Mathematik

Generating Block-Structured Kernels for Low Order Finite Element Methods

A High-Performance Oriented View

Inaugural Dissertation
zur Erlangung des Doktorgrades der Naturwissenschaften

– Dr. rer. nat. –

im Fachbereich Mathematik und Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät
der Westfälischen Wilhelms-Universität Münster

eingereicht von
Marcel Dominique Koch
aus
Gelsenkirchen

– 2021 –

Dekan:	Prof. Dr. Xiaoyi Jiang Westfälische Wilhelms-Universität Münster Münster, DE
Erster Gutachter:	Prof. Dr. Christian Engwer Westfälische Wilhelms-Universität Münster Münster, DE
Zweiter Gutachter:	Prof. Dr. Haral Köstler Friedrich Alexander Universität Erlangen-Nürnberg Erlangen, DE
Tag der mündlichen Prüfung:	14.04.2021
Tag der Promotion:	14.04.2021

A version optimized for digital reading can be found under
DOI: 10.5281/zenodo.4724412

Zusammenfassung

Das Ziel dieser Arbeit ist die Steigerung der Performanz für die Berechnung von FEM mit niedriger Ordnung. Ein wesentliches Problem von FEM Anwendungen ist der Memory Gap. Zu den vielversprechendsten Lösungsansätzen zählen die matrixfreien Berechnungen, bei denen die von den linearen Gleichungssystemlösern benötigte Operatoranwendung direkt berechnet wird, ohne eine Systemmatrix zu assemblieren. Für Diskretisierungen mit höherer Ordnung funktioniert dieser Ansatz, aber die naiven Umsetzungen für niedrige Ordnungen schlägt fehl. Um auch für niedrige Ordnung die matrixfreie Operatoranwendung zu beschleunigen, betrachtet diese Arbeit den Ansatz der blockstrukturierten Gitter. Das Hauptkonzept ist ein zweistufigen Gitters, bestehend aus groben Makro-Elementen, welche bei ihrer Abarbeitung uniform verfeinert werden. Damit werden neue Optimierungen möglich. Zum einen kann die globale Assemblierung optimiert werden, und zum anderen profitieren die Makro-Element-Kernel durch eine Reduzierung der FLOP sowie durch vektorisierte Operationen, neben weiteren Optimierungen. Um einen Vorkonditionierer bereitzustellen, der mit dem matrixfreien Ansatz kompatibel ist, wurde ein einfaches, nicht überlappendes Gebietszerlegungsverfahren auf blockstrukturierte Gitter angepasst. Die dargestellten Ansätze sind als Teil des Code-Generierung-Frameworks DUNE-CODEGEN implementiert, um die Nutzbarkeit der Optimierungen zu vereinfachen. Durch die Generierung der notwendigen Kernels kann dieselbe Performanz wie für eine handgeschriebene Implementierung erreicht werden.

Abstract

The aim of this thesis is to increase the performance of low order FEM computations. A major issue for FEM applications is the memory gap. One of the most promising approaches to overcome this problem are matrix-free computations, where the operator application required by most linear system solvers is computed directly without assembling a system matrix. For high order discretizations this approach works well, but naive realizations fail for low order discretizations. To accelerate the matrix-free operator application even for low order methods, the block-structured grids approach is considered here. The core concept is to use a two level grid, composed of coarse macro elements which are uniformly refined during their handling. This opens up new optimization possibilities. On the one hand, the global assembly can be optimized, and, on the other hand, the macro element kernels benefit from a reduced FLOP count and vectorized computations, among other optimizations. To provide a preconditioner compatible with the presented matrix-free approach, a simple non-overlapping domain decomposition method is adapted to block-structured grids. The presented approaches are implemented as part of the code generation framework DUNE-CODEGEN to ease the usage of the optimizations. By generating the necessary kernels, the same performance as for hand-written implementations can be reached.

Acknowledgements

I am deeply grateful to my supervisor Prof. Dr. Christian Engwer for his assistance at every stage of my thesis. He first introduced me to this interesting topic and continued to offer helpful advice if needed. I acknowledge the funding from German Federal Ministry of Education and Research (BMBF) under the grant 01IH16003A as part of the HPC^{2}SE project. Additionally, I would like to extend my sincere thanks to all members of the project. Especially Dr. Domimic Kempf and René Heß offered valuable help with all things concerning DUNE-CODEGEN. For his major contributions towards the discussion in the outlook of chapter 5.1 I would like to thank Dr. Lawrence Mitchell. Furthermore, I would like to express my deepest gratitude to all current and former colleges of the research group 'Applications of PDEs' for the pleasant working climate. In particular my gratitude goes to Nils Dreier and Michael Wenske, who proof-read this thesis and offered insightful comments and suggestions. An meine Eltern, Danke, dass ich mich immer auf eure Unterstützung und Rückhalt verlassen konnte.

Contents

List of Tables

List of Figures

List of Algorithms

CHAPTER 1
Introduction & Outline

THE COMPUTER HARDWARE LANDSCAPE is ever-changing. In recent decades, the main driver for performance advancements has been parallelism. Not only did the number of computing cores within one CPU increase, from single-core to multi-core CPUs even for average consumers, the addition of so-called instruction level parallelism introduced parallelism at the level of each core. With these advancements, the performance of one CPU could grow at a steady rate. An additional factor in the performance of any program is the data transfer rate (called bandwidth) between the main memory and the CPU. While this also advanced over the last decades, it did so at a significantly lower rate than the CPU's performance. Nowadays this leads to a disadvantage for programs that require transferring large volumes of data. Since their performance is restricted by the bandwidth, many resources of modern CPUs have to be left unused.

One application area that has been particullarily impacted are computer simulations. They dominate the physics based computations, which appear for example in weather predictions, structural analysis, measuring brain activity, prevention of earthquake induced hazards, or cancer research. All of these applications have in common that they are described by one or multiple mathematical equations. A particular successful method to solving these so-called partial differential equations (PDEs) is the finite element method (FEM). The approach translates the complex equations into an easier to solve system of linear equations, or at least into multiple steps consisting of linear systems. For conventional implementations of the FEM, computing the solution of the linear systems usually requires transferring large volumes of data and significantly less arithmetic operations upon these data in comparison. Therefore, the FEM is deeply affected by the gap between bandwidth and CPU performance mentioned above.

A promising technique to lessen this impact are so-called matrix-free methods, as opposed to the conventional matrix-based approaches. The matrix-free approach reduces the transferred data volume at the cost of noticeably increasing the number of required computations. As modern CPUs excel at multiple computations with the same data, this should reduce the overall runtime of the FEM. For certain classes of FEM, so-called high order discretizations, this works especially well. Other classes, so-called low order discretizations, show nearly the same problems using the matrix-free method as with the matrix-based method. In these cases, the number of arithmetic operations per transferred datum is still low and more adjustments are necessary to increase the performance of matrix-free methods.

This thesis considers the so-called block-structured grids approach to accelerate matrix-free methods for low order discretizations. The approach regularizes the data access and computation patterns during the matrix-free computation. Besides higher utilization of the bandwidth, new optimization opportunities emerge. As writing code that leverages these optimizations is cumbersome, the source code for the main building blocks of the FEM, the local integration kernels, are automatically generated, following a descriptive problem formulation in a simple programming language. By generating optimized code for block-structured grids, users can easily increase the performance of matrix-free methods with low order discretizations for a variety of different problems.

Related Work

Currently, matrix-free methods are a highly active field of research. In recent years the development of efficient methods using high order discretizations has been in the focus. Especially prominent are the discussions from the deal.II community [63, 64], the DUNE project [76], and as part of the CEED center [18] and the MFEM library [5]. All of these underline the high performance capabilities of the matrix-free approach for high order discretizations. An interesting performance comparison of current matrix-free methods can be found in [39]. While offering no comparison with matrix-based methods, this article also considers some lower order finite elements, although the focus lies clearly on the discussion of higher order discretizations.

Block-structured grids have been explored by several frameworks. One of the earliest applications can be found in the finite element package FEAST [90, 91], which uses a coarse, unstructured mesh of cubical elements and refines each coarse element with a tensor product structure. Exploiting the special structure allows assembling highly efficient sparse matrices. Similarly, the linear algebra library HYPRE [35] provides a specialized matrix format for block-structured grids. HYPRE also expects that the coarse grid is divided into cubical elements. However, in contrast to FEAST, the sparsity pattern of structured block can be described by a stencil. One framework that uses locally structured grids without explicitly forming matrices is the hierarchical hybrid girds approach [13, 61]. There, the regular structure is exploited to efficiently apply the matrix's stencil on-the-fly. Currently, the implementation is restricted to coarse grids with simplex elements. As part of the EXA-DUNE project [10], block-structured grids were investigated to increase the vectorization capabilities for low order methods, again on cubical coarse grids. Although the approach developed in this thesis also builds upon DUNE, the data layout and vectorization are significantly different from the earlier consideration.

Most of the mentioned frameworks based on block-structuring employ a variant of geometric multilevel method either as a solver or as a preconditioner. Non-overlapping domain decomposition methods, similar to the one considered in this thesis, are also highly efficient preconditioners. However, the current implementations of these methods (e.g. [7, 96]) are matrix-based. For modern non-overlapping domain decomposition methods matrix-free implementations are still missing.

One of the most prominent example of code generation within the FEM context is the FEniCS project [2, 69]. It defines the Unified Form Language (UFL), which allows the Python based description of PDEs. Its widespread success is mostly due to its ease-of-use as part of the Python ecosystem and the close resemblance of UFL to the mathematical formulation of a FEM. Other projects, such as Firedrake [80] also build upon UFL. The Firedrake implementation usually generates C code for the whole FEM program, while the FEniCS mostly generates kernels that are plugged into the C++ framework DOLPHIN [70]. Additionally, the Firedrake project employs a highly capable FLOP minimization algorithms [80] during its local kernel optimization.

Outline

The major contributions of this work are the evaluation and implementation of optimization techniques for low order discretizations. In the following, the structure of the thesis is laid out. The thesis begins with a review of the basic concepts and current issues in chapter 2, which the following chapters are build upon. Afterwards chapter 3 introduces the block-structured grids approach. This is the thesis' main technique to accelerate matrix-free computations for low order finite element methods. The chapter also contains a discussion of optimizations unique to the block-structured grids approach, which can be divided into local kernel and global assembly optimizations. Additionally, the performance gain of each optimization is closely investigated, using a handwritten implementation. In the following, a well known non-overlapping domain decomposition preconditioner is applied to block-structured grids in chapter 4. The outlined approach indicates how this kind of preconditioner can be adapted to matrix-free methods. Chapter 5 discusses the implementation of the results so far as part of the code generation pipeline DUNE-CODEGEN.

Thus, the benefits from block-structured grids can be easily applied to a variety of problems. Finally, for small subset of those problems the performance of the generated block-structured grids approach is evaluated on a modern CPU in chapter 6. Both a single operator application and a full system solve are considered, and set in comparison to a standard matrix-based method. Additionally, a theoretical analysis of the block-structured matrix-free operator application is provided.

CHAPTER 2
Foundational Considerations

THIS CHAPTER INTRODUCES the concepts and terminology necessary for the rest of this work. First, a brief overview of the finite element method is presented, with focus on implementation relevant parts. One of the first and most defining textbooks on the finite element methods is by Ciarlet [19], and many more have been released since. For example, the book by Brenner and Scott [17] delivers a thorough analytic discussion of the method and a compact introduction can be found in Braess' work [16]. Practical concerns, which are mostly left out in other works, are considered by Ern and Guermond in [33]. Secondly, basic properties of common linear solvers are discussed. Standard introductions on solution methods for linear systems can be found in the books by Saad [81] or by Rannacher [79], and an interesting discussion of the conjugate gradient method is given from Shewchuk in [86]. Afterwards, the software tools, DUNE and UFL, which are the fundations for the implementation of this thesis, are introduced. The final section, considers the performance of the conventional finite element approach, which relies heavily on sparse matrix vector multiplications.

2.1 Finite Element Methods

The finite element method (FEM) allows to approximately solve a wide variety of partial differential equations (PDEs), where it is not possible to analytically compute the solution in general. To simplify the introduction of basic concepts for the FEM only scalar linear second order elliptic PDEs are considered. These take the following form

$$
\begin{aligned}
-\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}\left(d_{ij}(x)\frac{\partial u(x)}{\partial x_j}\right) + \sum_{j=1}^{d} b_j(x)\frac{\partial u(x)}{\partial x_j} + c(x)u(x) = f(x) \text{ in } \Omega, \\
+\text{boundary conditions on } \partial\Omega,
\end{aligned}
$$

which may be compactly written as

$$
\begin{aligned}
\mathcal{L}u &= f \text{ in } \Omega, \\
Bu &= g \text{ on } \partial\Omega,
\end{aligned}
$$

where $\Omega \subset \mathbb{R}^d$ is a bounded, open, and polygonal domain, and the coefficients $D = [d_{ij}]_{i,j=1}^d : \Omega \mapsto \mathbb{R}^{d\times d}$, $b = [b_i]_{j=1}^d : \Omega \mapsto \mathbb{R}^d$, $c : \Omega \mapsto \mathbb{R}$, and the right-hand-side $f : \Omega \mapsto \mathbb{R}$ fulfil some suitable assumptions, which are described later on. The boundary conditions $Bu = g$ stemming from a physical problem can be divided into three distinct types

1. Dirichlet condition: $u(x) = g_D(x)$ on $\Gamma = \partial\Omega$,

2. Neumann condition: $n(x) \cdot D(x)\nabla u(x) = g_N(x)$ on Γ with the outer normal n,

3. Robin condition: $u(x) + n(x) \cdot D(x)\nabla u(x) = g_R(x)$ on Γ.

It is also possible to mix these boundary conditions. In this case, the boundary is divided into $\overline{\Gamma}_D \cup \overline{\Gamma}_N \cup \overline{\Gamma}_R = \Gamma$ with $\Gamma_i \cap \Gamma_j = \emptyset$ for $i \neq j$, $i, j \in \{D, N, R\}$, where the Dirichlet boundary condition is defined on $\Gamma_D \subset \Gamma$, the Neumann boundary condition on $\Gamma_N \subset \Gamma$, and the Robin condition on $\Gamma_R \subset \Gamma$.

2.1.1 Weak Formulation

The previous description of a PDE, called strong formulation, has the drawback that the existence and uniqueness of a solution can only be shown under idealized assumptions, which usually don't apply in real world applications. Therefore, the weak formulation was developed to create a well-posed problem, which is equivalent to the strong formulation if the idealized assumptions hold. This formulation can be deduced by multiplying the strong formulation with a test function $v \in C^\infty(\overline{\Omega})$ with $v|_{\Gamma_D} = 0$, integrating over Ω, and applying Green's formula, which results in (without Robin condition)

$$\underbrace{\int_\Omega \nabla v \cdot D\nabla u + v(b \cdot \nabla u + cu)\,\mathrm{dx}}_{:=a(u,v)} = \underbrace{\int_\Omega f v\,\mathrm{dx} + \int_{\Gamma_N} v g_N\,\mathrm{dS}}_{:=F(v)}.$$

This equation requires less regularity of u than the PDE in its strong formulation. Now, it is sufficient that the first order weak derivative of u exists and that it is L^2 integrable. The Sobolev space $H^1(\Omega)$ contains all these functions and the space $H^1_{\Gamma_D,0}(\Omega)$ additionally incorporates the homogeneous Dirichlet condition on Γ_D.

Hence, the problem in the weak formulation for homogeneous Dirichlet conditions is defined as:

$$\begin{aligned}&\text{Find } u \in H^1_{\Gamma_D,0}(\Omega) \text{ such that:}\\ &a(u,v) = F(v) \quad \forall v \in H^1_{\Gamma_D,0}(\Omega).\end{aligned}$$

A solution to this problem is called weak solution. For the analysis of weak formulations it is sufficient to consider homogeneous Dirichlet conditions, since a problem with a Dirichlet condition $g_D \neq 0$ can be reduced to problem with homogeneous conditions. Let $w \in H^1$ with $w|_{\Gamma_D} = g_D$, then finding $u = u_0 + w$, with $u_0 \in H^1_{\Gamma_D,0}$, to solve the weak formulation is equivalent to solve

$$a(u_0, v) = F(v) - a(w, v) \quad \forall v \in H^1_{\Gamma_D,0},$$

which has only homogeneous Dirichlet conditions. Thus, in the following, the Dirichlet conditions are considered to be homogeneous, unless stated otherwise.

The existence and uniqueness of a solution to the weak formulation requires some assumptions on the coefficients and the boundary conditions. The right-hand-side f and the boundary functions g_N, g_R need to be L^2 integrable, while the coefficients should be bounded almost everywhere, i.e. $D \in [L^\infty(\Omega)]^{d\times d}$, $b \in [L^\infty(\Omega)]^d$, and $c \in L^\infty$. These assumptions lead to the boundedness of both the linear form $F(\cdot)$ and the bilinear form $a(\cdot,\cdot)$. For the well-posedness of the weak formulation it is necessary that $a(\cdot,\cdot)$ is also coercive on H^1. This can be achieved through a multitude of different assumptions. One necessary, but not sufficient, assumption is that the matrix function D is symmetric and uniformly elliptic, i.e. $d_{ij} = d_{ji}$ and $\xi \cdot D\xi \geq \alpha||\xi||_2^2 > 0$ for all $\xi \in \mathbb{R}^d$ a.e. in Ω. In the case of pure Dirichlet boundary conditions, coercivity is guaranteed if $\alpha + \min(0, p/C_\Omega) > 0$, where C_Ω is a domain dependent constant appearing in the Poincaré inequality and $p = \inf\operatorname{ess}_{x\in\Omega}(c - \frac{1}{2}\operatorname{div} b)$. Further boundary conditions are considered, for example, in [33].

2.1.2 Discretization

In contrast to the strong formulation, the weak formulation is well-posed, but computing the solution analytically is not easier than before. This difficulty leads to the Galerkin method, which tries to solve the weak problem on a finite dimensional subspace $V_h \subset H^1_{\Gamma_D,0}$, resulting in the problem:

$$\begin{gathered}\text{Find } u_h \in V_h \text{ such that:}\\ a(u,v) = F(v) \quad \forall v \in V_h.\end{gathered}$$

The discrete solution u_h is quasi-optimal in the sense that it attains the minimal error to the weak solution over V_h, up to a constant, which is known as Céa's lemma

$$||u - u_h||_{H^1} \leq C \inf_{v_h \in V_h} ||u - v_h||_{H^1},$$

and shown in [17] for example. Choosing a basis $\{\phi_i\}_{i=1}^N$ for the subspace V_h reduces the discrete weak problem to N linear equations

$$a(u_h, \phi_i) = F(\phi_i), \quad i = 1, \dots, N,$$

and writing u_h as a linear combination of $\{\phi_i\}$ results in a linear system

$$\begin{aligned} a\left(\sum_{j=1}^N \mathbf{u}_j \phi_j, \phi_i\right) &= F(\phi_i), \quad i = 1, \dots, N,\\ \Leftrightarrow \sum_{j=1}^N a(\phi_j, \phi_i)\mathbf{u}_j &= F(\phi_i), \quad i = 1, \dots, N,\\ \Leftrightarrow A\mathbf{u} &= \mathbf{f}.\end{aligned}$$

The matrix $A = [a_{i,j}]_{i,j=1}^N = [a(\phi_j, \phi_i)]_{i,j=1}^N$ is known as the stiffness matrix, and the right-hand-side $\mathbf{f} = [F(\phi_i)]_{j=1}^N$ as the load vector. The entries in the coefficient vector $\mathbf{u} = [\mathbf{u}_i]_{i=1}^N$ are also referred to as degrees of freedom (DoFs).

The choice of the subspace V_h is critical for the efficient and accurate approximation of the weak solution. It should be large enough to minimize the error determined by Cea's lemma, which results in a large linear system to be solved. Therefore, a requirement on the basis $\{\phi_i\}$ of V_h is that the solution to linear system $A\mathbf{u} = \mathbf{f}$ can be computed in reasonable time, preferably in $\mathcal{O}(N)$, which prohibits a dense structure of the matrix A. Instead, the matrix A should be sparse, meaning that only $\mathcal{O}(N)$ entries are non-zero ($a_{i,j} = a(\phi_j, \phi_i) \neq 0$). Such a pattern emerges, if the support of each basis functions is highly restricted.

The first step towards the definition of a suitable basis is to divide the domain Ω into simpler subdomains. The subdivision of Ω is described by a grid $\mathcal{T} = \{\tau_1, \tau_2, \dots, \tau_M\}$, also called mesh, or triangulation, which consists of bounded domains τ_1 with Lipschitz boundary and

$$\Omega = \bigcup_{i=1}^{M} \overline{\tau}_i, \quad \tau_i \cap \tau_j = \emptyset \text{ if } i \neq j.$$

Usually the elements τ_i are triangles or quadrilaterals in 2D, and tetrahedrons, hexahedrals, prisms, or pyramids in 3D. A grid where $\operatorname{diam}(\tau_i) \leq h$ for all $i = 1, \dots, M$ is denoted by $\mathcal{T}_h$, and h is known as the mesh size. In some cases additional requirements may be applied to the triangulation to improve the error estimates, which can be found, for example, in [16].

It will become necessary to distinguish between parts of an element with different dimensionality, for example between the interior volume of a hexahedron and its faces. These parts are called entities, and they are defined recursively by their topological connections to lower dimensional entities. For example, a quadrilateral (dim $=$ 2) is constructed by four edges (dim $=$ 1), which are in turn constructed by two vertices (dim $=$ 0) each. In addition to the dimension of an entity, the codimension of an entity is defined as

$$\operatorname{codim}(e) = d - \dim(e).$$

All entities of codim $> c$ that are part of an entity e with codim $= c$ are referred to as the subentities of e. Without reference to a particular entity, the term subentities also refers to all entity of a grid element, the codim $= 0$ entity included. A more rigorous definition of entities and subentities, especially as part of the DUNE terminology, can be found in [8].

Furthermore, each element τ_i of the grid is defined by a geometry transformation T_i. The transformation T_i maps $\hat{\tau} \mapsto \tau_i$, where $\hat{\tau}$ is called reference element, and it is bijective and T_i, T_i^{-1} are continuous. If the elements are simplices, e.g. lines, triangles, or tetrahedron, the reference element is defined as $\hat{\tau} = \hat{S} = \{x \in \mathbb{R}_+^d : ||x||_1 \leq 1\}$, and for cubical elements, e.g. quadrilaterals, or hexahedrals, it is defined as $\hat{\tau} = \widehat{Q} = \{x \in R_+^d : ||x||_\infty \leq 1\}$. In some cases, the mapping is affine and thus can be written as $\mathcal{T}_i(x) = J_i x + c_i$ with $J_i \in \mathbb{R}^{d \times d}$ and $c_i \in R^d$. If every mapping is affine, the grid is also called *affine*. Furthermore, if the mappings are affine and only differ in the offset c_i, i.e. $J_i = J_k = J$ for all $i, k = 1, \dots, M$, the grid is called *structured*, or uniform, which becomes an important special case in the following chapters, and otherwise it will be denoted as *unstructured*. It is possible to have mappings for multiple reference elements, for example if the grid contains both triangles and quadrilaterals. However, it is assumed that the reference element is always the reference cube

$\widehat{Q}$ for the rest of this work.

