston. AMD Graphics Core Next (GCN) Architecture. Technical report, AMD Inc., 2011.

[77] J. Marcon, M. Turner, D. Moxey, S. Sherwin, and J. Peiró. A variational approach to high-order r-adaptation. In *26th International Meshing Roundtable*, September 2017.

[78] S. McIntosh-Smith, T. Wilson, A. Á. Ibarra, J. Crisp, and R. B. Sessions. Benchmarking energy efficiency, power costs and carbon emissions on heterogeneous systems. *Computer Journal*, 55(2):192–205, 2012.

[79] D. S. Medina, A. St-Cyr, and T. Warburton. OCCA: A unified approach to multithreading languages. *arXiv preprint arXiv:1403.0968*, pages 1–25, 2014.

[80] Microsoft. C++ AMP (Accelerated Massive Parallelism). https://docs.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-cpp-accelerated-massive-parallelism, last visited on 03/06/2020, 2016.

[81] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling. In *Proceedings of the 16th international conference on Supercomputing - ICS '02*, page 35, 2002.

[82] D. Moxey, R. Amici, and M. Kirby. Efficient matrix-free high-order finite element evaluation for simplicial elements. *SIAM Journal on Scientific Computing*, 42(3):C97–C123, 2020.

[83] D. Moxey, C. D. Cantwell, Y. Bao, A. Cassinelli, G. Castiglioni, S. Chun, E. Juda, E. Kazemi, K. Lackhove, J. Marcon, G. Mengaldo, D. Serson, M. Turner, H. Xu, J. Peiró, R. M. Kirby, and S. J. Sherwin. Nektar++: Enhancing the capability and application of high-fidelity spectral/hp element methods. *Computer Physics Communications*, 249:107110, 2020.

[84] D. Moxey, C. D. Cantwell, R. M. Kirby, and S. J. Sherwin. Optimising the performance of the spectral/hp element method with collective linear algebra operations. *Computer Methods in Applied Mechanics and Engineering*, 310:628–645, 2016.

[85] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. `http://www.nvidia.co.uk/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, last visited on 01/05/2017, 2012.

[86] NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, last visited on 03/06/2020, 2016.

[87] NVIDIA Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, last visited on 03/06/2020, 2017.

[88] CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/index.html`, last visited on 03/06/2020, 2019.

[89] Thrust Quick Start Guide. `http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf`, last visited on 03/06/2020, 2019.

[90] The OpenACC Application Programming Interface. `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf`, last visited on 03/06/2020, 2017.

[91] OpenMP 3.1 Specifications. `https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf`, last visited on 03/06/2020, 2011.

[92] OpenMP 4.5 Specifications. `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`, last visited on 03/06/2020, 2015.

[93] S. A. Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37(1):70–92, 1980.

[94] A. T. Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468 – 488, 1984.

[95] P.-O. Persson and J. Peraire. Curved Mesh Generation and Mesh Refinement using Lagrangian Solid Mechanics. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting and Exhibit*, pages 1–11, 2009.

[96] B. P. Pickering, C. W. Jackson, T. R. W. Scogland, W. C. Feng, and C. J. Roy. Directive-based GPU programming for computational fluid dynamics. *Computers and Fluids*, 114:242–253, 2015.

[97] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[98] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):24:1 – 24:27, 2016.

[99] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach. Performance Portable GPU Code Generation for Matrix Multiplication. In *GPGPU: Workshop on General Purpose Processor Using Graphics Processing Units*, pages 22–31, 2016.

[100] K. Rupp. Microprocessor trend data. `https://github.com/karlrupp/microprocessor-trend-data`, last visited on 03/06/2020, 2018.

[101] W. Schiesser. *The Numerical Method of Lines - Integration of Partial Differential Equations*. Academic Press, Inc., 1991.

[102] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance traps in OpenCL for CPUs. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 38–45, 2013.