The definition of a grid allows to construct a discrete space V_h with a basis ϕ_i conforming to the previously mentioned requirements. One possible choice is

$$V_h = \{v_h \in C^0(\Omega) \,:\, v_h|_\tau \in P,\ \forall \tau \in \mathcal{T}_h\}$$

with the basis $\{\phi_i\}_{i=1}^N \subset V_h$, satisfying $\phi_i(v_j) = \delta_{ij}$, where v_j are the vertices of the grid $\mathcal{T}_h$, and $P = \mathbb{P}_1$ if simplicial elements are used, or $P = \mathbb{Q}_1$ if cubical elements are used. These functions are also known as hat functions or the Lagrange basis of order 1. The support of one basis function ϕ_i is restricted to the elements sharing the vertex v_i. Therefore, the entry $a_{i,j} = a(\phi_j, \phi_i) \neq 0$ only for ϕ_j with their associated vertex v_j contained in one of the elements sharing v_i. For a large class of grids, so called quasi-uniform grids, the number of elements sharing one vertex is bounded, showing that the resulting stiffness matrix is sparse.

Other spaces are also viable, for example by increasing the element-wise polynomial degree, which requires nodes placed on entities with codimension less than d, in addition to the vertices, to define the condition $\phi_i(v_j) = \delta_{ij}$. All of those points v_j on the grid that define the basis $\{\phi_i\}$ are known as nodes. If the local polynomial degree is p, the basis is called Lagrange basis of order p, and for this basis the same argument for the sparsity of A holds. The discrete functions need to be continuous over $\overline{\Omega}$ to ensure $V_h \subset V$, otherwise the finite element method would be non-conforming, which exacerbates the definition of the discretized weak formulation. An example for a non-conforming method is the discontinuous Galerkin method (DG), where the continuity requirement along element boundaries is dropped, see [51], [20], or [77] for details.

2.1.3 Assembly

The construction of V_h and its basis $\{\phi_i\}$ above guarantee a sparse stiffness matrix A, but due to the abstract definition of ϕ_i, the computation of the integral $a(\phi_j, \phi_i)$ is not immediately clear. In 1D the functions ϕ_i can be explicitly defined, while in 2D or 3D this becomes difficult for unstructured grids. As a first step, it is necessary to define the local-to-global mapping $g^\tau : \{1, \dots, n\} \mapsto \{1, \dots, N\}$ for every element τ defined by the nodes v_i, $i \in \{1, \dots, N\}$, which describes a local numbering of the element's nodes by satisfying

$$T_\tau(\hat{v}_j) = v_{g^\tau(j))},$$

where $\hat{v}_j$, $j = 1, \ldots, n$, are the nodes of the reference element. Additionally, let $\widehat{\phi}_i$, $i = 1, \ldots, n$ be the Lagrange basis of order 1 on the reference element, i.e. $\widehat{\phi}_i \in \mathbb{Q}_1$ and $\widehat{\phi}_i(\hat{v}_j) = \delta_{ij}$ for $i, j = 1, \ldots, n$. The localization of a basis function ϕ_i with $\operatorname{supp}(\phi_i) \subset \tau$ can now be written as

$$\phi_i|_\tau = \sum_{l=1}^{n} \delta_{g^\tau(l),i}\, \widehat{\phi}_l \circ T_\tau^{-1}.$$

It should be noted that only one term is non-zero.

Using this localization, the computation of A reduces to computing local contributions and assembling them into the global data structure. As a first step, splitting the integral over Ω into integrals over the grid elements τ results in

$$\begin{aligned} a(\phi_j, \phi_i) &= \int_\Omega \nabla v \cdot D \nabla u + v(b \cdot \nabla u + cu) \, \mathrm{dx} = \int_\Omega h(\phi_j, \phi_i) \, \mathrm{dx} \\ &= \sum_{\tau \in \mathcal{T}_h} \int_\tau h(\phi_j, \phi_i) \, \mathrm{dx} \end{aligned}$$

for a problem without Robin boundary conditions. Then, applying the local description of the basis function leads to

$$\begin{aligned} a(\phi_j, \phi_i) &= \sum_{\tau \in \mathcal{T}_h} \sum_{l,k=1}^{n} \delta_{g^\tau(l),j}\, \delta_{g^\tau(k),i} \int_\tau h\left(\widehat{\phi}_k \circ T_\tau^{-1}, \widehat{\phi}_l \circ T_\tau^{-1}\right) \, \mathrm{dx} \\ &= \sum_{\tau \in \mathcal{T}_h} \sum_{l,k=1}^{n} \delta_{g^\tau(l),j}\, \delta_{g^\tau(k),i} A^\tau_{l,k}. \end{aligned}$$

The $n \times n$ matrix $A^\tau = \left[A^\tau_{l,k}\right]_{l,k}^{n}$ is known as the local stiffness matrix and together with the boolean restriction matrix $R^\tau = \left[\delta_{g^\tau(i),j}\right]_{i,j=1}^{n,N} \in \mathbb{R}^{n \times N}$ the computation of the full stiffness matrix can be written compactly as

$$A = \sum_{\tau \in \mathcal{T}} (R^\tau)^T A^\tau R^\tau.$$

The localized formulation of the basis functions shows that its evaluation is equivalent to evaluate the corresponding basis function on the reference element, which can be further exploited using the transformation rule for $\tau = T_\tau(\hat{\tau})$. Assume $b(x) = 0$ for simplicity, then by transforming the integration domain the integral can be written as

$$
\begin{aligned}
A^{\tau}_{l,k} &= \int_{\tau} \nabla\phi_k\left(T_\tau^{-1}(x)\right)\cdot D(x)\nabla\phi_l\left(T_\tau^{-1}(x)\right) + c(x)\phi_k\left(T_\tau^{-1}(x)\right)\phi_l\left(T_\tau^{-1}(x)\right)\,\mathrm{dx} \\
&= \int_{\hat\tau} |\det J(\hat x)|\Big(J^{-T}(\hat x)\nabla\widehat{\phi}_k(x)\cdot D\left(T_\tau(\hat x)\right)J^{-T}(\hat x)\nabla\widehat{\phi}_l(x) \\
&\qquad\qquad + c\left(T_\tau(\hat x)\right)\widehat{\phi}_k(x)\widehat{\phi}_l(x)\Big)\,\mathrm{d}\hat x,
\end{aligned}
$$

where J is the Jacobian matrix of T_τ, and the factor J^{-T} is introduced through the chain rule to scale the gradients correctly. Since these integrals cannot be computed analytically in general, a quadrature rule is used to approximate the integral. For a quadrature rule with points ξ_q and weights ω_q, this leads to

$$
A^{\tau}_{l,k} = \sum_q \omega_p H(\xi_q, \widehat{\phi}_k, \widehat{\phi}_l),
$$

where H is the evaluation of the integrand at a quadrature point. Algorithm 2.1 , based on the description in [33], displays the global assembly of the stiffness matrix A for a linear second order elliptic PDE with coefficients D, b, and c on a grid $\mathcal{T}$, together with the corresponding computation of the local stiffness matrices.

The application of the stiffness matrix A to a vector can be computed by assembling the element-local contributions, similar to the assembly of A. As a first step, the entry $(A\mathbf{u})_i$ is rewritten as

$$
\begin{aligned}
\mathbf{y}_i = (A\mathbf{u})_i &= \sum_{j=1}^{N} A_{i,j}\mathbf{u}_j = \sum_{j=1}^{N} a(\phi_j,\phi_i)\mathbf{u}_j \\
&= a\left(\sum_{j=1}^{N}\mathbf{u}_j\phi_j, \phi_i\right) = a(u_h,\phi_i).
\end{aligned}
$$

Now, using the local-to-global map g^τ defined above, the evaluation of $a(\cdot,\cdot)$ is split into the element-local computations

$$
\begin{aligned}
a(u_h,\phi_i) &= \sum_{\tau\in\mathcal{T}_h}\sum_{l=1}^{n}\delta_{g^\tau(l),j}\int_\tau h\left(\sum_{k=1}^{n}\mathbf{u}_{g^\tau(k)}\widehat{\phi}_k\circ T_\tau^{-1}, \widehat{\phi}_l\circ T_\tau^{-1}\right)\mathrm{dx} \\
&= \sum_{\tau\in\mathcal{T}_h}\sum_{l=1}^{n}\delta_{g^\tau(l),j}\,\mathbf{y}^\tau_l,
\end{aligned}
$$

Algorithm 2.1: Pseudo code for the assembly of global stiffness matrix A

```
Function: AssembleStiffnessMatrix($\mathcal{T}, D, b, c$)
    $A = 0$
    for $\tau \in \mathcal{T}$ do
        $A^\tau$ = LocalStiffnessMatrix($\tau, D, b, c$)
        for $l = 1, \dots, n$ do
            for $k = 1, \dots, n$ do
                $i$ = LocalToGlobal($\tau, l$)
                $j$ = LocalToGlobal($\tau, k$)
                $A_{i,j} = A_{i,j} + A^\tau_{l,k}$
Function: LocalStiffnessMatrix($\tau, D, b, c$)
    $A^\tau = 0$
    for $(\xi, \omega) \in$ QuadratureRule($\tau$, order) do
        $x = T^\tau(\xi)$
        jit = $DT_\tau(\xi)^{-T}$
        weight = $\omega * |\det DT_\tau(\xi)|$
        for $l = 1, \dots, n$ do
            $\mathrm{phi}_l = \hat{\phi}_l(\xi)$
            $\mathrm{grad}_l = \mathrm{jit} \cdot \nabla\hat{\phi}_l(\xi)$
        for $l = 1, \dots, n$ do
            for $k = 1, \dots, n$ do
                $t_1 = \sum_{i,j}^{d} \mathrm{grad}_{l,i} * D(x)_{i,j} * \mathrm{grad}_{k,j}$
                $t_2 = \mathrm{phi}_l * \sum_{i}^{d} b(x)_i * \mathrm{grad}_{k,i}$
                $t_3 = \mathrm{phi}_l * c(x) * \mathrm{phi}_k$
                $A^\tau_{l,k} = A^\tau_{l,k} + \mathrm{weight} * (t_1 + t_2 + t_3)$
```

and with the restriction matrix R^τ as before, the global application of A is given by

$$\mathbf{y} = \sum_{\tau \in \mathcal{T}_h} (R^\tau)^T \mathbf{y}^\tau.$$

This computation is called matrix-free operator application, and is displayed in algorithm 2.2 . The operation $\mathbf{u}_k^\tau = \mathbf{u}_{g^\tau(k)}$ is known as a gather operation, or short gather, and the reverse operation $\mathbf{y}_{g^\tau(l)} = \mathbf{y}_l^\tau$ as a scatter operation. Due to their localized nature, both the computation of the local stiffness matrix A^τ and the computation of the local operator application y^τ are called *local kernels*.

2.1.4 Extensions

The FEM introduced here was limited to linear second order elliptic PDEs, but the FEM can also be applied to other kinds of PDEs. Some of those are briefly discussed here. For the discretization of a non-linear PDE assume that it can be written as

$$\begin{aligned} &\text{Find } u \in V \text{ such that:} \\ &r(u, v) = 0 \quad \forall v \in V, \end{aligned}$$

with an appropriate function space V and the residual $r(\cdot, \cdot)$, which is linear in its second component. This so-called residual formulation can be derived using the same techniques as in the linear case, or from a minimization problem, as detailed in [34]. By restricting the problem to a finite dimensional subspace $V_h \subset V$ with basis $\{\phi_i\}_{i=1}^N$ the solution is found by solving the finite dimensional non-linear system

$$R(\mathbf{z}) = 0, \quad \text{with } R(\mathbf{z}) = \left[r\left(\sum_{j=1}^N \mathbf{z}_j \phi_j, \phi_i \right) \right]_{i=1}^N .$$

The non-linear system can be solved iteratively with the Newton method. For this method, the iterates are given by

$$\begin{aligned} \mathbf{z}^{k+1} &= \mathbf{z}^k + \alpha_k \Delta \mathbf{z}^k, \\ DR(\mathbf{z}^k)\Delta \mathbf{z}^k &= -R(\mathbf{z}^k), \end{aligned}$$

Algorithm 2.2: Pseudo code for the assembly of the operator application to a vector **u**

```
Function: MatrixFreeOperatorApplication($\mathcal{T}, \mathbf{u}, D, b, c$)
    $\mathbf{y} = 0$
    for $\tau \in \mathcal{T}$ do
        for $k = 1, \dots, n$ do
            $j$ = LocalToGlobal($\tau, k$)
            $\mathbf{u}^\tau_k = \mathbf{u}_j$
        $\mathbf{y}^\tau$ = LocalOperatorApplication($\tau, \mathbf{u}^\tau, D, b, c$)
        for $l = 1, \dots, n$ do
            $i$ = LocalToGlobal($\tau, l$)
            $\mathbf{y}_i = \mathbf{y}_i + \mathbf{y}^\tau_l$
Function: LocalOperatorApplication($\tau, \mathbf{u}^\tau, D, b, c$)
    $\mathbf{y}^\tau = 0$
    for $(\xi, \omega) \in$ QuadratureRule($\tau$, order) do
        $x = T^\tau(\xi)$
        jit = $DT_\tau(\xi)^{-T}$
        weight = $\omega * |\det DT_\tau(\xi)|$
        for $l = 1, \dots, n$ do
            $\mathrm{phi}_l = \widehat{\phi}_l(\xi)$
            $\mathrm{grad}_l = \mathrm{jit} \cdot \nabla\widehat{\phi}_l(\xi)$
        $\mathrm{u}_l = \sum_{l=1}^{n} \mathbf{u}^\tau_l * \mathrm{phi}_l$
        $\mathrm{gradu}_l = \sum_{l=1}^{n} \mathbf{u}^\tau_l * \mathrm{grad}_l$
        for $l = 1, \dots, n$ do
            $t_1 = \sum_{i,j}^{d} \mathrm{grad}_{l,i} * D(x)_{i,j} * \mathrm{gradu}_j$
            $t_2 = \mathrm{phi}_l * \sum_{i}^{d} b(x)_i * \mathrm{gradu}_k$
            $t_3 = \mathrm{phi}_l * c(x) * \mathrm{u}$
            $\mathbf{y}^\tau_l = \mathbf{y}^\tau_l + \mathrm{weight} * (t_1 + t_2 + t_3)$
```

where in each iteration the Jacobian of R has to be assembled and a linear system with it needs to be solved. A broad overview of the Newton method in different contexts can be found in [28]. If R is linear, i.e. $r(u, v) = a(u, v) - F(v)$, then the method is equivalent to the approach outlined before, since $DR(\mathbf{z}) = A$, and the method converges after one step.

The previous discussion was restricted to stationary PDEs, where the solution u does not vary in time. Two important categories of time-dependent PDEs are parabolic and hyperbolic PDEs, where a parabolic PDE can be written as

$$\partial_t u(t, x) + \mathcal{L}u(t, x) = f(t, x) \text{ on } [0, T] \times \Omega$$

and a hyperbolic PDE as

$$\partial_t u(t, x) + \operatorname{div} F(t, x, u(t, x)) = f(t, x) \text{ on } [0, T] \times \Omega.$$

Both categories require boundary conditions on $[0, T] \times \partial\Omega$ and on $\{0\} \times \Omega$.

In the parabolic case, which are discussed in [78] or [33] besides many others, the operator $\mathcal{L}$ is a second order elliptic operator, such as defined in the beginning of this section, and a weak formulation for this problem is deduced in the same manner as in the stationary case leading to

$$\frac{d}{dt} \int_\Omega u(t, x) v(x) \, \mathrm{dx} + r(u, v) = 0,$$

where the residual formulation is used. By applying the so-called method of lines, the PDE is first discretized in space with the FEM, resulting in

$$\frac{d}{dt} M u_h(t) + R(u_h(t)) = 0,$$

with the mass matrix $M = [(\phi_j, \phi_i)_{L^2}]_{i,j=1}^N$ and the time dependent coefficient vector $u_h : [0, T] \mapsto \mathbb{R}^N$. A suitable time stepping method is then used to discretize the equation in time, for example with the θ-method one obtains

$$M\mathbf{u}^{k+1} = M\mathbf{u}^k - \Delta t^k \left(\theta R(\mathbf{u}^k) - (1 - \theta) R(\mathbf{u}^{k+1})\right).$$

Higher order Runge-Kutta methods may be employed to increase the accuracy of the time discretization.

A hyperbolic PDE, where F may be a non-linear, vector-valued function, can be handled similarly. But, other discrete function spaces than those discussed in 2.1.2 should be used. Usually the solution for these types of PDEs contains discontinuities, which are only inadequately resolved by continuous basis functions. Instead, the discontinuous Galerkin method or the finite volume method could be used, for an introduction see the series [24, 25, 22, 21, 23] or [66].

2.2 Iterative Linear System Solvers

The discretization of a PDE with the finite element methods leads to a linear system

$$A\mathbf{x} = \mathbf{b},$$

which needs to be solved. Since the discrete function space V_h needs to be large to minimize the error w.r.t. the weak solution, direct solvers like the Gauß elimination are not feasible, and instead iterative methods are used. These methods solve the system approximately by constructing a sequence $\mathbf{x}^0, \mathbf{x}^1, \dots$ that converges to the solution $\mathbf{x}^*$. A useful quantity for most iterative methods is the residual, or defect,

$$\mathbf{r}^k = \mathbf{b} - A\mathbf{x}^k = A\mathbf{x}^* - A\mathbf{x}^k = A\left(\mathbf{x}^* - \mathbf{x}^k\right) = A\mathbf{e}^k.$$

This can be used to define the iteration sequence or as part of a stopping criterion to determine if the approximation is close enough.

2.2.1 Simple Iterative Methods

A simple definition of an iterative method is given by rewriting the linear system as a fixed-point equation

$$\mathbf{x} = G\mathbf{x} + \mathbf{f},$$

where $G = M^{-1}N$ and $\mathbf{f} = M^{-1}\mathbf{b}$ are defined by the matrix splitting

$$A = M - N,$$

with a non-singular matrix M, and M^{-1} and N non-negative. The iteration

$$\mathbf{x}^{k+1} = G\mathbf{x}^k + \mathbf{f}$$

converges to the fixed-point, which coincides with the solution of the linear system, if the spectral radius $\rho(G) < 1$. In the case that A is non-singular and A^{-1} is non-negative, convergence can be guaranteed, as shown in [81]. An equivalent formulation of this iteration reads as

$$\begin{aligned}\mathbf{x}^{k+1} &= M^{-1}N\mathbf{x}^k + M^{-1}\mathbf{b}\\ &= (\mathbb{1} - M^{-1}A)\mathbf{x}^k + M^{-1}\mathbf{b}\\ &= \mathbf{x}^k + M^{-1}(\mathbf{b} - A\mathbf{x}^k)\end{aligned}$$

For the efficient solution of the linear system some conditions on the choice of M need to be imposed. The convergence rate of this method is given by the Banach fixed-point theorem, which states that

$$||\mathbf{e}^{k+1}|| \leq q||\mathbf{e}^k|| \quad \text{and} \quad ||\mathbf{e}^k|| \leq \frac{q^k}{1-q}||\mathbf{x}^1 - \mathbf{x}^0||,$$

where $q = \rho(G) = \rho(M^{-1}N) = \rho(\mathbb{1} - M^{-1}A)$. It follows that the spectral radius of $\mathbb{1} - M^{-1}A$ should be as small as possible to ensure a fast convergence of this method. Furthermore, the application of M^{-1} should be cheap to compute, or, more specifically, comparable to the application of A. Balancing these two requirements is a difficult task, for example the best choice for the first condition is using $M^{-1} = A^{-1}$, but this also the worst choice for the second condition, since applying A^{-1} was the initial problem. On the other side, the application cost of M^{-1} is optimized with the Richardson method $M^{-1} = \alpha\mathbb{1}$, $\alpha > 0$, which neglects the first condition. Going forward, the matrix M^{-1} is called preconditioner within this context.

Some important methods are derived from the splitting $A = D + L + R$ with

$$R = \begin{bmatrix} 0 & A_{12} & \cdots & A_{1N}\\ 0 & \ddots & \ddots & \vdots\\ \vdots & \ddots & \ddots & A_{N-1,N}\\ 0 & \cdots & 0 & 0 \end{bmatrix}, L = \begin{bmatrix} 0 & 0 & \cdots & 0\\ A_{2,1} & \ddots & \ddots & \vdots\\ \vdots & \ddots & \ddots & 0\\ A_{N,1} & \cdots & A_{N,N-1} & 0 \end{bmatrix},$$

$$D = \begin{bmatrix} A_{11} & 0 & \cdots & 0\\ 0 & \ddots & \ddots & \vdots\\ \vdots & \ddots & \ddots & 0\\ 0 & \cdots & 0 & A_{NN} \end{bmatrix}.$$

The choice $M = D$ leads to the Jacobi method with the iteration

$$\mathbf{x}^{k+1} = \mathbf{x}^k + D^{-1}(\mathbf{b} - A\mathbf{x}^k),$$

which can also be formulated component-wise as

$$\mathbf{x}_i^{k+1} = \frac{1}{A_{ii}}\left(\mathbf{b}_i - \sum_{\substack{j=1\\ j\neq i}}^{N} A_{ij}\mathbf{x}_j^k\right).$$

Further iterative solver can be defined using this splitting. The Gauß-Seidel method uses $M = D + L$, the successive over relaxation (SOR) method $M = \left(\frac{D}{\omega} + L\right)$, $\omega \in (0,2)$, and, if $U = L^T$, the symmetrized SOR (SSOR) method can be used with $M = \left(\frac{D}{\omega} + L\right)\frac{\omega}{2-\omega}D^{-1}\left(\frac{D}{\omega} + R\right)$.

For symmetric and positive definite matrices A and M, the convergence rate can be estimated by considering the preconditioned system $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$ and applying the Richardson iteration. The resulting update is defined as

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha M^{-1}(\mathbf{b} - A\mathbf{x}^k).$$

The optimal convergence rate of the Richardson iteration is achieved for $\alpha = 2/(\lambda_{\min}(M^{-1}A)+\lambda_{\max}(M^{-1}A))$, where $\lambda_{\min}, \lambda_{max}$ denote the minimal and maximal eigenvalue of $M^{-1}A$ respectively, which leads to the convergence rate

$$q_{\text{opt}} = \frac{\kappa(M^{-1}A) - 1}{\kappa(M^{-1}A) + 1} = 1 - \frac{2}{\kappa(M^{-1}A)},$$

as shown in [81]. The quantity $\kappa(B) = \frac{\lambda_{\max}(B)}{\lambda_{\min}(B)}$ is known as the condition number of the matrix B. Therefore, the method requires

$$k \approx \frac{1}{2}\kappa(M^{-1}A)\ln\left(\frac{2}{\varepsilon}\right) = \mathcal{O}(\kappa(M^{-1}A))$$

iterations to reduce the initial error by a factor of ε. In the case of the FEM the condition number of the stiffness matrix has the estimate

$$\kappa(A_h) = \mathcal{O}(h^{-2}),$$

where h denotes the grid size, derived for example in [33], which shows that increasing the accuracy of the discretization also increases the number of iteration necessary, if no preconditioner is used. The other methods presented here so far display the same asymptotic behavior. A good choice for the preconditioner M^{-1} then satisfies the following relation

$$c_1\langle M\mathbf{x},\mathbf{x}\rangle_2 \leq \langle A\mathbf{x},\mathbf{x}\rangle_2 \leq c_2\langle M\mathbf{x},\mathbf{x}\rangle_2$$

with grid independent constants c_1, c_2, which leads to constant iteration numbers under grid refinement.

2.2.2 Krylov Subspace Methods

A better convergence rate can be obtained by using Krylov subspace methods, which seek an approximation $\mathbf{x}^m$ within an affine subspace $\mathbf{x}^0 + \mathcal{K}^m$ of dimension m. The subspace $\mathcal{K}^m$ is defined as

$$\mathcal{K}^m(A,\mathbf{r}^0) = \{\mathbf{r}^0, A\mathbf{r}^0, A^2\mathbf{r}^0, \dots, A^{m-1}\mathbf{r}^0\},$$

with the residual $\mathbf{r}^0 = \mathbf{b} - A\mathbf{x}^0$ and the initial guess $\mathbf{x}^0$. The most prominent Krylov subspace method is the CG (conjugate gradients) method, which requires that A is symmetric and positive definite. For general matrices the GMRES (generalized minimum residual) method or the BiCGStab method can be used. Since the discretization of an elliptic PDE without an advection term ($b(x) = 0$) usually leads to a symmetric and positive definite stiffness matrix, the CG method is especially popular in the context of the FEM.

While the construction of the iterates is more demanding for Krylov methods compared to the simple iterative methods, the benefit of these methods is the increased convergence rate. Nevertheless, the complexity of the Krylov method is not higher than the application of A or M^{-1}. For the preconditioned CG method it can be shown that

$$||\mathbf{x}^m - \mathbf{x}^*||_A \leq 2\left(\frac{\sqrt{\kappa(M^{-1}A)} - 1}{\sqrt{\kappa(M^{-1}A)} + 1}\right)^m ||\mathbf{x}^0 - \mathbf{x}^*||_A,$$

which reduces the necessary number of iterations to $\mathcal{O}(\sqrt{\kappa(M^{-1}A)})$, see [86]. Therefore, the linear systems appearing in the FEM context are usually solved using a Krylov method, with an appropriate preconditioner.

There are numerous preconditioner to choose from, for instance the simple iterative methods from the previous section 2.2.1. In this case, the application of the preconditioner $\mathbf{z} = M^{-1}\mathbf{r}$ is to be understood as solving the system $A\mathbf{z} = \mathbf{r}$ with one of the presented methods for a fixed number of iterations, usually 1–3. Of course more sophisticated preconditioners are possible. One example is the incomplete LU (ILU) factorization, which needs a more expansive setup phase, but yields better convergence than the discussed simple methods. Among the best currently available preconditioner for FEM are variants of a multigrid methods, which can achieve mesh size independent convergence. The highly PDE dependent geometric multigrid method constructs a hierarchy of discretized problems and propagates the solution for the coarsest problem to the finest one. A generic version without creating new discretization besides the finest one is the algebraic multigrid (AMG) method.

Regardless of which iterative method discussed here is used to solve the FEM discretized system $A\mathbf{u} = \mathbf{f}$ they all share similar features. Each method only requires that the evaluation of $A\mathbf{u}$ is available in some form, which could be provided as an extra function like in section 2.1.3 without assembling the stiffness matrix. A preconditioner is necessary to reduce the condition number of A, ideally in such a way that it does not depend on the mesh size. While the preconditioner should efficiently decrease the condition number of A, its evaluation should be fast and comparable to the evaluation of A. The preconditioners may lead to stricter requirements on A than the availability of the matrix-vector-multiplication. In fact, most preconditioners discussed so far need that the matrix A is fully assembled, which may be relaxed for the simple iterative methods to the requirement that each row may be provided. This is the case if, for example, a discretization with stencils is used.

2.3 Software Tools

The computer-aided solution of a PDE requires implementing the chosen discretization method using a suitable programming language. Since many components of the implementation are similar for different PDEs and discretization methods, software frameworks emerged providing these basic building blocks to reduce the user's coding efforts. The frameworks can be broadly divided into specialized and general-purpose tools. The specialized frameworks are targeted at a specific type of PDE or discretization method, most often at both, allowing them to optimize heavily for their specific targets, while also simplifying the necessary user implementations. Examples for these kinds of frameworks are OpenFOAM[1], which is restricted to finite volume methods, the computational fluid dynamics solvers Nek5000[2] and Clawpack[3], or code_aster[4],

[1]https://openfoam.com/
[2]https://nek5000.mcs.anl.gov/
[3]http://clawpack.org/
[4]https://code-aster.org/

which is targeted at structural mechanics. In contrast, general-purpose frameworks, such as deal.ii[5], libMesh[6], NGSolve[7], or DUNE[8], are less restrictive regarding the considered PDEs or discretization. The DUNE framework, for example, requires only that the discretizations involves a mesh. Usually the permissiveness comes at the cost of higher implementation burden on the user.

Due to the large linear or non linear systems, solving a discretized PDE usually requires high performance systems and software. Therefore, the software frameworks need to be implemented in a programming language that allows to efficiently use the hardware resources, which mostly reduced the language choices to FORTRAN, C, or C++. From the mentioned frameworks, Nek500 and Clawpack are written in FORTRAN, while the general-purpose frameworks, as well as OpenFOAM, are written in C++. The downside of using these languages is that they require a deep understanding of the language to achieve even simple tasks such that they are often perceived as user-unfriendly. Especially beginners may be discouraged from using these frameworks.

In recent years a push towards more user-friendly and simpler interfaces has been made, usually relying on higher level languages. For instance, scripting languages are used to provide simple wrappers for the underlying lower level framework, e.g. OpenFOAM, PyClaw for CLAWPACK [56], or deal2lkit for deal.ii [83]. A similar approach is the usage of so-called domain specific languages (DSL). Here, the description of a particular discretization is defined in a simple language, which could be either newly designed, like ExaSlang[9] [85] or FreeFEM[10] [50], or embedded within another language, such as the FEniCS component UFL embedded in Python.

In the following the general-purpose framework DUNE is discussed in more detail, as well as the UFL. Both are used in later parts of this work, DUNE as the basic infrastructure for a PDE solver and UFL as the input for generating efficient local kernels to be used within DUNE.

2.3.1 DUNE

DUNE [8, 9, 11] is a framework for the solution of PDEs, which uses modern C++ techniques to define flexible interfaces and achieve high efficiency. One of the main principles in the development of DUNE is the separation of algorithms

[5]https://dealii.org/
[6]http://libmesh.github.io/
[7]https://ngsolve.org/
[8]https://dune-project.org/
[9]https://www.exastencils.fau.de/
[10]https://freefem.org/

and data structures, similar to the design of the C++ STL, which allows designing general algorithms applicable to multiple problems and using a specialized data structure most suitable for the specific problem. To this end, DUNE defines abstract interfaces for the components of a PDE discretization, which can be implemented by different providers for specific use cases. The most notable example of this pattern is the grid interface implemented by different grid managers specializing for different types of grids. Writing an algorithm based only on the abstract grid interface and agnostic of a concrete implementation allows to easily exchange the grid manager. If a specialized grid manager is developed and swapped in, the benefits from the specialization are immediately available.

The interfaces defined by the DUNE framework are separated into distinct libraries, called modules, striving for a clear separation of concerns. Since DUNE predates the C++-20 modules, they should not be understood as those. The modules can be grouped into the following categories[11]

Core The foundational modules of the DUNE framework. They cover a specialized build system, dense linear algebra (DUNE-COMMON), sparse linear algebra (DUNE-ISTL), the grid interface and some implementations (DUNE-GRID), reference elements (DUNE-GEOMETRY), and local finite elements (DUNE-LOCALFUNCTIONS), among others.

Grid These modules offer additional implementations of the grid interface defined in DUNE-GRID, for example DUNE-UGGRID, or DUNE-ALUGRID.

Discretization Discretizing a PDE requires additional concepts that have not been covered so far. The discretization modules provide, for instance, discrete function spaces, generalized assembly, or simplified local kernel definitions. Widely used modules are DUNE-PDELAB, DUNE-FEM, DUNE-FUFEM, and DUNE-GDT.

Application Modules contained here are tailored to a specific kind of application, usually based on a discretization module, for example DuMu^{X}[12] or OPM[13], which focus on flow in porous media, or duneuro[14] for applications in neuroscience.

A thorough introduction to the core concepts and modules of DUNE can be found in Sander's textbook [82].

[11]Many of the modules can be found in under `https://gitlab.dune-project.org`
[12]`https://dumux.org`
[13]`https://opm-project.org`
[14]`http://duneuro.org/`

The most important DUNE module for this thesis is DUNE-PDELAB. Therefore, its addition beyond the scope of the core modules to the simplification of the solution process are discussed with more detail. One significant building block of the discretization not covered by the core modules is the definition of a discrete function space, in which the solution of the weak formulation is sought. DUNE-PDELAB introduces these with the `GridFunctionSpace` type that combines a grid view with information on the local finite element and global constraints on the discrete functions, which can be more complicated than simple Dirichlet constraints. The `LocalFiniteElementMap` interface provides context for the local finite element by mapping each grid element to a local finite element from DUNE-LOCALFUNCTIONS, with additional information on the local DoFs. It is possible to construct a function space as the product of two or more functions spaces, for example to define a vector valued finite element, or the Taylor-Hood element, using a compile-time tree structure. With the concept of a discrete function space available, other closely related concepts such as discrete functions, or interpolations, are also provided.

Another significant part of the solution process is the grid-based assembly of different quantities, e.g. residual evaluation, matrix assembly, or matrix-free operator application. In the DUNE-PDELAB module this is simplified by the `GridOperator` type, which applies these assembly processes for given function spaces. The local integration kernels required by the different assembly processes are gathered into a class, satisfying `LocalOperator` interface. Since the implementation depends directly on the PDE and the discretization used, the class has to be provided by the user, although it is only necessary to implement the relevant kernels. For standard PDEs, such as convection-diffusion equations, or Maxwell equations, default implementations already exist.

Using DUNE-PDELAB the implementation of a discretization method for the solution of a PDE reduces to picking the right building blocks provided by the module, adding user specific implementations where necessary, and fitting them together appropriately. Which part the user needs to supply depends strongly on the considered kind of application or research. For example, if a type of PDE not covered by the standard `LocalOperator` is studied, a new implementation of the `LocalOperator` interface has to be provided. However, if a new time stepping method is considered for a well known PDE it might not be necessary to implement a new `LocalOperator`, instead only adjustments to the available time stepping methods are required. Nevertheless, users might still want to use optimized `LocalOperator`, since the computations within the `LocalOperator` can take up significant portion of the total runtime, especially in matrix-free methods.

2.3.2 UFL

As outlined, DUNE provides most parts necessary for solving a PDE, reducing the user's effort noticeably, but this assumes at least some familiarity with modern C++. This entry barrier can be lowered by using a domain specific language (DSL), which models only the concepts required in the targeted domain. UFL [3, 4] is such a DSL for describing discrete weak formulations of PDEs directly embedded in Python, i.e. a valid UFL description is also a valid Python program. It is the front end of the FEniCS ecosystem [2], which tries to automate the solution process for a discretized PDE by generating finite element code based on an abstract problem definition. Additionally, it has been adapted by other code generator such as Firedrake [80].

The main focus of UFL is to provide a simple language that is as close as possible to the mathematical notation used for weak formulations. This is achieved by providing nearly direct mappings from mathematical concepts to Python functions or classes. For instance, `FunctionSpace(mesh, "CG", 1)` represents a discrete function space with a continuous Lagrange basis of order 1 on a predefined `mesh`, and `grad(u)` defines the gradient of a (possible vector valued) element within such a function space. Defining the integral of an expression over the whole domain is easily expressed as `expr * dx`, where `dx` is a predefined variable. Integrals over the boundary can be defined similarly with the variable `ds`, and for integration over interior faces `dS` is used. To demonstrate the simplicity of the UFL, the residual formulation of the Poisson problem in UFL reads as:

```
mesh = # triangulation of Omega
V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = # analytic definition of f
r = dot(grad(u), grad(v)) * dx - f * v * dx
```

Due to its similarity to the mathematical formulation, the intent of this excerpt can be understood without knowledge of the python language, which would not be true for the DUNE-PDELAB counterpart.

The previous discussion shows that using UFL as a front end leads to more accessible simulation software compared to the DUNE framework, which also enables rapid prototyping. But it should be noted that only discretizations based on the weak formulation are possible. This limits the support for other types of discretizations, while more complex frameworks, like DUNE, have less strict requirements on the discretizations. For example finite volume schemes are difficult to implement using UFL, since they require concepts that can not be represented as a weak formulation. An additional advantage of UFL is its capability to apply automatic differentiation to a symbolic representation of a non-linear form, which is often times missing from C++ frameworks, and thereby simplifying the derivation for non-linear PDEs.

The internal description of a linear or bilinear form in UFL uses abstract syntax trees (AST) to represent integrand expressions such as `dot(grad(u), grad(v))`. For reference, figure 2.1 displays the AST for the residual form defined above with $f = -4$. The nodes appearing in an AST are categorized in either terminal nodes, which are leaves in the tree, and operator nodes, which have child nodes. Mathematical operators, such as `Sum` or `Product`, and functions, e.g. `Sin` and `Abs`, are modeled as operator nodes, as well as indexing nodes, necessary for handling vector valued components like the gradient of a function. In addition, compound derivatives, such as `Grad`, `Div`, or `Curl`, and spatial derivatives are also considered as operator nodes. The terminal nodes contain the form arguments, i.e. the function space arguments of the represented linear or bilinear form, geometric quantities, such as `SpatialCoordinate` denoting the position of a quadrature point, or `JacobianInverse` modeling DT_τ^{-T}, indices used in indexed nodes, and constant values.

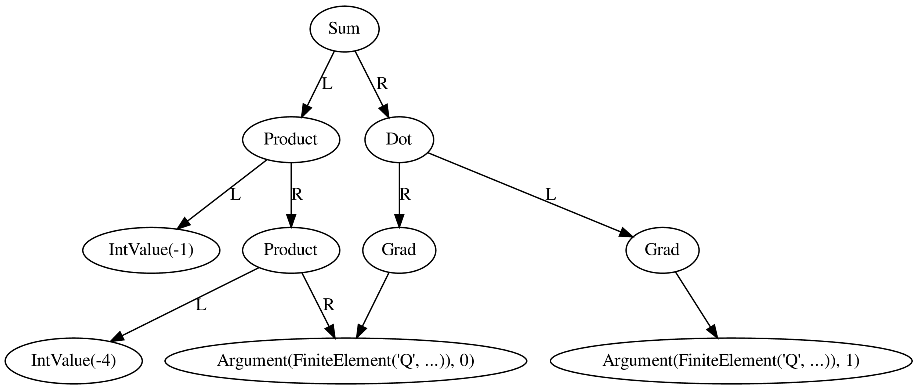
Sum
L
R
Product
Dot
L
R
R
L
IntValue(-1)
Product
Grad
Grad
L
R
IntValue(-4)
Argument(FiniteElement('Q', ...)), 0)
Argument(FiniteElement('Q', ...)), 1)

Figure 2.1: Example of an AST in UFL representing the expression $\nabla u \cdot \nabla v - (-4)v$.

UFL provides several algorithms acting on the AST representation, with efficient implementations for tree traversals and type-based dispatch at its core. The dispatch approach allows simple implementations of tree transformations based on the visitor pattern[15], i.e. the AST is traversed and at each node of the AST the correct transformation, depending on the node's type, is chosen. Several essential transformations are supplied by UFL. These contain the rewriting of so-called compound algebraic nodes (e.g. `dot`) in terms of simple indexed sums, applying standard derivation rules, or transforming the integration domain into reference coordinates, as explained in 2.1.3, which automatically introduces geometry scaling terms such as $\det DT_\tau(x)$ or DT_τ^{-T}.

The DSL defined by UFL provides convenient tools to define a weak formulation for a PDE, close enough to the mathematical notation that it can be easily used even without programming experience. However, the representation in UFL does not solve the PDE by itself. Instead, the provided AST and algorithms are used to translate the mathematical formulation into code that can be compiled or executed. For example, FEniCS uses its form compiler FFC and UFC to generates C++ code for the element local operations and local-to-global maps which are used within the DOLFIN framework, and Firedrake uses the TSFC to generate C code for both the general assembly and the local operations. Since the generation is handled automatically after the definition of the weak formulation in UFL, solving a PDE with a tool chain that uses UFL as a front end is nearly as simple as deriving the mathematical formulation.

2.4 Performance on Modern Hardware

For a simulation framework, its convenience is not the only important feature, it has to be balanced with the performance that can be achieved. It may not be an advantage of a framework to implement a particular discretization in a couple of minutes, if the solution process takes hours, or even days, due to inefficiencies in the framework. One particular bottleneck that is shared by most frameworks is the operator application during the iterative solution of a linear system. The classical approach for solving the linear system associated with a discretization uses a preconditioned Krylov solver with the assembled stiffness matrix. Central to this approach is the sparse matrix vector multiplication, short SpMxV. In the early days of the FEM the SpMxV was very efficient, but, due to the development of computer hardware, the efficiency steadily decreased. Nowadays it leaves many resources in modern CPUs unused. The reasons for this failure are explored in this section.

Every program consists of operations that are executed and data that is operated on. Therefore, the performance of a program is determined by how fast

[15]An introduction to this design pattern can be found in the standard reference [41].

the necessary operations are processed by the CPU and how fast the data is transferred to the CPU. Although the performance of both CPUs, measured as floating-point operations per seconds (FLOP/s), and main memory, measured as bandwidth (Byte/s), grew exponentially, the growth rate for the CPUs exceeded the main memory one, leading to significant performance differences nowadays. This problem is known as the memory wall, or gap, and it has been acknowledged since the 1990s. One of the most prominent description can be found in [93] and the subsection discussion in [74]. For example, a modern Intel server CPU can execute one addition instruction in 4 clock cycles, while fetching data from the main memory exceeds 100 cycles, illustrating the performance discrepancy. To fix this gap, both hardware and software has to be improved together.

One of the most significant hardware advances w.r.t. memory performance are caches, which are small memory components directly located on the CPU. Instead of directly transferring data from the main memory into the registers on the CPU, the data is first stored in a cache and then transferred into the registers. Accessing data stored in these caches is significantly faster than accessing the main memory, usually 1-2 cycles due to the proximity to the cores, which is especially beneficial if the same data is accessed repeatedly. Furthermore, the caches are organized in cache lines of fixed size, typically 64 bytes. Even if only a subset of one cache line is currently needed the complete cache line is transferred, favoring sequential data access, i.e. stepping consecutively through a dense array, while discouraging random access patterns. These benefits come at the cost significantly reduced sizes compared to the main memory. Typically, modern CPU have multiple levels of cache with different sizes and access latencies, where larger sizes correlate with lower bandwidth. For example, the Intel Skylake server CPUs have three caches, L1 with size of 32 KiB and 64 Byte/cycle transfer rate into the registers, L2 with 1 MiB and 32 Byte/cycle, and L3 with 10–28 MiB and 16 Byte/cycle.

To fully exploit the benefits from caching, software has to be adjusted to optimally reuse data and minimize data transfers between cache levels and main memory. For some algorithms this is difficult to achieve. Consider the standard procedure for a SpMxV in the CRS format in algorithm 2.3, where each entry $y[i]$ can be cached during the accumulation, but no value of A is required twice and therefore transferring them into a cache yields no benefits. The elements of x are read multiple times and should therefore be cached, although the access sequence of the elements depends on the sparsity pattern. Since the sparsity pattern is usually irregular, elements already in the cache are often replaced before they may be reused.

Algorithm 2.3: Sparse matrix vector multiplication

```
for $i = 1, \dots, N$ do
    $y[i] = 0$
    for idx = rowstart($A$, $i$), ..., rowstart($A$, $i + 1$) do
        $j$ = colindex($A$, idx)
        val = value($A$, idx)
        $y[i] = y[i] + \text{val} * x[j]$
```

The major driver for increasing CPU performance since the mid 2000s has been parallelization. Due to physical constraints, the CPU clock speed of one core could not be increased anymore, which is known as the end of Dennard's scaling. However, the transistor density kept growing exponentially, which led CPUs with multiple cores on one chip. Each of these cores can handle individual instructions concurrently. Since parallelization was necessary on supercomputers, efficient techniques for parallel sparse matrix vector multiplication or alternative discretization methods that directly incorporating parallelization, as touched on in chapter 4, have already been developed. Regardless of the parallelization approach, each core still needs to compute a sparse matrix vector multiplication, and therefore needs to efficiently utilize each single core.

The parallelization benefits are not restricted to adding new cores on one CPU, also the single core performance grew significantly due to parallelization approaches. One of the earliest models for this so-called instruction level parallelism is pipelining. This technique allows the CPU itself to split each instruction into predefined stages and execute multiple instruction at different stages in parallel. A more recent technology is vectorization, where a single operation is executed on multiple data at once within one instruction, for example adding two 4-sized vectors together in one instruction. The instructions are known as single instruction multiple data (SIMD) instructions. To benefit from vectorization, it is necessary to explicitly issue these special instructions. In some cases, modern compilers can automatically generate appropriate instructions, but this is still an active research area. Furthermore, oftentimes users are required to adjust the used algorithms and data structures such that vectorization becomes possible.

In the sparse matrix case, the native CSR format does not allow for automatic vectorization, mostly due to the non-trivial data access and variable loop sizes, which complicate the compiler's heuristics to decide if a loop should be vectorized or not. Therefore, specialized sparse matrix formats are required to regularize the matrix access pattern. This could be achieved, for example, by padding the values in one row with zeros such that every row has the same number of entries, or more advanced storage formats as discussed for example in [62]. Still, the resulting performance would is not optimal, since SIMD instructions work best on packed data, i.e. the used data elements are directly adjacent in memory, which is not given during the read of x. Instead, special gather instructions would be necessary to efficiently load the elements of x.

The discussion shows that the main building block for the solution of the discretized problem, the SpMxV, can not exploit modern hardware architecture to its fullest. In fact, the memory inefficiencies are so severe that the peak CPU performance can never be achieved, unless all data resides within a cache, due to the memory gap. This claim can be justified by considering the *roofline model*, which was first introduced in [92]. According to this model, the FLOP/s performance of a particular kernel is fully determined by the necessary data transfers and the number of computations. The model uses two key assumptions

1. the CPU can overlap the execution of data transfers and arithmetic operations,

2. the memory part can attain peak bandwidth BW w.r.t. the memory hierarchy the data resides in, and the compute-part can attain peak FLOP/s P.

With these assumptions, the total runtime of the kernel reduces to which part takes longer, i.e.

$$T = \max\left(\frac{\#\text{Byte}}{BW}, \frac{\#\text{FLOP}}{P}\right).$$

Consequently, the FLOP/s performance of a kernel is given by

$$\text{FLOP/s} = \frac{\#\text{FLOP}}{T} = \min\left(\frac{\#\text{FLOP}}{\#\text{Byte}} \cdot BW, P\right),$$

where AI = #FLOP/#Byte is known as the arithmetic intensity of the kernel. The roofline model therefore reduces the performance of a program to a single program dependent variable, and two machine dependent variables, highlighting its ease of use.

Applying this model to the SpMxV shows that the peak FLOP/s performance cannot be reached on modern CPUs due to a low arithmetic intensity. First, some assumptions on the sparse matrix format are necessary. The matrix has dimension N, which equals the number of DoFs, with nnz non-zero entries and the data type for the entries and indices uses 8 bytes. Now, revisiting algorithm 2.3 gives an estimate for the AI as

$$\text{AI} = \frac{\#\text{FLOP}}{\#\text{Byte}} = \frac{2 \cdot nnz}{8 \cdot \{(2+\alpha) \cdot nnz + N\}} \approx \frac{2}{8 \cdot (2+\alpha)}.$$

The parameter α measures the cache reuse for accessing x, where $\alpha = 1$ is the worst case without any cache reuse, and $\alpha = N/nnz$ is the best case, where only one element of x needs to be loaded for every row of A. Therefore, the arithmetic intensity is bounded by $1/12 < \text{AI} < 1/8$. Additionally, the more domain oriented measure DoF/s can also be derived, using the same assumptions, as

$$\begin{aligned}\text{DoF/s} = \frac{\#\text{DoF}}{\text{T}} &= \frac{N}{\max\left(\frac{\#\text{Byte}}{BW}, \frac{\#\text{FLOP}}{P}\right)} \\ &= \min\left(\frac{N}{8 \cdot (2+\alpha) \cdot nnz} \cdot BW, \frac{N}{2 \cdot nnz} \cdot P\right) \\ &= \frac{N}{8 \cdot (2+\alpha) \cdot nnz} \cdot BW \quad (\text{assuming}\, BW < P).\end{aligned}$$

Fig. 2.2 shows the estimated performance on one core of a modern Intel architecture, as described in A.1, for the SpMxV. Although for typical FEM applications the main memory bandwidth is the only relevant one, the bandwidths of each cache level are also presented. Using the main memory bandwidth, the SpMxV achieves only 0.3–0.5 GFLOP/s, which is less than 1% of the theoretical peak performance. Even in the best case, the data fits completely into the L1 cache, the best possible performance of the SpMxV is only ~1/3 of the peak performance. Additionally, Fig. 2.2 illustrates that efficiently operating on data from main memory requires a high number of arithmetic operations per byte to reach the peak performance. Half of the peak can only be achieved with an AI of ~8, while the full peak requires an AI of ~15. Thus, it becomes clear that simulation techniques relying on assembled matrices cannot attain any significant portion of the peak performance, and instead other approaches need to be pursued.