[103] S. J. Sherwin and J. Peiró. Mesh generation in curvilinear domains using high-order elements. *International Journal for Numerical Methods in Engineering*, 53(1):207–223, 2002.

[104] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In *Proceedings - 23rd IEEE International Conference on High Performance Computing, HiPC 2016*, pages 193–202, 2017.

[105] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis. CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences. Technical Report Nasa Cr-2014-21878, NASA, 2014.

[106] ISO CPP P0024R2. `https://isocpp.org/files/papers/P0024R2.html`, last visited on 03/06/2020, 2016.

[107] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan. Early Experiences Co-Scheduling Work and Communication Tasks for Hybrid MPI+X Applications. In *Proceedings of ExaMPI 2014*, pages 9–19, 2015.

[108] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, University of Illinois at Urbana-Champaign, 2012.

[109] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. Top500 List. `https://www.top500.org/`, last visited on 03/06/2020, 2020.

[110] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins. An Overview of Performance Portability in the Uintah Runtime System Through the Use of Kokkos. In *Second International Workshop on Extreme Scale Programming Models and Middleware*, pages 1–4, 2016.

[111] POSIX Threads. `http://pubs.opengroup.org/onlinepubs/007908799/xsh/threads.html`, last visited on 03/06/2020, 1997.

[112] K. Thouti. Comparison of OpenMP & OpenCL Parallel Processing Technologies. *International Journal of Advanced Computer Science and Applications*, 3(4):56–61, 2012.

[113] A. Trigkas. Investigation of the OpenCL SYCL Programming Model. Master thesis, University of Edinburgh, 2014.

[114] M. Turner, D. Moxey, S. J. Sherwin, and J. Peiró. Automatic Generation of 3D Unstructured High-Order Curvilinear Meshes. In *ECCOMAS Congress 2016 VII European Congress on Computational Methods in Applied Sciences and Engineering*, pages 5–10, 2016.

[115] M. Turner, J. Peiró, and D. Moxey. A Variational Framework for High-Order Mesh Generation. *Procedia Engineering*, 163:340–352, 2016.

[116] M. Turner, J. Peiró, and D. Moxey. Curvilinear mesh generation using a variational framework. *Computer-Aided Design*, 2017.

[117] MPI 3.1 Specifications. `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, last visted on 03/06/2020, 2015.

[118] Exascale Computing Project. `https://exascaleproject.org/`, last visited on 06/09/2020, 2018.

[119] B. C. Vermeire, F. D. Witherden, and P. E. Vincent. On the utility of GPU accelerated high-order methods for unsteady flow simulations: A comparison with industry-standard tools. *Journal of Computational Physics*, 334:497–521, 2017.

[120] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for Programming with Millions of Lightweight Threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[121] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[122] F. D. Witherden, A. M. Farrington, and P. E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028–3040, 2014.

[123] F. D. Witherden and P. E. Vincent. On the identification of symmetric quadrature rules for finite element methods. *Computers and Mathematics with Applications*, 69(10):1232–1241, 2015.

[124] Z. Q. Xie, R. Sevilla, O. Hassan, and K. Morgan. The generation of arbitrary order curved meshes for 3D finite element analysis. *Computational Mechanics*, 51(3):361–374, 2013.

[125] R. Xu, S. Chandrasekaran, and B. Chapman. Exploring Programming Multi-GPUs Using OpenMP and OpenACC-Based Hybrid Model. In *2013 IEEE International Symposium on Parallel & Distributed Processing*, pages 1169–1176, 2013.

[126] Y. Yan, P.-H. Lin, C. Liao, B. R. de Supinski, and D. J. Quinlan. Supporting multiple accelerators in high-level programming models. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '15*, pages 170–180, 2015.

[127] K. Świrydowicz, N. Chalmers, A. Karakus, and T. Warburton. Acceleration of tensor-product operations for high-order finite element methods. *The International Journal of High Performance Computing Applications*, 33(4):735–757, 2019.