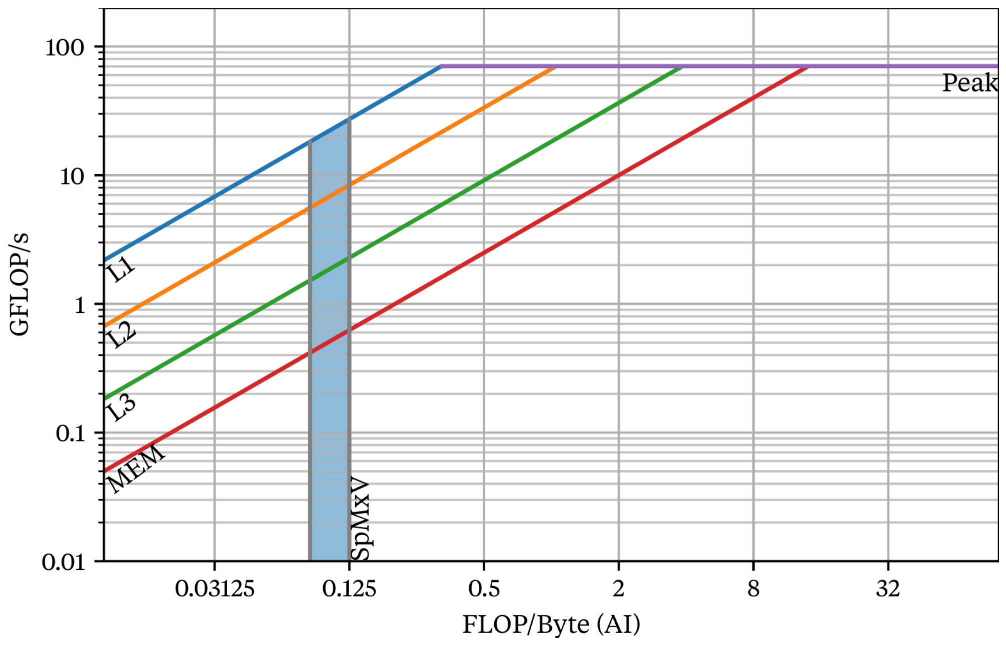

Figure 2.2: Roofline plot showing the maximal and minimal performance for the SpMxV for multiple memory levels on one core. In the most common case (MEM bound) the SpMxV achieves only 0.3–0.5 GFLOP/s and in the best case (L1 bound) 10–20 GFLOP/s are possible.

CHAPTER 3
Block-Structured Grids

ONE OF THE MOST LIMITING FACTORS of matrix-based FEM is the main memory throughput, since the operator application requires the loading large volumes of data for the matrix and nearly random access pattern due to the sparsity pattern of the matrix. Matrix-free methods can reduce these memory inefficiencies, see [76] for an application to higher order DG methods. However, for low order FEM they are still memory bound. This chapter introduces the concept of block-structured grids for low order FEM in order to increase the memory efficiency of these matrix-free methods. The viability of this approach can be seen from the use in existing numerical software, e.g. the Lattice-Boltzman simulation framework WALBERLA [38] and [47], in the context of hierarichal hybrid grid methods used by TERRA-NEO [46] and HyTeg [61], or the finite element toolkit FEAST [91]. In the following, at first the method is generally described, then issues arising from the handling of degrees of freedom associated with faces or edges are discussed, and afterwards optimization techniques using SIMD are presented. Finally, numerical examples are examined and possible future directions are discussed.

3.1 General Description

The *block-structured grids* approach is centered around the idea of virtually refining each grid element uniformly into multiple sub elements, referred to as *micro elements*. While the standard uniform refinement approaches add these micro elements to the grid structure as new elements, this is not the case for block-structured grids. Instead, the grid structure contains only the original elements, in the following called *macro elements*, and the iteration over micro elements happens during the handling of their associated macro element. For this work, the macro elements are always cubical elements, intervals in 1D, quadrilaterals in 2D, or hexahedrals in 3D. Nevertheless, this approach is also applicable to simplex elements, as seen in for example [61]. Fig. 3.1 illustrates the concept of a block-structured grid, where every macro element of the grid has been divided into 4×4 micro elements. Since cubical elements have a tensor product form, the numbering of micro elements also adapts this tensor product form, i.e. micro elements in $\mathbb{R}^d$ are indexed using d indices $[e_1, \dots, e_d]$, which are aliased in 2D or 3D with their common unit direction names leading to $[e_x, e_y]$ in 2D or $[e_x, e_y, e_z]$ in 3D.

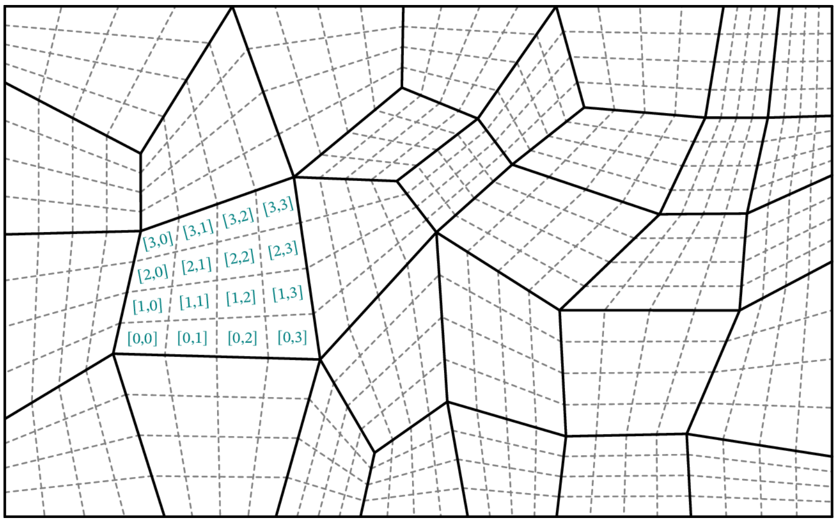

Figure 3.1: A block-structured quadrilateral grid. Additionally, the micro element numbering for one macro element is shown.

While uniform refinement works in powers of two, i.e. for one level of refinement each quadrilateral gets divided into four new quadrilateral in 2D, the local refinement for block-structured grids may be arbitrary but equal for each spatial direction. Within this work, the macro elements are usually subdivided into 10–100 micro element per spatial direction. In this range the benefits from block-structuring will become already visible, while allowing the usage of grids adapted to complex geometries. Higher local refinement in the order of 1000s or more is possible for certain application, especially within the geometrical multigrid context, or with higher order geometry support allowing for better geometry approximation by the macro elements.

The concept described above leads to the following definition. A triangulation $\mathcal{T}$ of a domain $\Omega \subset \mathbb{R}^d$ is called block-structured with local refinement size (synonym with *block size*) $k_1, k_2, \dots, k_d$, or simply k when $k_1 = k_2 = \cdots = k_d$, if

1. $\mathcal{T}$ can be divided into equally sized parts $\mathcal{T} = \bigcup \mathcal{T}_E$, with $\mathcal{T}_E \cap \mathcal{T}_{E'} = \emptyset$, $\#\mathcal{T}_E = \prod_{i=1}^{d} k_i$.

2. The union of all elements from one part $\mathcal{T}_E$, $E = \bigcup_{e \in \mathcal{T}_E} e \subset \mathbb{R}^d$, is simply connected with a C^1 diffeomorphism $T_E : \widehat{E} \mapsto E$, where $\widehat{E}$ is a reference element, and for each $e \in \mathcal{T}_E$ exists a map $T_{\hat{e}} : \hat{e} \mapsto \widehat{E}$ with a possi-

bly different reference element $\hat{e}$ such that $T_E \circ T_{\hat{e}} : \hat{e} \mapsto e$ is also a C^1 diffeomorphism.

The elements e of $\mathcal{T}$ are denoted as micro elements and the part-wise union E of these elements are called macro elements. Although the macro elements are defined as the union of micro elements, in practice the macro grid containing only macro elements $\mathcal{T}_H$ is created first and the micro elements are constructed by refining each macro element.

The following gives a short overview of the concrete implications of this approach, applied to the computation of the matrix-free operator application of the Poisson operator

$$y_i = (Au)_i = a(u, \phi_i) = \int_\Omega \nabla u \cdot \nabla \phi_i \, \mathrm{dx}.$$

In the block-structured case, the local kernel does not only iterate over the quadrature points and local basis, additionally it iterates over the micro elements. The pseudo code for the assembly of the vector y without using a block-structured grid was already given in algorithm 2.2 in the previous chapter, and the corresponding pseudo code with a block-structured grid is given by algorithm 3.1 with a slightly different notation. Since the global assembly is nearly the same as before, with some new optimization possibilities discussed later, only the local kernel is considered. Here $k \in \mathbb{N}$ is the number of micro element per direction, $T_{E[e_x,e_y]}$ denotes the geometry transformation of the reference element $\hat{e}$ onto the micro element with index $[e_x, e_y]$ within the macro element E. During the quadrature loop, $u[e_x, e_y, i]$ and $y[e_x, e_y, i]$ are accessing the i-th DoF associated with the micro element $[e_x, e_y]$ of the vector u and y respectively. Finally, n denotes the number of DoF per micro element.

3.1.1 Geometry Transformation

In the following, the special form of the geometry transformation is examined. For block-structured grids, the mapping $T_{E[e_1,\dots,e_d]} : \hat{e} \mapsto E[e_1, \dots, e_d]$ for a micro element $E[e_1, \dots, e_d]$ within a macro element E is defined as the composition of two transformations $T_{\hat{e}[e_1,\dots,e_d]} : \hat{e} \mapsto \widehat{E}$ and $T_E : \widehat{E} \mapsto E$, see Fig. 3.2. In the following k denotes the number of micro elements per direction and $1/k \cdot \hat{v}_0$ is the translation of the origin onto the origin of the micro element $[e_1, \dots, e_d]$ within the reference element $\widehat{E}$, e.g. in 2D $\hat{v}_0 = (e_x \ e_y)^T$. Now, the first transformation, mapping the reference element into the refined reference

Algorithm 3.1: Pseudo code for the local assembly on a block-structured grid.

```
Function: LocalOperatorApplication($E, u^E$)
    $y^E = 0$
    /* The order of the quadrature and micro element
       loops can be changed                                  */
    for $e_y = 1, \dots, k$ do
        for $e_x = 1, \dots, k$ do
            for $(\xi, \omega) \in$ QuadratureRule($E$, $order$) do
                weight = $\omega * |\det DT_{E[e_x,e_y]}(\xi)|$
                jit = $DT_{E[e_x,e_y]}(\xi)^{-T}$
                $\nabla u = \sum_{l=1}^{n} \nabla\hat{\phi}_i(\xi) u^E[e_x, e_y, i]$
                for $l = 1, \dots, n$ do
                    $y^E[e_x, e_y, i] =$
                      $y^E[e_x, e_y, i] + \text{weight} * \left(\text{jit} \cdot \nabla u \cdot \text{jit} \cdot \nabla\hat{\phi}_i(\xi)\right)$
```

through scaling and translation, is defined as

$$T_{\hat{e}[e_x,e_y]}(\hat{x}) = \frac{1}{k}(\mathbb{1} \cdot \hat{x} + \hat{v}_0).$$

The second transformation is the mapping from the reference element into the physical grid entity. An alternative approach would be to define $T_{E[e_1,\dots,e_d]}$ directly in terms of the corners of the micro elements as a linear combination of $\mathbb{Q}_1$ basis functions, similar to the definition of T_E. Since the computation of the required micro element corners is more complex than the composition, this approach is not used during this work.

Exploiting the structure of the geometry transformation improves the computation of geometrical quantities. For the computation of the local kernel, both the determinant and the transposed inverse of the Jacobian of the geometry transformation are needed. Applying the chain rule yields the following formula for the Jacobian of the transformation $T_{E[e_1,\dots,e_d]}$

$$\begin{aligned} D\,T_{E[e_1,\dots,e_d]}(\hat{x}) &= DT_{\hat{e}[e_1,\dots,e_d]}(\hat{x}) \cdot DT_{E[e_1,\dots,e_d]}(T_{\hat{e}[e_1,\dots,e_d]}(\hat{x})) \\ &= \frac{1}{k} DT_E(T_{\hat{e}[e_1,\dots,e_d]}(\hat{x})). \end{aligned}$$

Inspecting this formula shows that the determinant and the transposed in-

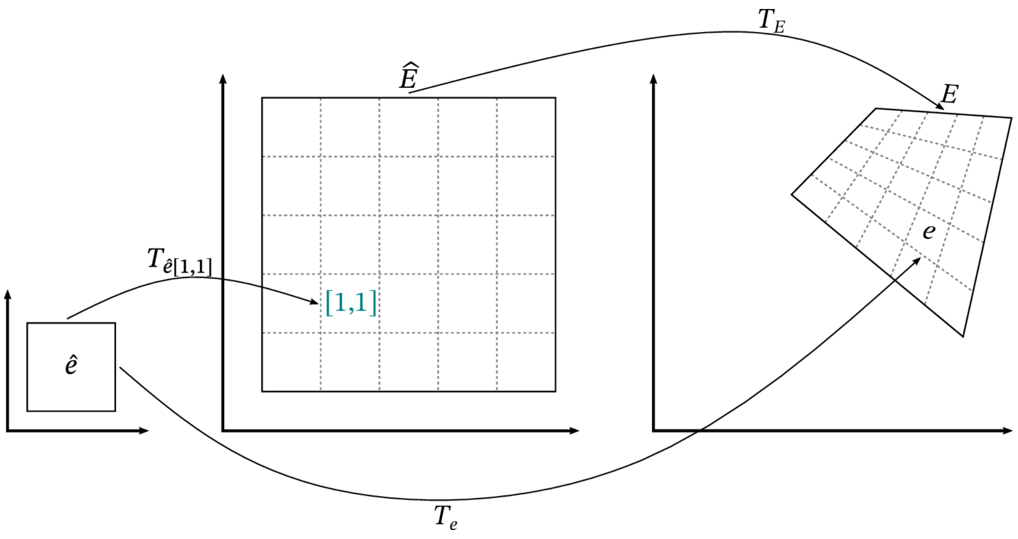

Figure 3.2: Composition of geometry transformation for a micro element in 2D. $T_{\hat{e}[1,1]}$ maps the reference elements of the micro element $\hat{e}$ into the reference element of the macro element $\widehat{E}$. T_E is the usual geometry transformation for the macro element, mapping its reference element $\widehat{E}$ into the physical macro element E. T_e is the composition of both transformations, which maps $\hat{e}$ into the physical coordinates of the micro element e.

verse simplify to

$$\begin{aligned} \det T_{E[e_1,\dots,e_d]}(\hat{x}) &= 1/k^d \det DT_E(T_{\hat{e}[e_1,\dots,e_d]}(\hat{x})) \quad \text{and} \\ DT^{-T}_{E[e_1,\dots,e_d]}(\hat{x}) &= k^d DT_E^{-T}(T_{\hat{e}[e_1,\dots,e_d]}(\hat{x})). \end{aligned}$$

Thus, only the evaluation of the macro element quantities at a transformed reference point is required. Usually, the evaluations of the quantities T_E^{-T} and $\det T_E$ are provided by the grid manager. However, these are not optimized for the evaluation in the context of block-structured grids. Specifically the repeated evaluation of either quantities within the same macro element can be optimized by precomputing terms depending only on the macro element vertices.

While the previous discussion of the geometry transformation holds for any multilinear macro element, in some cases further optimizations are possible. For instance, if the underlying macro elements are parallelepipeds, e.g. in a cartesian grid. Then, the geometry transformation of the macro element is an affine mapping and, as such, needs to be evaluated only one for each macro element. More precisely, in that case $T_E(\hat{x}) = [v_{2^j} - v_0]_{j=0,\dots,d-1} \cdot \hat{x} + v_0$, where

v_i, $i = 1, \dots, 2^d$, are the vertices of the macro element, and the micro element transformation simplifies to

$$T_{E[e_1,\dots,e_d]}(\hat{x}) = \frac{1}{k} \left[v_1 - v_0 \; \dots \; v_{2^{d-1}} - v_0 \right] \cdot (\hat{x} + \hat{v}_0) + v_0.$$

Computing the inverse Jacobian of this map is independent of the evaluation point and thus can be computed outside the micro element and quadrature loop. This reduces the computational cost of the local kernel significantly, since this allows additional precomputations, for example the scaling of the basis gradients, outside the micro element loop.

3.1.2 Local Data Structure

The block-structured kernel requires a certain local data layout, discussed in the following. As seen in the example pseudo code in algorithm 3.1, the local kernel does not access the global data directly, instead all data belonging to the DoF associated with a macro element are gathered into a local data structure and after the processing of the local kernel, the local data is scattered back into the global data structure. For a scalar $\mathbb{Q}_p$ finite element the local data structure is a flat array of size $(pk+1)^d$, where k is the number of micro elements per direction, and the local DoFs of the macro element are lexicographically ordered, according to their position in the refined reference element. Fig. 3.3 shows a schematic view of these DoFs for a $\mathbb{Q}_2$ basis function and $k = 3$. In the case of mixed or vector-valued finite elements, for example Taylor-Hood $\mathbb{Q}_p^d \times \mathbb{Q}_{p-1}$, each component of the finite element is represented as a flat array, the same way as described above, and all arrays are combined lexicographically into one flat array. Considering again at the Taylor-Hood element in 2D $\mathbb{Q}_2^2 \times \mathbb{Q}_1$, this means that the local data structure has size $2(2k+1)^2 + (k+1)^2$ in total. The first and second velocity component take up the first and second $(2k+1)^2$ DoFs, and the last $(k+1)^2$ DoFs belong to the pressure component.

During the handling of one macro element it is necessary to construct a mapping that assigns every local DoF of each micro element the corresponding index in the local data structure. For example, in Fig. 3.3 the index in the local data structure of the second DoF of the micro element $E[1,1]$, as well as of the eighth DoF of the micro element $E[0,1]$, is 18. More generally in 2D the local index of the $E[e_x, e_y]$ micro element DoF i of a $\mathbb{Q}_p$ element within a k-refined macro element is given by the mapping

$$\operatorname{idx}(e_x, e_y, i) = pe_x + i \bmod p + (pe_y + \lfloor i/p \rfloor)(kp + 1).$$

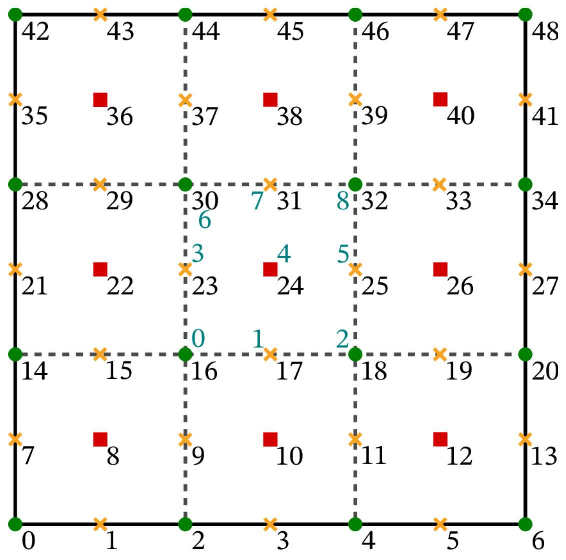

Figure 3.3: Local ordering of the macro element DoFs for a $\mathbb{Q}_2$ basis on a block-structured grid with $k = 3$. Additionally, the micro element local numbering for the element $E[1, 1]$ is shown in blue. The different marker denote the different types of subentities of the micro element subentities (codimension 0: red square, codimension 1: yellow x, codimension 2: green dot).

This computation gets significantly simpler if the micro element local DoFs are also numbered using a tensor index, similar to the micro elements. With this approach the index computation for a scalar finite element in any dimension simplifies to

$$\mathrm{idx}(e_1, \dots, e_d, i_1, \dots, i_d) = \sum_{j=1}^{d} (pe_j + i_j)(pk + 1)^{j-1},$$

removing the need for expensive integer modulo and division operations. For $\mathbb{Q}_p$ elements this can be applied, but for other element types different constructions are necessary. The local data structure can now be understood as a $2d$-dimensional array with non-unit stride, which will be clarified in chapter 5 with explicitly denoted strides. Due to the lexicographic ordering of the finite element components, the index maps for each component are simply offset by the overall size of the previous components.

Beyond simplifying the index computation, the local data layout implies an optimal loop ordering maximizing cache reuse. The index map above defines the first direction as the fastest changing one, and this should be represented in the iteration over the micro elements, as used in algorithm 3.1. In particular, using the index of this direction as the innermost micro element loop increases the cache locality compared to other loop orderings, since modern CPU prefetcher always load multiple adjacent bytes into the cache, and these bytes correspond to the DoFs of the next micro element w.r.t. the fastest changing direction. Another factor for cache locality is the order of the micro element and quadrature loops. If the local data structure does not fit fully into the L1 cache, it is advantageous to nests the iteration over the quadrature points within the micro element loops as in algorithm 3.1. In that case, it is sufficient that $2(k+1)$ DoFs fit into the L1 cache for each local vector to achieve minimal L1 misses during the computation of the next micro element $E[e_x+1, e_y]$, or in arbitrary dimensions $2(k+1)^{d-1}$ DoFs for $\mathbb{Q}_1$ elements. This effect becomes less striking for larger polynomial degrees p. For increasing p, the number of DoFs on the intersections of micro elements, which are the only DoFs that can be reused between micro elements, grows slower than the number of DoFs in the volume. Changing the micro element and quadrature loop would increase the number of L1 misses significantly if the whole local data structure does not fit into the L1 cache, but it would also allow for additional precomputations, making it a viable choice for small local refinements.

3.1.3 Global Data Structure

The efficient handling of the global-to-local gathering and scattering requires special care. Reducing the number of macro elements directly reduces the grid iteration overhead and increases the memory efficiency, since the local data is stored or loaded block wise instead of one-by-one for each micro element. Nevertheless, more adjustments are necessary to fully exploit the structured local data layout. The simple approach, which is the default in DUNE-PDELAB, queries for each local DoF its global index. This is defined by the global index of the entity the DoF belongs to and the local index w.r.t. the entity of the DoF. Fig. 3.4 illustrates this approach. Unfortunately, this disregards the structure of the local data. Since the macro element is refined uniformly, multiple DoFs belong to the same entity, e.g. in Fig. 3.4 there are 5 DoFs associated with each edge and 25 DoFs associated with the cell volume. The computation of the global index for these entities is therefore redundant in many instances.

Instead of this default approach, an improved version is used in this thesis that iterates over each subentity of the macro element, computes its global index once and then only needs to increment the global index for each local DoF of that subentity. This version does not only reduce the computation of redundant global indices of subentities, it also improves the memory access

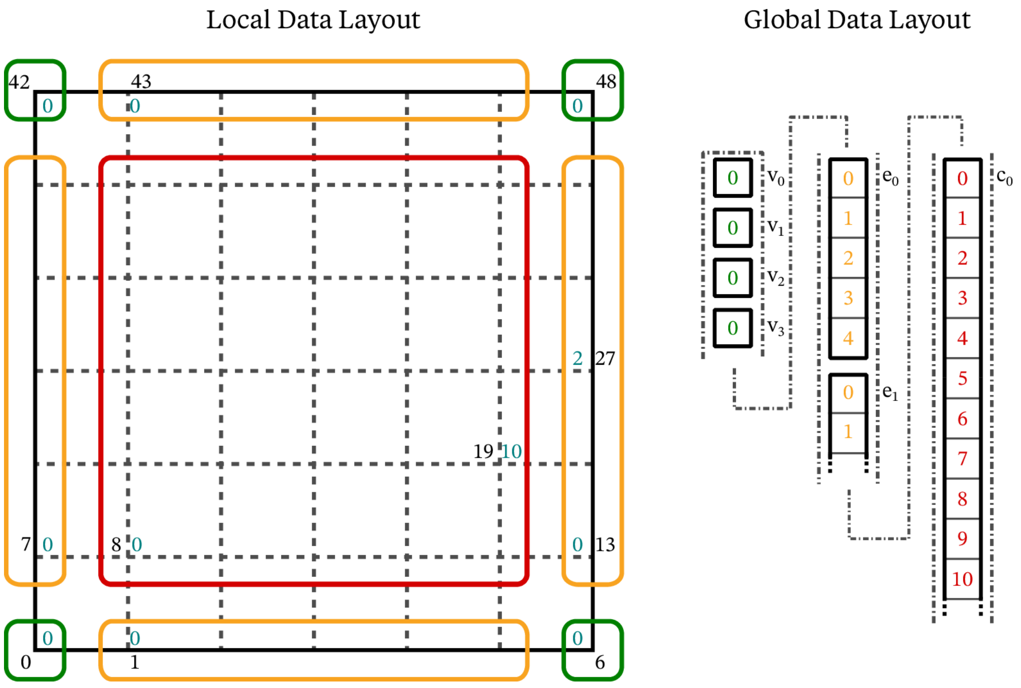

Figure 3.4: Connection between the local and global data layout. Besides the local DoF numbering in black, the local numbering w.r.t. the subentity containing the DoF is displayed in blue. v_i, e_i, and c_i denote the offset into the global vector for corresponding subentity.

into the global data. The global data is blocked according to the grid entities. Thus, iterating over the subentities of the macro element leads to consecutive access into the global data at the cost of accessing the local data not strictly consecutive. If the local data fits completely into the L1 or L2 cache, which is the case for small and medium block sizes, the extra cost is negligible. However, for large block sizes it might become noticeable. As long as the local data size does not exceed the higher level caches, the benefits of this approach become more pronounced for larger k and higher dimension, since the ratio of volume to surface DoFs increases.

3.2 Ensuring Consistency

An often encountered problem is the handling of multiple DoFs associated with entities of codimension $0 < c < d$. In the case of continuous basis functions, DoFs on these entities are shared between all elements containing the entity. The sharing implies that for each element the information on how these

DoFs restricted to itself relate to the DoFs restricted to all other elements needs to be available. For example the $\mathbb{Q}_3$ basis on quadrilaterals has DoFs associated with the points 1/3 and 2/3 along each edge, and thus adjacent elements need to have a consistent definition of the position of these points. This is a general problem in finite element assembly, and it has other applications besides block-structured grids, for instance in higher order methods. A thorough discussion can be found in [95].

One solution is to require additional mappings to correct for each subentity the local-to-global mapping of each element containing this entity. Let $g^s : \hat{I}^s \mapsto I$ be the reference index mapping of the subentity s, contained by the elements e_j, and $g^{e_j}|_s : \hat{I}^s \mapsto I$ the restriction of the reference mappings for the element e_j to the subentity s. Then, the additional mappings $C^s_{e_j}$ should satisfy

$$g^s \circ C^s_{e_j} \circ g^{e_j}|_s^{-1} = \mathbb{1}.$$

To clarify, first for each entity a reference local coordinate system is chosen. Then a mapping from the local coordinate system of each element containing that entity into the reference coordinate system is constructed, such that the DoFs on that entity are accessed according to the reference order.

Depending on the dimension of the subentity this approach is more or less straight forward. For 1-dimensional entities this can be easily realized by flipping the local order if necessary. First, define the reference local coordinate system for an edge (v_1, v_2) with global DoFs $n_1 < \cdots < n_l$ as $g^s(i) = n_i$ if $v_1 < v_2$. If the local coordinate system within the element does not match the reference coordinate system, then flip the local order of DoFs on an edge with $C^s_{e_j}(i) = l - i + 1$. Fig. 3.5 illustrates this example. In the case that $v_2 < v_1$ the ordering from g^s might also be reversed. For 2-dimensional entities this approach becomes significantly harder and is rarely done in numerical software.

In addition, this approach leads to unavoidable runtime overhead, as there are conflicting memory access patterns implied by the correction mapping. The assumption that for each shared entity all local coordinate systems w.r.t. the containing element coincide with each other, i.e. the mapping is always $C^s_{e_j} = Id$, would eliminate this overhead, since only one access pattern is actually used. A grid for which this assumption holds is called *consistently oriented*. Obviously, this assumption does not hold automatically for all grids, but any grid may be modified as a preprocessing step to make it consistently oriented. If the macro elements are simplices, then the local coordinate system for each entity is chosen such that the global indices of the local vertices are in ascending order. If the macro elements are cubes, then this preprocessing becomes

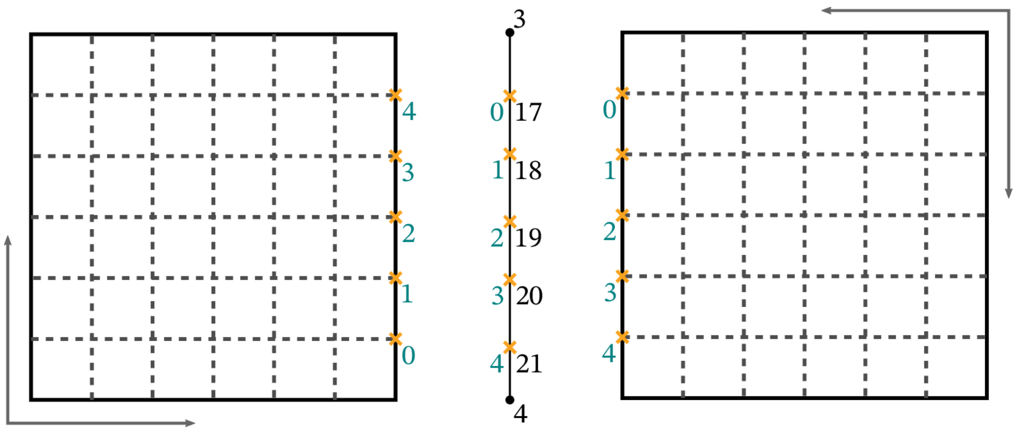

Figure 3.5: Two macro elements with different local numbering (blue) on the shared intersection. The global numbering of the DoFs is provided in black

more complicated. The deal.II library uses an approach described by Agelek et al. in [1] to achieve consistently ordered grids with cubes since the early 2000s, which has also been adopted by the Firedrake project published in [53] with an extension to parallel grids.

The algorithm described by Agelek et al. computes new local coordinate systems for each cube element in 2D or 3D such that its restriction to edges coincides for all elements sharing the edge. This works for every orientable 2D manifold, but may fail in 3D. In that case, after globally refining the grid once, the algorithm will always produce a grid with consistently oriented edges. As a downside, this algorithm does not guarantee that the local coordinate systems of a face coincides for all joined elements. In Fig. 3.6 the problem is exemplified, although the edges are consistently oriented, the coordinate systems of the faces are not aligned, since there are two possible face coordinate systems given any prescribed edge orientation.

The edge consistency algorithm is implemented as a standalone header-only C++ library, which is agnostic of any concrete grid implementation, and also accepts simplex elements. The API of the library is defined by the function

```
template<typename Vertex, typename Element>
bool orient_consistently(const std::vector<Vertex>& vertices,
                         std::vector<Element>& elements,
                         /* tag */)
```

where `Vertex` is a type modeling a point in $\mathbb{R}^d$ and `Element` is a random-access container holding the global indices of the element vertices. The `elements` parameter is an in-out parameter and holds the correct local numbering of the

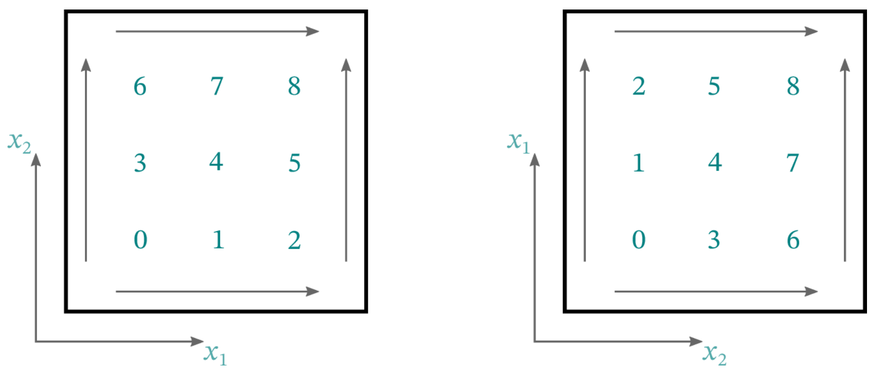

Figure 3.6: Two quadrilaterals with coinciding local coordinate systems on each edge (as indicated by the arrows), but with different local numbering of the volume DoFs.

element vertices such that the local coordinate system of each edge matches for all sharing elements. The `tag` parameter determines how the elements are interpreted, either as cube elements or simplex elements, and can be used as a runtime or compile-time parameter. If the `elements` parameter has been changed the return value is `true`, otherwise if the grid was already consistently oriented and no change was necessary it is `false`.

3.3 Efficient Vectorization

In the following, a vectorization approach with two variants that can utilize the block structure of the macro element is discussed. If the local basis and the element geometry type are fixed, then the computation at each DoF of an element does not differ between any two elements in terms of the needed operations, only in terms of the needed data. This similarity can be used to vectorize the computation at the same local DoFs across multiple elements, which is referred to as cross-element vectorization. The local data structure explained in section 3.1 is especially suited for this vectorization technique. Since the macro element DoFs are ordered lexicographically, the data associated with each vertex of a micro element is directly adjacent to the data of the vertex at the same position in the neighboring element. However, there is no alignment guarantee, as for an even number of micro elements per direction the number of DoFs per direction is odd. The ordering simplifies the loading and storing into/from the CPU SIMD registers further, because it can operate on contiguous data. It should be noted that this only holds for $\mathbb{Q}_1$ elements. An extension for higher order is discussed in the outlook of this chapter.

Recalling the example from algorithm 3.1, the listing 3.7 shows C++ code of the vectorized kernel. Instead of a scalar type, the underlying type of `detJ` or `grad_u[d]` are now SIMD vectors, which could be implemented using the C++ compiler intrinsics directly or designated C++ classes adding wrappers to these intrinsics. This work uses the wrappers provided by the vector class library (VCL) [40]. The SIMD vector has width w and $E[e_x + i, e_y]$ with $i = 0, \dots, w-1$ denotes the w micro element adjacent in the first direction beginning at $[e_x, e_y]$. The geometry quantities need to be evaluated for the same quadrature point in multiple micro elements, which is equivalent to evaluate T_E at multiple points, due to the composition of the geometry transformation, which can easily be vectorized. Accessing the local vectors `u`, `y` needs specific instructions depending on the SIMD vector type and are left out for brevity. It should be noted that neither `u` nor `y` needs to be read from or written to each iteration of the quadrature loop, loading `u` before the quadrature loop and updating `y` afterwards is sufficient.

```
for(ey = 0; ey < k; ++ey)
  for(ex = 0; ex < k; ex += w)
    for(auto [qp, qw]: quadrature){
      // the scalar type is replaced by a SIMD vector type
      // denoted by the indices i=0,...,w-1
      detJ = |det(DT_E[ex + i, ey](qp))|
      jit = DT_E[ex + i, y](qp)^-T
      grad_u = sum(j, grad_phi[j](qp) * u[ex + i, ey, j])
      // for Q_1 in 2D the basis size is 4
      for(j = 0; j < 4; ++j)
        y[ex+i, ey, j] += qw * detJ * (jit * grad_u * jit
                                         * grad_phi[j](qp))
    }
```

Figure 3.7: Pseudo code displaying the vectorized local assembly.

In practice, this approach does not yield the expected performance gains, if the workload per micro element is extremely low or if the local update is written to memory within every quadrature loop iteration. An explanation for this behavior are store forwarding stalls, caused by misalignment. Most modern processors are capable of forwarding a store directly to a following read of the same data, but if the data does not exactly overlap an extra penalty may occur[1]. This is the case in the approach described above; the SIMD vector containing the lower left DoF of w adjacent micro elements overlaps with the SIMD vector containing the lower right DoF of these elements with an offset of one

[1]See section 9.4 in the VCL manual and section 11.13 in Agner Fog's "The microarchitecture of Intel, AMD and VIA CPUs". Both can be found in `https://agner.org/optimize/`

scalar entry. Thus, the vector with the lower left DoFs must be written back into memory before the computation of the lower right DoFs can continue.

This problem is illustrated in listing 3.8, which is restricted to the innermost loops over the micro element DoFs. For clarification, the index map described in section 3.1 has been substituted, showing that the second iteration of the ix loop directly reads the last $w - 1$ entries written by the first iteration, causing the delayed execution. A partial mitigation is to change the micro element DoF loop order. When adding the newly computed result to the local data the currently innermost micro element DoF loop, which uses the same direction as the vectorized loop, is swapped with the outermost micro element DoF loop. This approach of the cross-element vectorization is called the *overlapping* variant. In listing 3.8 interchanging the ix and iy loop results in the overlapping variant. For complex local kernels this may already be sufficient, since the out-of-order execution may hide the write-back by other meaningful computations.

```
for(iy = 0; iy < 2; ++iy)
  for(ix = 0; ix < 2; ++ix){
    v = load(w, &y[(ey + iy) * (k + 1) + ex + ix])
    v += qw * detJ * jit * grad_u * jit * grad_phi[ix + 2 * iy](qp)
    store(v, w, &y[(ey + iy) * (k + 1) + ex + ix])
  }
```

Figure 3.8: Innermost loop of listing 3.7 with load and store instructions. Additionally, the tensor product structure of the $\mathbb{Q}_1$ basis has been exploited and the flat index has been substituted.

Another approach that does not suffer from overlapping stores and reads, is to combine the computation of DoFs adjacent w.r.t. the vectorization direction. Let rhs[ix] denote the right-hand side of the += operation above w.r.t. the innermost loop. The elements rhs[0][1..w-1] in the first iteration and rhs[1][0..w-2] in the second iteration update the same DoFs, which can be seen after unrolling. Therefore, the DoFs belonging to [(ey+iy)*(k+1)+ex+j] with j=1,...,w-1 can be updated in one step by combining both innermost iterations. As a consequence, only one load and one store to this memory segment is needed. Thus, this approach is called the *non-overlapping* variant of the cross-element vectorization.

A downside of this approach is the more elaborate carry handling. The innermost loop in listing 3.8 also updates the DoFs [(ey+iy)*(k+1)+ex+j] with j=0,w, which are currently left out, because these do not overlap between the two ix iterations and thus require special treatment. Since the 'rightmost' DoF, i.e. j=w, of the current w micro elements and the 'leftmost' DoF, i.e. j=0, of the next w elements are identical, both rhs[1][w-1] of the current w micro elements and rhs[0][0] of the next w micro elements contain the update for this DoF. Using this relation, carrying over rhs[1][w-1] into the next ex iteration, and adding it to rhs[0][0], computes the full update. Additionally, at the start of each ex loop the carry must be initialized to 0 and at the end of the loop the carry must be added to the last DoFs of that iteration, i.e. to the DoFs with indices ex=k, j=1. Fig. 3.9 illustrates this approach.

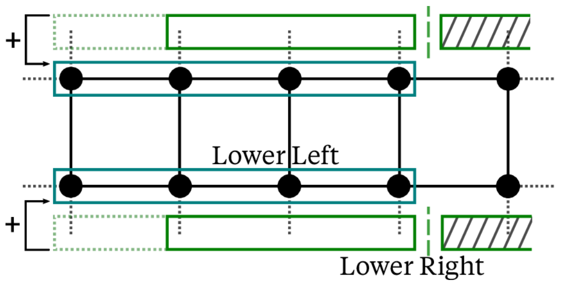

Figure 3.9: Visualization of the non-overlapping vectorization. The first three values of SIMD vector containing the lower right nodes of the elements (green) are added to the last three values of the SIMD vector with the lower left nodes (blue). The last entry of the green vector is used as a carry for the next four elements.

Listing 3.10 shows this approach applied to the previous listing 3.8. Here, rhs[j] are persistent across the inner micro element loop ex and permute shifts the entries of rhs[1] by 1 and sets rhs[1][0] to 0. The permute operation is also known as a shuffle. This approach is easily extendible to higher dimensions, since only the innermost loop over the micro elements and the micro element DoFs are contributing to the overlap problem. A possible disadvantage of this approach is that it may interfere with the C++ compiler optimizations and thus may be less effective for complex kernels, where FLOP reductions are more important.

```
for(iy = 0; iy < 2; ++iy){
  rhs[2 * iy] = qw * detJ * jit * grad_u * jit
                * grad_phi[2 * iy](qp) + {rhs[1][0], 0, ..., 0}
  rhs[2 * iy + 1] = qw * detJ * jit * grad_u * jit
                    * grad_phi[2 * iy + 1](qp)
  v = load(w, &y[(ey + iy) * (k + 1) + ex])
  v += rhs[2 * iy] + permute(rhs[2 * iy + 1])
  store(v, w, &y[(ey + iy) * (k + 1) + ex])
}
```

Figure 3.10: Innermost loop of 3.7 with non-overlapping computation of the two ix nodes.

The previous discussion assumed that the number of micro elements per direction k is a multiple of the SIMD vector width w, although this is quite limiting especially for vectorization based on the AVX-512 instruction set. If $k \neq c \cdot w$ with $c \in \mathbb{N}$, both vectorization methods can be applied by splitting the innermost micro element loop into two parts. The bulk part ranges from 0 to $\lfloor k/w \rfloor$ and is vectorized as described before, while the second part, the tail, ranges from $\lfloor k/w \rfloor$ to k and may be vectorized with a smaller SIMD vector width $w' < w$. For example if $k = 20$ and the largest vector width is $w = 8$, a vectorized handling of the tail with $w' = 4$ is possible. However, it may not be desirable to manually implement this tail vectorization, since it requires lots of code duplication, which are known to be error prone, while the benefits from vectorizing the tail are negligible compared to the vectorization of the bulk.

3.4 Benchmarks

In the following, benchmarks are considered to individually highlight each improvement from the block-structured. All examples compute the matrix-free Laplace operator application with constant coefficients

$$y_i = (Au)_i = \int_\Omega \nabla u \cdot \nabla \phi_i \, \mathrm{dx},$$

on a 2D quadrilateral grid with a total number of 1024×1024 elements, although the size of the macro elements varies. Depending on the example the geometry transformation may be affine or multilinear. Using this simple benchmark allows focusing on each aspect individually, reducing the noise from other optimizations. The hardware and software related information, as well as the measuring techniques used in the following discussions, is summarized in appendix A.1.

The first example considers the improvement block-structured grids provide without any optimizations within the local kernels. The next two examples examine the impact of the kernel optimizations on the local assembly, discussed earlier in this chapter, the efficient computation of the micro element geometry transformation and the cross-element vectorization. Both optimizations are specific to block-structured grids and cannot be easily applied to non block-structured grids. Lastly the runtime dependency on the macro element size is evaluated, specifically w.r.t. different loop orderings.

3.4.1 Global Assembly

The first example considers the improvement block-structured grids provide without any optimizations within the local kernels. Therefore, only the effect of reducing the gather and scatter overhead is measured, while disregarding most macro kernel optimizations. The adjustments to the local kernels are restricted to adding the micro element loops and the corresponding index mapping. The underlying grid is assumed to be axiparallel and equidistant, which reduces the FLOP count in the micro element local kernel to a minimum, since the inverse and the determinant of the geometry transformation's Jacobian is the same for each macro element and thus can be precomputed once for all macro elements. To simulate micro element local kernels with higher FLOP count, the quadrature is increased, ranging from order 2 up to order 60, which correlates to a 240-fold increase in the FLOP count. The size of the macro elements ranges between 1×1 and 1024×1024 micro elements.

As the plot 3.11 shows, the block-structured approach succeeds in shifting the focus of the execution towards the computation within the macro element kernels. Since the total runtime also decreases, see Fig. 3.12, the difference is not a result of an increase in the runtime of the macro element kernels. Instead, the previously described combination of specialized local and global data structure removes unnecessary operations, which do not contribute to the actual computation of the operator evaluation. This effect becomes less pronounced as the intensity of the micro element kernels increases, because in these cases the ratio is already high to begin with. Nevertheless, the total runtime benefits significantly from the block-structuring even for those examples, although a smaller local refinement is sufficient. In the lower quadrature order cases, the local kernel against total runtime ratio improves notably with medium local refinement, although it decreases for large refinements ($k \geq 512$) again. The decline is most likely due to increased cache misses, and could perhaps be prevented by tiling.

3.4.2 Efficient Geometry Computation

The first local assembly example investigates the computation of the Jacobian inverse and Jacobian determinant of the micro element geometry transformation. To estimate realistically the impact of the optimizations, the computation of the same Poisson operator application as before is considered, since it requires both quantities. Two types of macro element geometry transformations are considered, affine and multilinear transformations. For each ge-

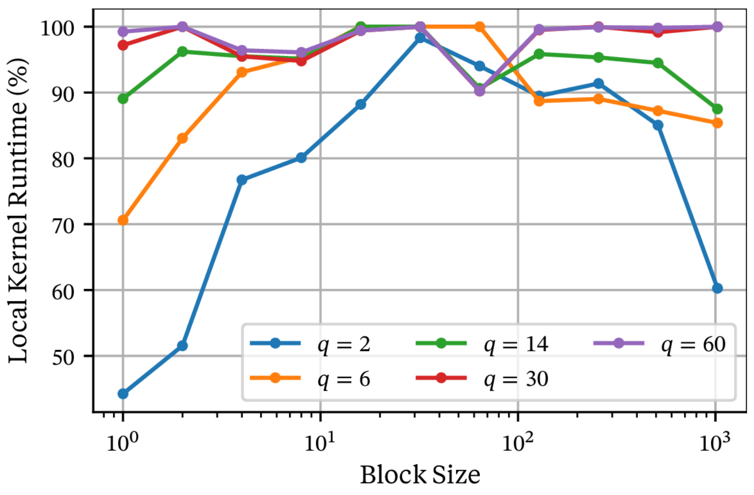

Figure 3.11: Ratio of effective computational time compared to total runtime for varying local refinement sizes. Increasing the block size significantly reduces the gather-scatter overhead, especially for local kernels with low amount of work. With block-structuring, around 90% of the runtime is spent within the local kernels, regardless of the kernels' intensity.

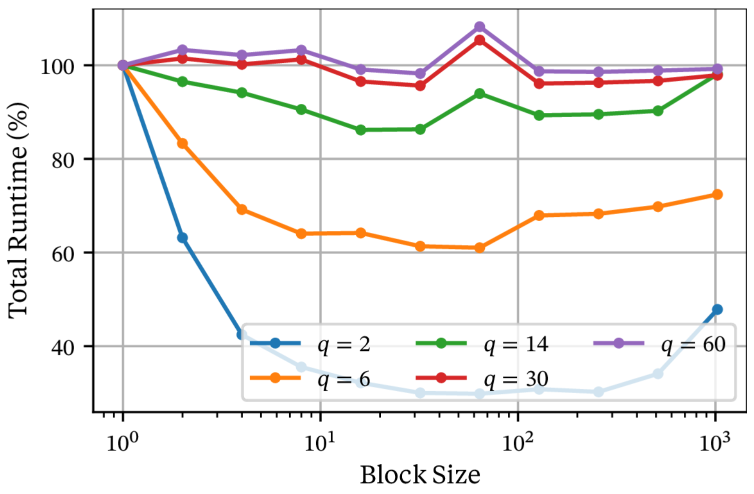

Figure 3.12: The total runtime of the operator application normalized against $k = 1$ instance. The overhead reduction from Fig. 3.11 directly leads to a total runtime reduction. Depending on the work intensity of the local kernel, a decrease up to 30% can be realized.

ometry transformation type, three local kernel versions are used, one without block-structuring and two with block-structuring enabled. The unoptimized block-structured kernel recomputes the geometry transformation for each micro element using the DUNE grid manager `YASPGrid` in the affine case and `UGGrid` in the multilinear case, while the optimized variant uses the precomputations outlined in 3.1.1. Besides precomputing the geometry transformations, no other precomputations, such as scaling the gradients, are performed during the optimized kernel in order to single out the effects of the optimized geometry transformation. The runtime within the macro element kernel is measured for each geometry transformation type and kernel version, and the block-structured versions are compared against the non block-structured one.

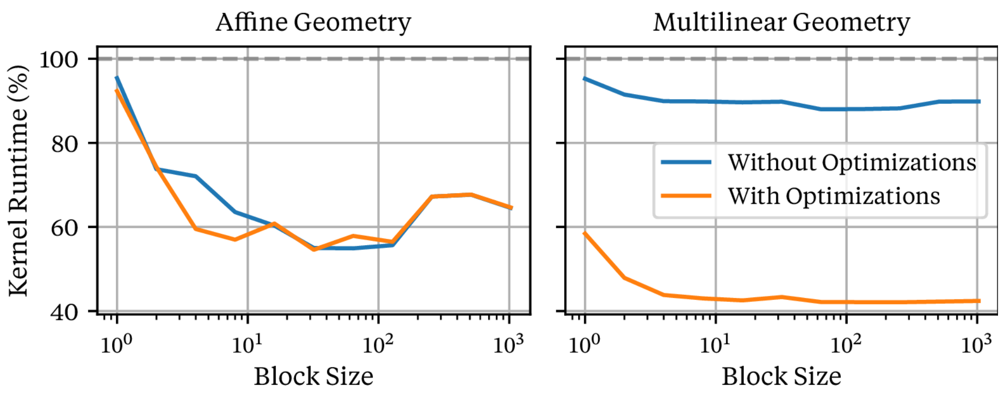

Figure 3.13: The local kernel runtime of block-structured kernels with and without optimizations on grids with affine and multilinear geometries, reported as percentage of the non block-structured kernel. In the affine case, the C++ compiler can realize the same optimizations as the optimized by hand kernel. For multilinear geometries, it is necessary to implement these optimizations manually.

Fig. 3.13 depicts the runtime reduction using the geometry optimizations. On the affine grid, the unoptimized version achieves nearly the same runtime as the optimized version. Both variants reduce the local kernel runtime to ~60% of the non block-structured kernel, where the best reduction is attained for medium-sized blocks, $k \in [10, 100]$. Due to the heavy template use within DUNE, the C++ compiler is able to determine that the computation of the geometry quantities does not depend on the quadrature point and can therefore be hoisted outside the quadrature and micro element loop. Concerning the multilinear grid, the C++ compiler is not capable to determine impactful precomputations anymore, and the runtime is still around ~90% of the non block-structured one. Instead, these precomputations need to be specified by hand to achieve a meaningful improvement. Even for low block sizes, the precomputations are useful, since the geometry transformation is split into a quadrature point dependent part and a quadrature point independent part, which is not recomputed in the optimized case. Compared to the affine grid, this leads to an even better reduction to ∼40% of the non block-structured local kernel runtime in the optimized case.

3.4.3 Efficient Vectorization

The second local assembly example considers the cross-element vectorization described in section 3.3. There are three versions of the macro element kernel: without cross-element vectorization, vectorized without the overlapping approach, and vectorized with the non-overlapping computation outlined in section 3.3. The C++ compiler's auto vectorization is enabled for the case without explicit cross-element vectorization and disabled if manual vectorization is used. Each vectorized kernel has two variants, one with the SIMD vector width 4 (AVX2) and one with 8 (AVX-512). The micro element geometries are axiparallel and equidistant, such that the transformation can be precomputed again, reducing the FLOP count within the kernel, while higher FLOP counts are simulated through higher quadrature orders as before. The runtime of the macro element kernels is measured and the speedups of the vectorized kernels compared to the non vectorized ones are shown in Fig. 3.14.

An ideal vectorization of a local kernel should result in speed-up as high as the SIMD width. In the AVX2 case, an even better speed-up is shown by the $q = 6, 14$ variants, while the other variants still achieve a speed-up higher than 3×. Presumably, the high speed-ups are caused by the inefficient auto vectorization used by the C++ compiler (g++ 9.3), which vectorizes roughly 2/3 of all FLOP for $q > 2$. In the AVX-512 case, the optimal speed-up is never attained, as the clock speed of the CPU during the execution of AVX-512 heavy code gets reduced, see appendix A.1. Incorporating this reduction results in an ideal speed-up of ∼7.3× for AVX-512, which only the $q = 6$ case using the non-overlapping approach can reach. Nevertheless, in most cases a speed-up between 5–7× are achieved across the board.

Although they reach different speed-ups, similarities between the AVX2 and AVX-512 performance are noticeable. In both cases, the non-overlapping varian increases the local kernel performance more consistently than the overlapping approach, especially for low and medium quadrature orders. The higher quadrature orders exhibit worse scaling most likely due to the high amount of temporaries necessary to store the precomputed gradients at each quadrature point. The degradation for high block sizes, $k \geq 512$, in the $q = 2$ case

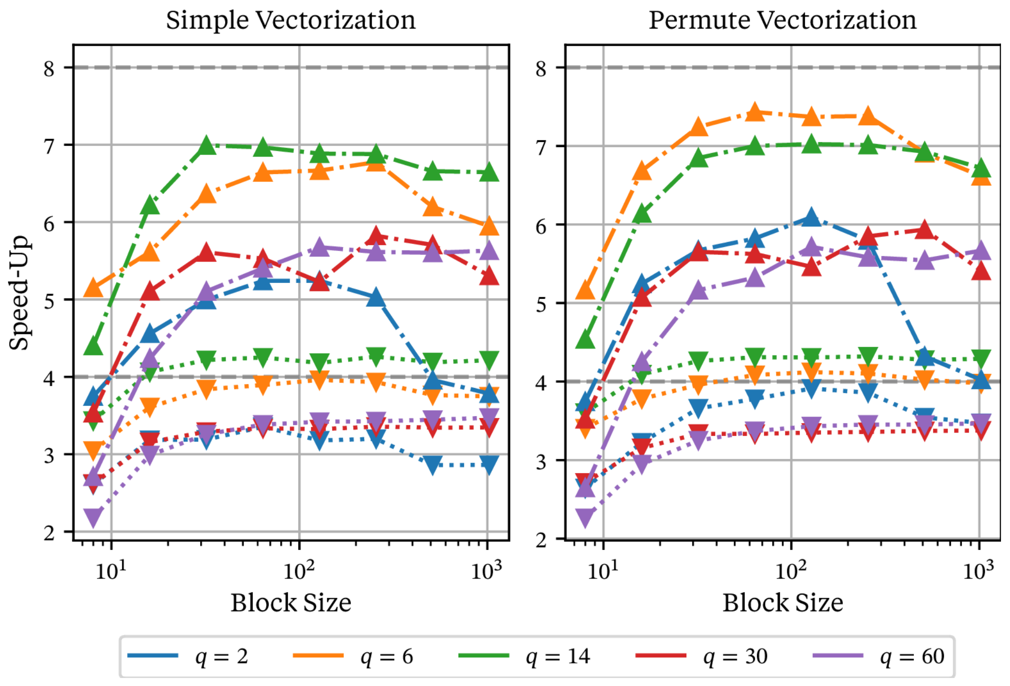

Figure 3.14: Runtime Speed-ups of the vectorized local kernels compared to unvectorized kernels for varying macro element sizes. The dotted lines correspond to AVX2 vectorized kernels (SIMD width 4) and the dashed-dotted lines to AVX-512 vectorized kernels (SIMD width 8). The explicitly vectorize kernel achieve nearly ideal speed-ups. In the AVX2 case most kernels achieve a runtime speed-up of 3–4×, and kernels with medium work intensity can achieve a nearly ideal speed-up of ~7× in the AVX-512 case. The differences between the overlapping variant (simple vectorization) and the non-overlapping variant are limited.

is most likely due to the missing tiling. Other Kernels are not significantly impacted by this, since the cache misses are hidden by the additional computations. Since the kernels with explicit vectorization achieve nearly optimal speed-ups despite allowing auto vectorization for the baseline kernels, this benchmark shows that the vectorization of the C++ compiler is not reliable for finite element kernels. Manual implementations are necessary to fully exploit the hardware's capabilities. Furthermore, the auto vectorization is only available for high quadrature orders, which are unlikely for low order methods.

It should be noted that the reported speed-ups only hold for the local kernels, not for the whole operator application. When considering the whole assembly, the gather-scatter overhead discussed earlier is not influenced by the vectorization, and therefore reduces the total speed-up. The overhead to local kernel ratios from Fig. 3.11 show that for kernels with low amount of work around 90%

of the runtime are spent within the local kernel. Combining this with the local kernel speed-up of ~5.5× results in an expected operator application speed-up of ~3.7× compared to a non vectorized block-structured operator application.

3.4.4 Operator Application

The last example combines the optimizations discussed before in order to examine their impact on the operator application as a whole. Additionally, a different loop ordering within the macro kernel is considered, which has been used during the early stages of this thesis, where the micro element loop is nested inside the quadrature loop instead of the other way around. As before, the total number of micro elements is fixed to 1024 per direction, and only the block size varies. An affine grid is used and the local kernels are always vectorized with the largest available SIMD width. This choice highlights the best case improvements gained by block-structuring, and more realistic settings are explored in chapter 6. The reversed loop ordering may have some advantages over the default ordering on affine grids, since in the reversed case the basis evaluation and subsequent scaling of the gradients happens only once for each quadrature point, while in the default case the scaled gradients need to be precomputed for every quadrature point, increasing the L1 cache usage. Fig. 3.15 depicts the floating point performance of the macro element kernels as percentage of the LINPACK peak, which has been missing so far. Finally, Fig. 3.16 shows the whole operator performance reported as DoF/s.

Concerning the FLOP/s performance in Fig. 3.15, the most notable behavior is that the LINPACK peak is surpassed on multiple occasions. Since the LINPACK benchmark uses a highly tuned implementation from Intel, it is unusual that for certain quadrature orders, $q = 6, 14$, kernels using the default loop ordering perform better. During the computation of the operator application on one micro element, the local result and the local coefficient can be kept in registers, thus most of the computations requires only little data movement between the registers and the L1 cache. For higher quadrature orders more precomputed gradients are necessary and therefore the data movement and L1 occupation increases again, which results in noticeably lower performance for the highest quadrature order. As a consequence, simulating local kernels with high work per micro element by increasing the quadrature order is not quite suitable. It clearly overestimates the efficiency of the macro element kernels, while it misses detrimental effects such as long dependency chains or instructions with high latency. Still, even if a low quadrature order is used a significant portion of the peak performance ~80–90% can be reached, assuming AVX-512

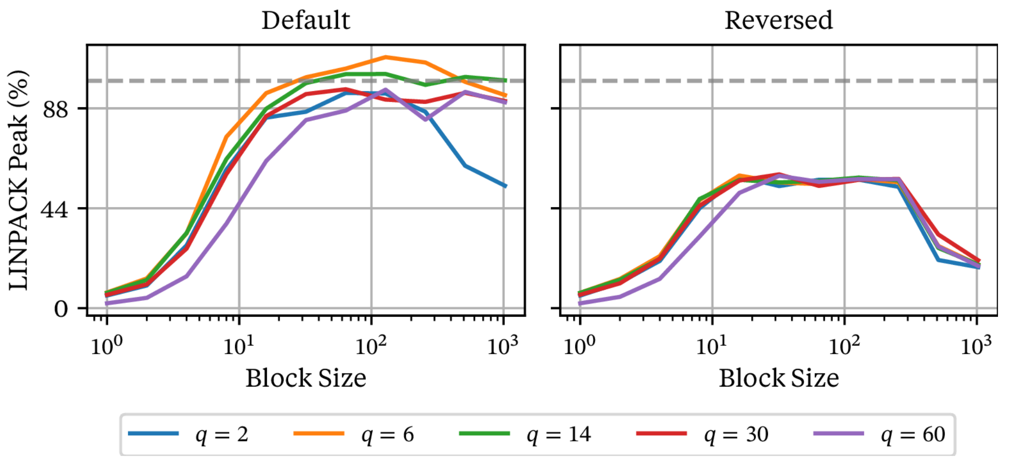

Figure 3.15: FLOP/s performance of the macro element kernels reported as % of LINPACK peak performance for the default loop ordering and the reversed loop ordering (i.e. quadrature loop before micro element loop). 100% of LINPACK peak on one core corresponds to ~45.5 GFLOP/s. Employing the block-structured grids approach leads to local kernel with a performance comparable to LINPACK. Already for small and medium block sized ($k = 16, 32$) the peak performance is achieved. The reversed loop ordering is consistently inferior to the default order.

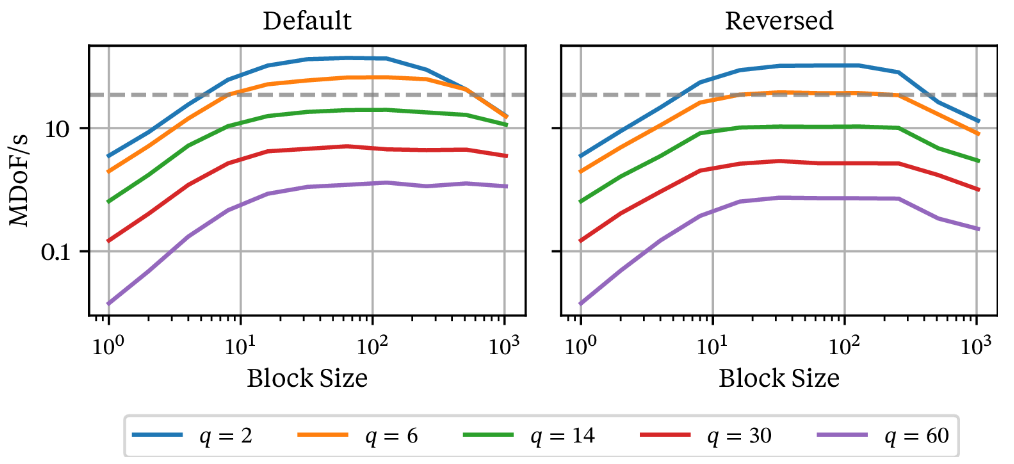

Figure 3.16: Operator application performance, measured in DoF/s for the default loop ordering and the reversed loop ordering (i.e. quadrature loop before micro element loop). The optimal performance for a matrix-based operator application (~35 MDoF/s) is depicted as the dashed, grayed-out line. For low work intensity kernels, the matrix-free operator application, using a block-structured grid, surpasses the performance of the corresponding matrix-based application by a factor of up to 4. Similar to the FLOP/s performance, reversing the loop order deteriorates the performance.

instructions can be used.

In contrast to the default loop ordering, the reversed loop order achieves less FLOP/s. Although some new precomputations are possible and the L1 cache use is reduced, reversing the loop order requires explicit reads and writes to the local vectors once per quadrature point for each micro element DoFs, which hinders the performance significantly.

The performance of the whole operator application, seen in Fig. 3.16, increases until a plateau is reached after $k = 16, 32$, which corresponds directly to both the increased FLOP/s performance and the overhead reduction. If the amount of work per macro element is low, the performance degenerates again for large block sizes ($k \geq 128$). This is most likely caused by an increased number of cache misses, which increases the overhead as observed in Fig. 3.11. In the other cases, the misses are masked in the other cases by more computations. The usage of tiling could circumvent cache misses, and would therefore represent an interesting option for further optimizations. Since the FLOP/s performance using the reversed loop order was one half of the default performance, only half of the DoF/s are reached. Therefore, the reversed order is not investigated further.

Naturally, the variant with the lowest quadrature order attains the highest performance. Interestingly, the performance in the other cases does not scale down as expected. With the chosen quadrature order the number of quadrature points is increased by a factor of 4, which should result in a performance decrease by the same factor. However, for medium quadrature orders a better down scaling of $\sim$2 can be noticed. The performance increase for higher quadrature order, as seen in Fig. 3.15, seems to compensate for the higher work per macro element, until the FLOP/s performance is saturated. The lowest quadrature order case achieves 100–130 MDoF/s for medium block sizes, which is significantly faster than the corresponding matrix-based operator application $\sim$35 MDoF/s, based the theoretical estimate from 2.4. Even the $q = 6$ case exceed the matrix-based performance, although only by a factor of 2 instead of 4. In the other case the matrix application is faster, due to the high local work. Nevertheless, the application of the block-structuring approach increases the matrix-free performance by a factor of 30–40, or even 90 for $q = 60$, compared to a non block-structured matrix-free application, i.e. $k = 1$.

3.4.5 Summary

The benchmarks show that block-structured grids increase significantly the performance of matrix-free operator applications for low order discretizations.

These benefits arise from two aspects, first the global assembly improvements, and secondly the newly possible optimizations within the local kernel. Block-structured grids use less grid elements compared to a non block-structured grid of the same size. Thus, the overhead from the grid element iteration gets reduced. Additionally, the overhead from gathering and scattering the global data into local data and vice versa is reduced by requiring less global index computations and accessing global data in a streaming fashion. The performance of the micro element wise computations are improved by the increased data locality, and optimizations which are not possible otherwise, like the efficient geometry transformation evaluation or the cross-element vectorization. As seen in the previous benchmarks, the highest improvements are achieved for medium block sizes, and no additional increases are found for larger block sizes. Tiling the micro element loops could alleviate this issue. Since this is currently not implemented, the focus lies on medium block sized $k \in [8, 128]$.

3.5 Outlook

Although block-structured girds show promising results already, there are still multiple areas left for further investigation. The use cases of the method presented here are currently restricted to multilinear cubical geometries and continuous Lagrange finite elements, although there are no conceptual barriers to apply this approach to other geometries or finite elements. Additionally, different local data structures could be explored to facilitate vectorization for local basis function with degree greater than 1. Since implementing local kernels for block-structured grids and exploiting the now possible vectorization opportunities is cumbersome and error prone, automating this process would be beneficial. The latter topic will be discussed in chapter 5, while the former will be briefly examined in the following.

The presented approach is targeted at cubical elements, because of their tensor product form, but there are several drawbacks compared to simplex elements. Creating meshes for complex geometries requires external meshing tools. The most common meshing tools, like Gmsh [43], offer superior support for simplex elements, especially in 3D, whereas fully unstructured, hexahedral meshes are usually not available, see the discussions in [15], [42]. Since the geometry transformation of a simplex macro element is always affine linear, it is independent of the micro element loop and can be precomputed, regardless of the macro grid. Therefore, simplices offer more flexibility and reduced computational cost. However, more complex loop nestings and index maps are needed, because these elements do not have a simple tensor product form.

Another approach to increase the flexibility w.r.t. complex geometries are higher order geometry transformations. In this case, the geometry transformation of the macro element uses a higher order basis than the $\mathbb{Q}_1$, which results in better approximation of curved boundaries without refining the macro grid. This allows the usage of coarser, but highly locally refined, macro elements and thus profiting from the efficient global gather and scatter operations even more. One side effect of this method is the increased computational complexity of the local kernel, since more FLOP are required to compute the geometry transformation and the higher basis degree increases the necessary quadrature order. Both of these disadvantages can be reduced by using approximations, as described in [32] and [12] for stencil local kernels. Furthermore, this could be already useful in reducing the computational cost for multilinear geometries.

Low order discontinuous Galerkin finite elements could be an interesting choice of basis functions for block-structured grids, since common optimization techniques, such as sum factorization, are less effective for low order functions, see [76] or [55]. Additionally, there are less vectorization opportunities for low order DG. Thus, the cross-element vectorization approach could be a good choice for these finite element types. Besides the iteration over the volume of the micro elements, the DG basis requires the iteration over the intersections of micro elements within one macro element, which could be implemented either in combination with the volume kernel or as a separate kernel. The computation of the intersection integrals between two macro elements needs a careful implementation of the index map, such that only micro elements are selected which intersect with the macro element boundary.

While the local data layout described in section 3.1 allowes an simple approach to cross-element vectorization technique, as explained in section 3.3, it also restricts it to $\mathbb{Q}_1$ finite elements. Switching to a different local data structure could lift this restriction. For example splitting the macro element into w equally shaped subdomains, as in Fig. 3.17, and interleaving the w DoFs of each subdomain at the same position, allows operating on these subdomains in lock-step. This also removes the need for the non-overlapping vectorization construction. The change in the local data structure implies adjustments in the global gather and scatter operation, since some DoFs on the boundary of these new subdomains are duplicated.

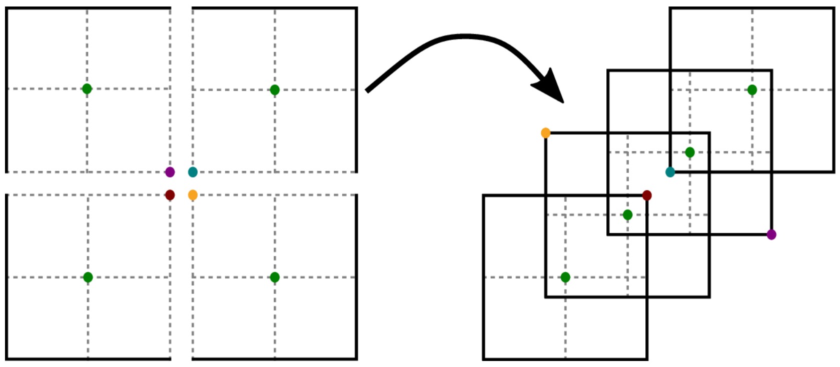

Figure 3.17: Vectorization-friendly data layout ($w = 4$). DoFs with the same color denote vectorized handling. The different colors of the same DoFs indicate the duplication of the corresponding DoF.

CHAPTER 4

Preconditioners for Block-Structured Grids

FOR THE EFFICIENT SOLUTION of a linear system, using a suitable preconditioner is essential, as discussed in section 2.2. Many of the most common and most efficient preconditioners for the systems stemming from a discretized weak formulation require the full assembly of the stiffness matrix, for example the AMG or ILU preconditioners. In some cases it is sufficient to be able to compute the full row for each DoF, e.g. for the Jacobi or Gauß-Seidel preconditioner. Neither works well out of the box with matrix-free approaches. Therefore, in this chapter, a non-overlapping domain decomposition method is adapted to allow its matrix-free application on block-structured grids. Although usually a means to enable parallelism over multiple cores, domain decomposition methods also act as powerful preconditioners, which can achieve mesh size independent convergence rates, and thus are useful for solving on a single core. Examples for other kinds of preconditioners for matrix-free methods, suitable for block-structured grids, are hierarchical multigrid methods, which are proven to be efficient for example in [13, 45].

4.1 Domain Decomposition Theory

This section introduces the algebraic formulation of a non-overlapping domain decomposition method, based on the description found in [88]. Other introductions to domain decomposition methods can be found for example in [6], which has a more theoretical approach and discusses more evolved non-overlapping methods, or in [31], which also discusses modern approach such as GenEO, and inspires some notation used here. As described in section 2.1.2, the discretization of a weak formulation leads to a linear system

$$A\mathbf{u} = \mathbf{f},$$

where A is the assembled stiffness matrix, $\mathbf{u}$ is the DoF vector, and $\mathbf{f}$ is the right-hand side. As an introduction, the decomposition into two subdomains is considered first in detail, with a generalization added later on.

4.1.1 Non-Overlapping Methods

Let the triangulation $\mathcal{T}$ for the domain Ω of the weak formulation be divided into two sub-triangulations $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$, each covering the subdomain $\overline{\Omega}_i = \bigcup_{\tau \in \mathcal{T}_i} \tau$, $i = 1, 2$, and denote with $\Gamma = \overline{\Omega}_1 \cap \overline{\Omega}_2 \setminus \partial\Omega$ the interior boundary of both subdomains. Using this decomposition, the set of DoFs $\mathcal{N}$ can be partitioned into two set of interior DoFs, $\mathcal{N}_1$ and $\mathcal{N}_2$, and a set of interface DoFs $\mathcal{N}_B$. The interior DoF sets $\mathcal{N}_i$ contain all DoFs corresponding exclusively to the subdomain Ω_i, which may include DoFs associated with the true boundary of Ω_i, while the interface set $\mathcal{N}_B$ contains DoF shared by both subdomains. By reordering the DoF vector according to the above partition reveals the following structure of the linear system

$$\begin{bmatrix} A_{11} & 0 & A_{1B} \\ 0 & A_{22} & A_{2B} \\ A_{B1} & A_{B2} & A_{BB} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{f}_B \end{bmatrix}.$$

This formulation allows for a factorization of A as

$$A = \begin{bmatrix} \mathbb{1} & 0 & 0 \\ 0 & \mathbb{1} & 0 \\ A_{B1}A_{11}^{-1} & A_{B2}A_{22}^{-1} & \mathbb{1} \end{bmatrix} \begin{bmatrix} A_{11} & 0 & A_{1B} \\ 0 & A_{22} & A_{2B} \\ 0 & 0 & S \end{bmatrix},$$

where S is the Schur complement of the block A_{BB} w.r.t. A defined by

$$S = A_{BB} - A_{B1}A_{11}^{-1}A_{1B} - A_{B2}A_{22}^{-1}A_{2B}.$$

Applying forward elimination to the system $A\mathbf{u} = \mathbf{f}$ results in

$$\begin{bmatrix} A_{11} & 0 & A_{1B} \\ 0 & A_{22} & A_{2B} \\ 0 & 0 & S \end{bmatrix} \mathbf{u} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{g} \end{bmatrix}, \text{ with } \mathbf{g} = \mathbf{f}_B - A_{B1}A_{11}^{-1}\mathbf{f}_1 - A_{B2}A_{22}^{-1}\mathbf{f}_2,$$

and then the unknown at the interface can be determined by solving the reduced Schur complement system

$$S\mathbf{u}_B = \mathbf{g}.$$

With these interface values, the backwards elimination yields the interior DoFs after solving the systems

$$\begin{aligned}A_{11}\mathbf{u}_1 &= \mathbf{f}_1 - A_{1B}\mathbf{u}_B,\\ A_{22}\mathbf{u}_2 &= \mathbf{f}_2 - A_{2B}\mathbf{u}_B.\end{aligned}$$

The advantage of using this factorization is the possibility of parallelizing the solution of the systems involving A_{11} and A_{22}, since these systems don't update shared DoFs. Although not directly noticeable, the Schur complement S and the right-hand side $\mathbf{g}$ can also be assembled by subdomain-local contributions. Since A_{BB} can be split into

$$A_{BB} = A^1_{BB} + A^2_{BB},$$

where A^i_{BB} are computed only w.r.t. the subdomain Ω_i, this leads to the following formulation of S:

$$S = S^1 + S^2, \text{ with } S^i = A^i_{BB} - A_{Bi}A_{ii}^{-1}A_{iB}, \; i = 1, 2,$$

where S^i are called local Schur complements. In the same manner $\mathbf{f}_B$ may be split into $\mathbf{f}_B = \mathbf{f}^1_B + \mathbf{f}^2_B$ and therefore $\mathbf{g}$ reduces to

$$\mathbf{g} = \mathbf{g}^1 + \mathbf{g}^2, \text{ with } \mathbf{g}^i = \mathbf{f}^i_B - A_{Bi}A_{ii}^{-1}\mathbf{f}_i.$$

The reduced Schur complement system can be solved in parallel by employing a Krylov-subspace method, since it requires only the application of S, which consists of two independent local Schur complement applications. In practice, the inverse of A_{ii}, necessary for the matrix-vector product with S and the forward and backward elimination, is not computed explicitly. Instead, a linear system with A_{ii} is solved, which is equivalent to solving the weak formulation on Ω_i with an additional zero Dirichlet condition on the interface Γ.

The procedure described above can be generalized to multiple subdomains. Similar as before, the triangulation is partitiond $\mathcal{T}$ into N parts $\mathcal{T}_i$ and non-overlapping subdomains are defined as $\Omega_i = \bigcup_{\tau\in\mathcal{T}_i}\tau$, $i = 1, \dots, N$ covering each part of the triangulation. Furthermore, the interface between all subdomains is denoted as $\Gamma = \bigcup_{i\neq j}\overline{\Omega}_i \cap \overline{\Omega}_j$. The DoFs $\mathcal{N}$ are grouped according to this partition by defining

$$\begin{aligned}\overline{\mathcal{N}}_i &= \{j \in \mathcal{N} : \operatorname{supp}(\phi_j) \cap \Omega_i \neq \emptyset\},\\ \mathcal{N}_B &= \bigcup_{i\neq j} \overline{\mathcal{N}}_i \cap \overline{\mathcal{N}}_j,\end{aligned}$$

where $(\phi_i)_{i\in\mathcal{N}}$ is the chosen finite element basis of the discretization. As a consequence $\overline{\mathcal{N}}_i$ contains all DoF associated with the subdomain $\overline{\Omega}_i$, and $\mathcal{N}_B$ all DoFs shared by two or more subdomains. Additionally, the DoFs exclusively owned by subdomain Ω_i are defined as

$$\mathcal{N}_i = \overline{\mathcal{N}}_i \setminus \mathcal{N}_B.$$

By reordering the DoFs, the same blocking as before of the stiffness matrix A and the vectors $\mathbf{u}$ and $\mathbf{f}$ can be achieved:

$$\begin{bmatrix} A_{11} & 0 & \cdots & 0 & A_{1B} \\ 0 & A_{22} & & & A_{2B} \\ \vdots & & \ddots & & \vdots \\ 0 & & & A_{NN} & A_{NB} \\ A_{B1} & A_{B2} & \cdots & A_{BN} & A_{BB} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{u}_N \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_N \\ \mathbf{f}_B \end{bmatrix},$$

where A_{ii} is the matrix block containing the coupling between ϕ_k, ϕ_l with $k, l \in \mathcal{N}_i$.

The definition of the reduced Schur complement system requires introducing additional interface DoF subsets $\mathcal{N}_{B_i}$ as

$$\mathcal{N}_{B_i} = \mathcal{N}_B \cap \overline{\mathcal{N}}_i,$$

which contain the DoFs belonging to the interface Γ and to the subdomain $\overline{\Omega}_i$. Let R_i be the boolean matrix restricting the index set $\mathcal{N}_B$ to $\mathcal{N}_{B_i}$, such that $\mathbf{u}_{B_i} = R_i \mathbf{u}_B$ consists of the DoFs on the interface of Ω_i. This results in the following definition of the Schur complement, using again the additive splitting as described above:

$$S = \sum_{i=1}^{N} R_i^T S_i R_i,$$
$$S_i = A_{B_iB_i} - A_{B_ii} A_{ii}^{-1} A_{iB_i},$$

where S_i are the local Schur complements. The right-hand side of the system is defined analogously as

$$\mathbf{g} = \sum_{i=1}^{N} R_i^T (\mathbf{f}_{B_i} - A_{B_ii} A_{ii}^{-1} f_i).$$

The following steps summarize the general approach for a non-overlapping domain decomposition method:

1. Compute right-hand side $\mathbf{g} = \sum_{i=1}^{N} \mathbf{g}_i$ by assembling the local contributions.

2. Solve the reduced Schur complement system $S\mathbf{u}_B = \mathbf{g}$ iteratively. This requires the application of S_i for each subdomain, which, in turn, requires the solution of the local Dirichlet problem

$$A_{ii}\mathbf{x} = A_{iB_i}\mathbf{y}.$$

3. Use backwards elimination to determine the interior DoF, again by solving a local Dirichlet problem

$$A_{ii}\mathbf{u}_i = \mathbf{f}_i - A_{iB_i}\mathbf{u}_{B_i}.$$

4.1.2 Neumann-Neumann Method

The matrix S has a better condition number ($\mathcal{O}(1/h)$) compared to the stiffness matrix A ($\mathcal{O}(1/h^2)$), see [14] or [94], but it is still dependent on the mesh size. Therefore, the Krylov solver for Schur complement system needs a preconditioner to achieve mesh independent convergence rates. A wide variety of preconditioner for this system have been investigated, some notable examples are iterative substructuring methods, see chapter 4.3 in [88], FETI [36] and FETI-DP [37] methods, or the BDD [72] and BDDC [29, 73] methods. Some of these methods are also discussed as part of the review [94]. The last two preconditioners are variants of the Neumann-Neumann method, which will be presented here in its simplest form.

To motivate the Neumann-Neumann method, the two subdomain case is examined again. On a structured grid with uniform mesh size and two equally sized subdomains, which are mirror images of each other, the local Schur complements S_1, S_2 are equal under the assumption that the weak formulation has constant coefficients. Therefore, $S_1 = S_2 = \frac{1}{2}S$ and consequently $B = \frac{1}{2}(S_1^{-1} + S_1^{-1})\frac{1}{2}$ is a good preconditioner for S since

$$BS = \frac{1}{2}(S_1^{-1} + S_2^{-1})\frac{1}{2}S = S^{-1}S = \mathbb{1}.$$

If the subdomains are not mirror images, B is still suitable since

$$BS = \frac{1}{2}(2 \cdot \mathbb{1} + S_1^{-1}S_2 + S_2^{-1}S_1)\frac{1}{2}$$

and the eigenvalues of $S_1^{-1}S_2$ and $S_2^{-1}S_1$ are uniformly bounded from above and below, see section 1.3.3 and 1.3.4 in [6].

Closely related to the application of S_i, the matrix-vector product with S_i^{-1} can be computed without assembling S_i or S_i^{-1}, by solving a local problem instead. To demonstrate that, let A_i be the local stiffness matrix for the subdomain Ω_i with blocking

$$A_i = \begin{bmatrix} A_{ii} & A_{iB_i} \\ A_{B_i i} & A_{B_i B_i} \end{bmatrix}.$$

Then, the inverse of A_i can be written as

$$A_i^{-1} = \begin{bmatrix} \mathbb{1} & -A_{ii}^{-1}A_{iB_i} \\ 0 & \mathbb{1} \end{bmatrix} \begin{bmatrix} A_{ii}^{-1} & 0 \\ 0 & S_i^{-1} \end{bmatrix} \begin{bmatrix} \mathbb{1} & 0 \\ -A_{B_i i}A_{ii}^{-1} & \mathbb{1} \end{bmatrix},$$

which shows that applying S_i^{-1} is equivalent to solving a local Neumann problem, i.e.

$$y = S_i^{-1}x \quad \Leftrightarrow \quad A_i \begin{bmatrix} v \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ x \end{bmatrix},$$

where the interior values v of the solution vector are discarded. Conceptually, solving the reduced Schur complement system with this preconditioner applies correction terms to the solutions of the two local Dirichlet problems that have the same trace on the interface in order to align the normal derivative of the solutions.

The Neumann-Neumann method may be easily extended to more subdomains. As s first step, it is necessary to define for each interface DoF the number of subdomains containing that DoF

$$\delta_l = \#\{j \,:\, 1 \le j \le N \text{ and } l \in \overline{\mathcal{N}}_j\}.$$

Afterwards, the following diagonal scaling matrices

$$(D_i)_{ll} = \frac{1}{\delta_l}, \quad \forall l \in \mathcal{N}_{B_i}$$

are constructed to achieve good convergence, similar to the scaling factor 1/2 used in the two subdomain case. These could also be defined as weight functions, if the PDE has varying coefficients. With these scaling matrices, the generalization of the two subdomain preconditioner is straight forward,

$$B_{NN} = \sum_{i=1}^{N} R_i^T D_i S_i^{-1} D_i R_i.$$

Again, the application of S_i^{-1} can be realized as the solution to a local Neumann problem.

However, unlike the two subdomain case, some Neumann problems are not uniquely solvable. This problem arises from so-called floating subdomains. In this case the subdomain does not touch the true Dirichlet boundary of the whole domain, which results in a local problem with pure Neumann conditions, implying that A_i has a non-trivial kernel. Thus, a unique solution does not exist. One approach to circumvent this difficulty is to use the pseudo-inverse of S_i, although this has a similar computational cost to directly inverting S_i, which should be avoided if possible. A cheaper approach is solving a regularized problem with $\tilde{A}_i = A_i + \alpha T$, where α is a suitable constant and T is a non-singular matrix. In this work, $T = \mathbb{1}$ is chosen for simplicity and α is experimentally chosen such that $\tilde{A}_i$ is close to A_i and the local solves converge resonably fast.

With the Neumann-Neumann preconditioner the convergence rate of the Krylov method is improved, but it is still not truly mesh independent. The condition number of the preconditioned system can be estimated by

$$\kappa(B_{NN}S) \le \frac{C}{H^2}\left(1 + \log\frac{H}{h}\right)^2,$$

where H is the upper bound for the diameter of the subdomains, which shows that by increasing the number of subdomains the convergence of the Krylov solver will deteriorate.

4.1.3 Two Level Methods

The dependency of the convergence on the subdomain size H is typical for one-level domain decomposition methods. Colloquially speaking, this is due to the fact that the solution at any point depends on the values of right-hand side on the whole domain and the boundary conditions. For example, the solution of a linear second order elliptic PDE can be written as

$$u(x) = \int_\Omega G(x,y)f(y)\,\mathrm{d}y,$$

with an appropriate Green's function, see e.g. [34], showing that the solution at a point $x \in \Omega_i$ depends additionally on information only available outside Ω_i. Through use of subdomain local solves, the solution update can only incorporate the information from directly adjacent subdomains during one step of the global solve. On a cartesian grid with N_d subdomains per direction, the information about the right-hand side in one subdomain requires about N_d iterations until it reaches any other subdomain, which implies that the Krylov solver needs also N_d iterations until convergence is possible. Recognizing that $N_d = H^{-1}$ underlines the initially stated dependency.

Correspondingly, a preconditioner for the reduced Schur complement system needs to add long range coupling between the subdomains to achieve convergence independent of both the element and subdomain size. A standard approach is a coarse grid correction. The necessary coarse grid $\mathcal{T}_0$ is defined by using the subdomains as coarse elements, i.e. $\mathcal{T}_0 = \bigcup_{i=1}^N \tau_i$ with $\tau_i = \bigcup_{\tau\in\mathcal{T}_i} \tau$. Suppose $\mathbf{u}^n$ is an approximate solution of $S\mathbf{u} = \mathbf{g}$ as part of an iterative procedure, then the update $\mathbf{u}^{n+1} = \mathbf{u}^n + \mathbf{e}^n$ would result in the exact solution if $\mathbf{e}^n$ solves

$$S\mathbf{e}^n = \mathbf{r}^n, \text{ with } \mathbf{r}^n = g - S\mathbf{u}^n.$$

This problem is just as difficult as the original problem, but the complexity can be significantly reduced by transferring the problem onto the coarse grid, i.e. solving for a correction $\mathbf{e}_0$ on the coarse grid

$$S_0\mathbf{e}_0 = \mathbf{r}_0,$$

where S_0 is the discretized Schur complement on $\mathcal{T}_0$ and $\mathbf{r}_0$ is the residual $\mathbf{r}^n$ restricted onto the coarse grid. Afterwards, the fine grid correction $\mathbf{e}^n$ is reconstructed by prolonging $\mathbf{e}_0$ onto the fine grid.

The matrix S_0 is the Schur complement of the coarse grid stiffness matrix A_0, which results from the discretization over $\mathcal{T}_0$. In the case of piece-wise linear or multilinear finite elements on $\mathcal{T}_0$, A_0 does not have any interior DoFs, and therefore S_0 and A_0 are equal, simplifying the construction of S_0 substantially. Due to this simple description of the coarse Schur complement, other finite elements on the coarse grid are not considered further. The coarse grid residual $\mathbf{r}_0$ is obtained by applying the restriction matrix R_0 to $\mathbf{r}^n$, where R_0 is the matrix representation of the operator $\mathcal{R}_0 : V(\Gamma) \mapsto V(\Gamma)^H$ and the finite element spaces $V(\Gamma)$ and $V(\Gamma)^H$ are defined as

$$\begin{aligned} V(\Gamma) &= \operatorname{span}\{\phi_i|_\Gamma : i \in \mathcal{N}_B\} \\ V(\Gamma)^H &= \operatorname{span}\{\phi_i^H|_\Gamma : i \in \mathcal{N}_0\} \subset V(\Gamma), \end{aligned}$$

with the Lagrange basis $\{\phi_i\}_{i\in\mathcal{N}_B}$ on the fine grid and $\{\phi_i^H\}_{i\in\mathcal{N}_0}$ on the coarse grid. Since the finite element spaces are nested, $\mathcal{R}_0$ is the identity on $V(\Gamma)$, i.e. $\mathcal{R}_0 v = v$ for all $v \in V(\Gamma)^H$, which leads to the following definition of R_0

$$\phi_i^H = \sum_{j\in\mathcal{N}_B} (R_0)_{ij}\phi_j = \sum_{j\in\mathcal{N}_B} \phi_i^H(x_j)\phi_j,$$

where x_j are the Lagrange nodes of $\{\phi_i\}_{i\in\mathcal{N}}$. The prolongation of the coarse grid correction is performed by applying the transpose of the restriction matrix, $\mathbf{e}^n = R_0^T\mathbf{e}_0$. Writing these steps compactly results in the coarse grid preconditioner

$$B_0 = R_0^T A_0^{-1} R_0.$$

Although employing the coarse grid preconditioner results in a reasonable correction of the current error, it cannot be used alone, since B_0 has a large null space, and instead it has to be used in conjunction with a fine grid preconditioner. Now, the final additive two-level Neumann-Neumann preconditioner is defined by

$$B = B_0 + B_{NN} = R_0^T A_0^{-1} R_0 + \sum_{i=1}^{N} R_i^T D_i S_i^{-1} D_i R_i.$$

It is also possible to define a multiplicative version of this preconditioner. Although the resulting preconditioner leads to a better condition number, it has a higher computational cost, and therefore is not investigated further. In [94], the condition number with additive preconditioner is given by

$$\kappa(BS) \le C\left(1 + \log\frac{H}{h}\right)^2,$$

which depends only on the ratio between the subdomain size and the element size, showing that grid independent convergence can be achieved as long as the ratio stays constant.

Simple Fine Grid Preconditioners

As discussed above, the coarse grid correction needs to be employed with a fine grid preconditioner to reduce the parts of the error corresponding to its null space. Besides the Neumann-Neumann preconditioner, other types can be used for the fine grid part. The discussed two-level preconditioner has the general structure

$$B = B_0 + B_F,$$

where B_0 is the coarse grid correction as before, and B_F is the preconditioner on the fine grid. The Neumann-Neumann method uses explicit knowledge of the stiffness matrix' structure to construct a preconditioner for the Schur complement system and employs elaborate local solves to compute approximations of the local solutions. Simpler preconditioners, without complex local solves or even the need for the Schur complement system, could replace the Neumann-Neumann preconditioner on the fine grid to reduce the work per macro element. The faster preconditione application comes usually at the cost of slower convergence.

In the same manner as for the Schur complement system, a two-level preconditioner can be applied to the original system $A\mathbf{u} = \mathbf{f}$, with suitable adjustments. On the coarse grid, the preconditioner is defined as $B_0 = \widetilde{R}_0^T A_0 \widetilde{R}_0$, where A_0 is the coarse grid discretization of A and $\widetilde{R}_0$ the extension of R_0 to the full fine grid basis. With a fitting DoF numbering $\widetilde{R}_0$ can be written as $\widetilde{R}_0 = [R_0 \ \widehat{R}_0]$. This shows that only the implementation of $\widetilde{R}_0$ is strictly necessary, as $R_0 v = [\mathbb{1} \ 0]\widetilde{R}_0[\mathbb{1} \ 0]^T v$, although it is beneficial to specialize R_0 to reduce the number of unnecessary operations. On the fine grid, iterative linear solvers, like the Jacobi method, the Gauß-Seidel method, or the SSOR method, are suitable preconditioners to correct the errors in the kernel of B_0, while being cheap to evaluate. Using these methods, the application of the preconditioner B_F is computed by solving $A\mathbf{v} = \mathbf{r}^n$ for a fixed number of iterations, where $\mathbf{r}^n$ is the current residual.

4.2 Application to Block-structured Grids

The block-structured grid approach, introduced in chapter 3, lends itself naturally to domain decomposition methods and two-level methods in general. Due to the construction based on macro elements and micro elements, the partition of the grid into subdomains is already available. Furthermore, the unrefined macro elements build a suitable coarse grid, necessary for the coarse grid correction. Since the local data for one macro element does not overlap with any other macro element, the non-overlapping domain decomposition methods are better suited than overlapping domain decomposition methods.

As discussed in section 4.1.1, the non-overlapping methods contain the following three steps:

1. Transforming $A\mathbf{u} = \mathbf{f}$ into a reduced Schur complement system.
2. Solve the reduced Schur complement system $S\mathbf{u}_B = \mathbf{g}$ with a preconditioned Krylov method.
3. Use the backwards elimination to compute the interior values.

Each of these steps can be executed without assembling any matrices on the fine grid. The matrices S_i and S_i^{-1} require the application of A_{ii}^{-1} or A_i^{-1} to a vector, which is realized by solving a local problem of Dirichlet or Neumann type respectively. By solving the local problems iteratively, it is not necessary to assemble A_{ii} or A_i, instead it suffices to compute the local application of these matrices. Additionally, A_{ii} can be applied by using the full local matrix A_i, since $A_{ii} = [\mathbb{1}\ 0]A_i[\mathbb{1}\ 0]^T$ and therefore

$$y = A_{ii}x \quad \Leftrightarrow \quad \begin{bmatrix} y \\ y' \end{bmatrix} = A_i \begin{bmatrix} x \\ 0 \end{bmatrix}.$$

In the same manner, the application of the local matrix blocks $A_{B_iB_i}$, A_{B_ii}, and A_{iB_i} can be reduced to applying A_i to a vector padded with zeros appropriately and restricting the result. Therefore, multiplying S or S^{-1} with a vector does not need any assembled matrices.

The preconditioner for step 2 requires a coarse grid correction, as discussed earlier, which is computed by directly solving the coarse problem. Since the coarse problem usually has orders of magnitude less DoFs, fully assembling the stiffness matrix A_0 and factorizing it once has a negligible cost compared to the repeated application of the fine grid preconditioner. If the coarse grid problem does become too large to solve directly, the system can also be solved using a matrix-based preconditioner like AMG. As is generally the case, the restriction and prolongation matrices are not formed explicitly. Instead, their application to a vector is computed through iterating over the macro elements and evaluating the local restriction or prolongation, where an additional scaling has to be introduced to account for multiple updates on DoFs shared between multiple macro elements. This scaling coincides exactly with the diagonal scaling matrices D_i^{-1} necessary for the Neumann-Neumann preconditioner. The local evaluation of the restriction or prolongation operator is the same as the standard evaluation on a structured grid, except that it only operates on the boundary DoFs.

Due to the simple choice of coarse grid correction, the handling of the local Neumann problems needs special care. As mentioned in 4.1.2, the system is undetermined. The regularization discussed there consists of solving the problem with $\widetilde{A}_i = A_i + \alpha\mathbb{1}$. A fitting choice for the parameter α could be $\alpha = ||A_i|| \cdot \gamma$ with an appropriate norm and a regularization parameter γ, which is fixed to $\gamma = 10^{-3}$. The choice allows reasonably fast convergence, while reducing the deviation from A_i, at least for the problems considered in this thesis. However, the computation of any norm of A_i is not trivial if the local problem is solved matrix-free, and therefore an approximation of the norm is necessary, which does not require a matrix assembly.

The approach taken in this work is to derive an upper bound for the infinity norm $||A||_\infty = \max_i \sum_j |A_{ij}|$, where the subdomain index i has been dropped for clarity. Let $h(u, v)$ be the micro-element local integration kernel appearing in the weak formulation of the PDE, then

$$\begin{aligned}\max_i \sum_j |A_i| &= \max_i \sum_j |\sum_{e\in\mathcal{T}_{ij}} \int_e h(\phi_i, \phi_j)\,\mathrm{dx}| \\ &\leq \max_i \underbrace{\sum_j \sum_{e\in\mathcal{T}_{ij}} \int_e |h(\phi_i, \phi_j)|\,\mathrm{dx}}_{=n_i}\end{aligned}$$

with $\mathcal{T}_{ij} = \{e \in \mathcal{T} \,:\, e \in \operatorname{supp}(\phi_i) \cap \operatorname{supp}(\phi_j)\}$. The vector $n = (n_i)_i$ can be computed similarly to the local stiffness matrix assembly, but instead of scattering the entries $|h(\phi_i, \phi_j)|$ into the local matrix they are added up row wise, which is easily implemented by exchanging the local matrix data structure. This requires the same number of operations as assembling the local matrix, but the memory usage is significantly reduced.

The previous discussion illustrates that the two-level Neumann-Neumann preconditioner can be implemented in a matrix-free fashion, as it does not require the assembly of any matrices, except the coarse grid stiffness matrix. Furthermore, most operations can be reduced to applying the macro-element local stiffness matrix. This is realized by padding or restricting the input to the local kernel. For example the application of $A_{B_iB_i}$ requires that all interior DoFs are set to zero before applying the local kernel, and during the solution of a local Neumann problem it is necessary to set the DoFs, which are part of the true Dirichlet boundary of the PDE, to zero, while keeping the DoFs on the interior boundary. Therefore, the implementation uses only few problem dependent kernels. These are bundled and passed into wrapper classes that offers an interface to the macro element local operations, such as solving a Dirichlet or Neumann problem locally, or applying the boundary-interior coupling matrix. This results in an easy-to-use and generic implementation of the preconditioner.

If a two-level method without non-overlapping domain decomposition is used, only solving the system $A\mathbf{u} = \mathbf{f}$ with a preconditioned Krylov method is necessary. The coarse grid correction works as before, except that the input and output vectors are not restricted to the interface DoFs anymore. Under certain circumstances, the iterative solvers mentioned previously can be applied in a matrix-free fashion, for example if the full matrix row for each DoF can be computed on-the-fly. Due to the non-overlapping local data structure of the macro element DoFs, the couplings with DoFs outside the current element are not available, and therefore the matrix row for a DoF on the macro element boundary cannot fully be assembled. Still, for the Jacobi method it is possible to apply it in a mostly matrix-free context, after a preprocessing step. This can be seen from examining one step of this method

$$x^{k+1} = x^k + D^{-1}(r - Ax^k),$$

where Ax^k can be computed matrix-free and the diagonal $D = \operatorname{diag}(A)$ can be precomputed similarily by recognizing that $D_{jj} = a(\phi_j, \phi_j)$. Since the computation and application of D has the same complexity, operation and memory wise, as the matrix-free evaluation of Ax^k, one step of the Jacobi preconditioner does not require the assembly of the stiffness matrix A at any time. Therefore, the two-level method with a Jacobi preconditioner on the fine grid is also implemented for block-structured grids.

4.3 Validation

Finally, in this section the convergence properties of the discussed preconditioner, the two-level Neumann-Neumann method and the two-level method with the Jacobi preconditioner on the fine grid, are examined. The example problem is a Poisson problem

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega, \\ u &= g \text{ on } \partial\Omega, \end{aligned}$$

where $u(x) = ||x||_2^2$ is chosen as the exact solution. The domain $\Omega = [0, 1]^2$ is discretized with equidistant quadrilaterals into N^2 macro-elements, which are, in turn, refined into k^2 micro-elements piece-wise $\mathbb{Q}_1$ functions are used for the finite element space. The resulting linear system is solved with the preconditioned BiCGStab Krylov method until a reduction of the residual norm of 10^{-10} is reached. The implementation of the coarse grid correction uses the direct solver SuperLU [68] for the coarse problem. During the Neumann-Neumann fine grid preconditioner application, the local solves also use the BiCGStab method, although without preconditioning. As all systems are symmetric, the CG method would be more efficient. However, as part of the generic implementation the BiCGStab method was chosen. Finally, all local kernels are generated, as discussed in the following section, only the driver is handwritten.

There are two interesting behaviors to inspect for these two preconditioners. The first one is to analyze the convergence behavior under fixed $H/h = k$ with decreasing mesh sizes, and the second one is to keep the mesh size h constant but increase the macro-element size H. In the first case, both preconditioner should yield constant convergence rates. In the second case, the condition number for the Neumann-Neumann method should grow as $\log^2(H/h) = \log^2 k$, which results in an expected increase in Krylov solver iterations of $\log k$. The necessary iterations of the two-level Jacobi method should increase at a faster rate, although there is no theoretical estimate. Fig. 4.1 displays both behaviors.

As expected, both preconditioners exhibit grid independent convergence rates for fixed block sizes (H/h), although the Jacobi preconditioner increases for small grid sizes before flattening for medium and larger grid sizes. With the Neumann-Neumann preconditioner, the solver converges in significantly fewer iterations than with the Jacobi preconditioner. However, the Neumann-Neumann advantage is revoked by its higher runtime per iteration. Applying the Neumann-Neumann fine grid preconditioner requires the solution of a local problem on each macro element, thereby increasing the runtime of the preconditioner application by the number of local solver iterations compared to one operator application. In the case $H/h = 16$, this results in an increased cost per runtime of $\sim 10\times$, which also grows for larger H/h, since the local solvers converge slower.

The required number of iterations increases for fixed fine mesh sizes (h) if the block size is increased. In the Jacobi preconditioner case a nearly linear dependency can be observed, indicating a condition number of $\mathcal{O}(H/h)$. The Neumann-Neumann preconditioner has a better theoretical estimate, which

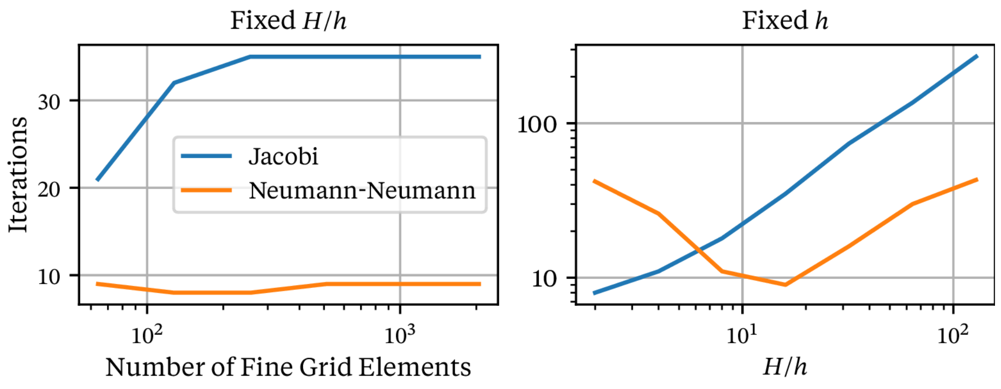

Figure 4.1: Number of iterations until error reduction of 10^{-10} for decreasing h with fixed $H/h = 16$ on the left side and for increasing H/h, or equivalently for increasing block size, with fixed $h = 1/1024$. With fixed H/h the convergence rate for both preconditioners is independent of the fine grid size h. For fixed h the Jacobi preconditioner shows a growth rate of H/h in the number of iterations, while the Neumann-Neumann preconditioner displays the expected $\log(H/h)$ growth after an initial decline.

can be observed after $k = 16$. Before that point, the number of iterations decreases. The reason for the decline is currently unclear, although it might be connected to the simple regularization used here. Nevertheless, this shows again that medium block sizes are more favorable than smaller sizes, similar to the discussion in 3.4.

4.4 Outlook

This chapter introduced two preconditioners for block-structured grids that do not require the assembly of the fine grid stiffness matrix and demonstrate good convergence properties. Nevertheless, the convergence rate of the two-level Neumann-Neumann preconditioner can be increased significantly by employing a different coarse space correction. For instance the balancing domain decomposition (BDD) method or the balancing domain decomposition with constraint (BDDC) method could be used, which have the additional benefit of restricting the local Neumann problem to a subspace on which A_i is invertible. Other non-overlapping domain decomposition methods, like the FETI or FETI-DP method, without the Neumann-Neumann preconditioner on the fine grid, promise comparable convergence and could thus be evaluated. Regardless of which preconditioner is examined more closely, the important property which the preconditioner needs to fulfil is that it can be applied matrix-free, after an optional preprocessing step.

Although the two-level Neumann-Neumann preconditioner shows better a convergence rate than the two-level Jacobi method, it is not the best choice. Since the Neumann-Neumann preconditioner requires expensive local solves of either Dirichlet or Neumann type, the time-to-solution is too high in most cases. Therefore, to use the Neumann-Neumann method efficiently, the local solve time needs to be reduced. If the PDE has constant coefficients and the macro-elements are equidistant and axiparallel, it is possible to precompute and factorize the local matrix A_i, since there are only a handful of different A_i. Otherwise, the local solver itself could be accelerated with a preconditioner, or the local problems are only solved approximately using this preconditioner. The latter approach requires, at first, recognizing that

$$A^{-1} = \begin{bmatrix} \mathbb{1} & -A_{II}^{-1}A_{IB} \\ 0 & \mathbb{1} \end{bmatrix} \begin{bmatrix} A_{II}^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} \mathbb{1} & 0 \\ -A_{BI}A_{II}^{-1} & \mathbb{1} \end{bmatrix},$$

where A_{II} is the block-diagonal matrix containing all local A_{ii}, and A_{BI}, A_{IB} are its coupling with the interface DoFs. With suitable preconditioners B_I for A_{II} and B_S for S it follows that

$$B = \begin{bmatrix} \mathbb{1} & -B_I A_{IB} \\ 0 & \mathbb{1} \end{bmatrix} \begin{bmatrix} B_I & 0 \\ 0 & B_S \end{bmatrix} \begin{bmatrix} \mathbb{1} & 0 \\ -A_{BI}B_I & 0 \end{bmatrix}$$

is a reasonable preconditioner for the system $A\mathbf{u} = \mathbf{f}$. The preconditioners B_I, B_S could now use inexact solvers for the local problems and comparable convergence can still be expected, which is shown for other types of preconditioners, such as the BDDC method [67] and the FETI-DP method [57].

CHAPTER 5 Generating Local Kernels

NUMERICAL SOFTWARE FRAMEWORKS should aim to support end users as much as possible to ease the development of their specific software. This means frameworks should strife to allow users to focus on implementing parts most vital to their research, while reducing the burden to develop the remaining necessary parts of the software. In particular, users should not be bothered with implementing advanced optimizations that are not part of their research area. Instead, the framework should be designed in such a way that users can get these benefits for minimal effort. One approach to automate the application of the block-structured grid optimization in the context of C++ frameworks, like DUNE or deal.II, could be to implement a wrapper for user defined local kernels, which handles the optimized geometry computations and local data structure. However, automating the vectorization described in 3.3 in this way is not viable. Another approach, used by FEniCS [2, 69] and Firedrake [80] among others, is to generate a local kernel, which already incorporates these optimizations, based on a simple description of the local integrals. The following chapter considers the code generation process for block-structured kernels and the application of optimizations for these local kernels.

5.1 Code Generation Framework

The code generation pipeline used in this thesis is implemented in the DUNE-CODEGEN framework, which was established in [54] and [55]. Based on input files using UFL to describe a weak formulation of a PDE a C++ class is generated, satisfying the `LocalOperator` concept used in DUNE-PDELAB. The chosen interface is suitable to realize code generation within well-defined bounds, which reduces the overall complexity. Additionally, interfacing with DUNE allows the reuse of advanced and thoroughly tested features like MPI parallelization or advanced grid managers. A downside to this approach is the tight coupling between the code generator and DUNE, and especially DUNE-PDELAB, which is often times only implicitly defined, whereas the whole program generation approach, as examined for example in the ExaStencils project [65, 85], allows greater control over all interfaces at the cost of increased generation complexity.

5.1.1 Intermediate Representation

To allow for greater control and additional optimizations, an intermediate representation (IR) is introduced between the weak formulation in UFL and the final C++ code. DUNE-CODEGEN uses the python package Loopy as IR, which is described in [58]. In this IR the computational kernel consists of a collection of loop domains, instructions, and arguments. This description follows the polyhedral model and Loopy is already bundled with kernel transformation exploiting that model, e.g. loop tiling, vectorization, or unrolling. Listing 5.1 shows a verbose implementation of a Loopy kernel for the computation of the matrix-free application of the Laplace operator in 1D, and the resulting Loopy representation of kernel is printed in listing 5.2 using Loopy's visualization capabilities.

In the following, the elements of a Loopy kernel are discussed in detail. Firstly, the loop domains are integer sets defined in the integer set library (ISL) syntax, which consists of variables, in the Loopy context commonly referred to as 'inames', and affine constraints on these variables. The ISL, internally used by Loopy, allows polyhedral computations on these set and, as such, is also used by the polyhedral optimizers of the Clang and GNU compilers [48, 87]. Lines 8–10 in listing 5.1 define three of these sets. The first one makes use of a parameterized bound, while the size of the other two is fixed. Therefore, four loops are introduced into the kernel, although there is no relationship between these inames at this state, not even between `i` and `j` except that both inames have the same constraints. Loopy determines the actual nesting and ordering of the loops using a polyhedral scheduling algorithm after the kernel's creation, depending on the instructions used in the kernel.

Secondly, instructions within the Loopy kernel are either symbolic expressions, which are defined using the python package pymbolic [59], or opaque strings representing C or C++ code snippets. In contrast to other symbolic mathematics packages, e.g. SymPy [75], pymbolic does not employ any automatic simplification heuristics. Instead, pymbolic preserves the expression exactly as stated. If requested, DUNE-CODEGEN can apply SymPy's simplification algorithm as a post-processing step. The lines 11–26 in listing 5.1 show the AST structure of the pymbolic expressions appearing in the computation of the operator application.

Each instruction carries additional dependency information used to define the schedule later on. The loop iname dependency defines implicitly the loop nesting. In listing 5.1 only one instruction (l.11) depends on the `e`, `q`, and `i` loop, while the other two (l.18 & l.22) depend only on the `e` loop. To prohibit adding

```python
from loopy import *
from pymbolic.primitives import *
import numpy as np

r, u, x, grad_u, h, qw, grad_phi, e, i, j, q, N = \
    variables("r u x grad_u h qw grad_phi e i j q N")

dom = ["[N] -> {[e]: 0 <= e < N}",
        "{[i, j]: 0 <= i,j < 2}",
        "{[q]: 0 <= q < 2}"]
ins = [Assignment(assignee=Subscript(r, (e, i)),
                  expression=Sum((Subscript(r, (e, i)),
                                  Product((Quotient(grad_u, h),
                                           Quotient(Subscript(grad_phi,
                                                              (i,)), h),
                                           Subscript(qw, (q,)), h)))),
                  within_inames=frozenset(("e", "i", "q"))),
        Assignment(assignee=h,
                  expression=Sum((Subscript(x, (Sum((e, 1)),)),
                                  Product((-1, Subscript(x, (e,)))))),
                  within_inames=frozenset(("e",))),
        Assignment(assignee=grad_u,
                  expression=Reduction("sum", inames=j,
                                       expr=Product((Subscript(grad_phi,
                                                               (j,)),
                                                     Subscript(u, (e, j))
                                                     ))),
                  within_inames=frozenset(("e",)))]
arg = [ValueArg("N", dtype=np.int64),
        GlobalArg("r", dtype=np.float64, shape=(N, 2), strides=(1, 1)),
        GlobalArg("u", dtype=np.float64, shape=(N, 2), strides=(1, 1)),
        GlobalArg("x", dtype=np.float64, shape=(N + 1,)),
        GlobalArg("grad_phi", dtype=np.float64, shape=(2,)),
        GlobalArg("qw", dtype=np.float64, shape=(2,)),
        TemporaryVariable("grad_u", dtype=np.float64),
        TemporaryVariable("h", dtype=np.float64)]
knl = make_kernel(dom, ins, arg)
```

Figure 5.1: Explicit instantiation of a Loopy kernel representing a 1D Poisson operator application assembly.

```
ARGUMENTS:
N: ValueArg, type: np:dtype('int64')
grad_phi: type: np:dtype('float64'), shape: (2), dim_tags: (N0:stride:1)
 ↪  aspace: global
qw: type: np:dtype('float64'), shape: (2), dim_tags: (N0:stride:1) aspace:
 ↪  global
r: type: np:dtype('float64'), shape: (N, 2), dim_tags: (stride:1,
 ↪  stride:1) aspace: global
u: type: np:dtype('float64'), shape: (N, 2), dim_tags: (stride:1,
 ↪  stride:1) aspace: global
x: type: np:dtype('float64'), shape: (N + 1), dim_tags: (N0:stride:1)
 ↪  aspace: global
───────────────────────
DOMAINS:
[N] -> { [e] : 0 <= e < N }
{ [i, j] : 0 <= i <= 1 and 0 <= j <= 1 }
{ [q] : 0 <= q <= 1 }
───────────────────────
TEMPORARIES:
acc_j: type: np:dtype('float64'), shape: () scope:private
grad_u: type: np:dtype('float64'), shape: () scope:private
h: type: np:dtype('float64'), shape: () scope:private
───────────────────────
INSTRUCTIONS:
   for e
↱    h = x[e + 1] + (-1)*x[e]  {id=insn_0}
│↱   grad_u = reduce(sum, [j], grad_phi[j]*u[e, j])  {id=insn_1}
││   for q, i
└└       r[e, i] = r[e, i] + (grad_u / h)*(grad_phi[i] / h)*qw[q]*h
 ↪ {id=insn}
   end e, q, i
───────────────────────
SCHEDULE:
   0: CALL KERNEL loopy_kernel(extra_args=[], extra_inames=[])
   1: for e
   2:     acc_j = 0  {id=insn_1_j_init}
   3:     h = x[e + 1] + (-1)*x[e]  {id=insn_0}
   4:     for j
   5:         acc_j = acc_j + grad_phi[j]*u[e, j]  {id=insn_1_j_update}
   6:     end j
   7:     grad_u = acc_j  {id=insn_1_0}
   8:     for q
   9:         for i
  10:             r[e, i] = r[e, i] + (grad_u / h)*(grad_phi[i] /
   ↪  h)*qw[q]*h  {id=insn}
  11:         end i
  12:     end q
  13: end e
  14: RETURN FROM KERNEL loopy_kernel
```

Figure 5.2: Loopy's internal representation of the previous kernel, after preprocessing and scheduling.

unnecessary loop dependencies to these two instructions, the `q` and `i` loop are nested within the `e` loop. Additionally, the dependencies between instructions may be explicitly stated, although Loopy's heuristic that each instruction depends on all write instructions to its read variables is usually sufficient.

Lastly, all variables appearing in the kernel need to be defined as arguments. The variables can represent temporary variables or global variables that are passed into the kernel as parameters. For multidimensional data the extent for each dimension has to be defined. Additionally, a stride for each dimension may be defined, if the canonical strides are not suitable. The combination of extent (e_i) and stride (s_i) completely defines an index map by

$$\mathrm{idx}(i_1, \dots, i_n) = \sum_{l=1}^{n} i_l \cdot s_l, \quad \text{with } i_l \in \{0, \dots, e_i - 1\}.$$

In listing 5.1, a non-unit stride is explicitly stated for the `r` an `u` variables, both of which model global vectors vector. These are indexed by the current element and the local DoF of that element, and using a non default stride for them results the index map $[e, i] \mapsto e + i$, which maps each local DoF to the corresponding global DoF.

5.1.2 UFL to Intermediate Representation

The Loopy kernel, as defined above, is created by transforming the UFL AST into a pymbolic expression using a visitor pattern. The resulting pymbolic expression, called accumulation expression, represents the local contribution of that kernel, e.g. the local residual or the local stiffness matrix, and it is cached as a Loopy instruction. If a system of PDEs is considered, the accumulation expression for each component is derived separately. Each UFL type is mapped onto appropriate pymbolic types, using the algorithms provided in the UFL package. During the tree traversal some UFL nodes are mapped to precomputed temporary variables, most notably the `Coefficient` node, representing the evaluation of the coefficient function or gradient, and geometry nodes like `SpatialCoordinate`, `JacobianDeterminant`, or `JacobianInverse`. Additionally, simple constant folding and zero elimination is automatically applied during the traversal. The Loopy instructions for the computation of these temporaries are cached, alongside with temporary variable and loop domain definitions encountered throughout the course of the AST traversal. In the final kernel creation step, these objects, together with the final accumulation instructions, are extracted from the cache and used to create the Loopy kernel.

The visitor used in the AST traversal within DUNE-CODEGEN can be specialized for block-structured grids or sum-factorized kernels, through the mixin pattern, which is discussed for block-structured in the next section. This is achieved by splitting the visitor into the following five parts, each one handling conceptually similar UFL nodes:

Geometry handles nodes concerning geometric quantities like the Jacobian determinant or the face outer normal.

Basis handles the basis and coefficient evaluation.

Quadrature handles the quadrature rule evaluation.

Accumulation although not connected to any UFL node directly this handles the creation of the accumulation instruction based on its pymbolic expression.

Auxiliary handles nodes not covered by the other groups, which are mostly algebraic or indexing nodes.

For each of these parts a default mixin class exists, and the first four parts can be specialized by providing a custom mixin class. The specializations enable certain optimizations during the AST transformation. For instance, the default geometry mixin exists in multiple variants, simplifying the computation of geometric quantities on axiparallel or equidistant grids. Currently, the block-structured and sum-factorized specializations are mutually exclusive, although it should be noted that sum-factorization of micro element kernels is conceptually possible on block-structured grids. The specialized mixin classes can be chosen by providing the according options in an additional `ini` file, otherwise the default mixins are used.

The overall generation process is summarized in Fig. 5.3. A specialized visitor transforms the UFL AST into a pymbolic AST symbolizing the accumulation instruction, which is added to the Loopy kernel alongside previously cached instructions. Once the kernel is created, certain transformations are applied, which consists of optimizations depending on type of the tree visitor and the user specification. Using the existing code generation capabilities of Loopy a C++ function is generated and fitted into a DUNE-PDELAB `LocalOperator` class. For a complete program a handwritten main file, also called driver, using the DUNE framework needs to be supplied. In simple test cases the code generator is capable to automatically provide the driver, but for more

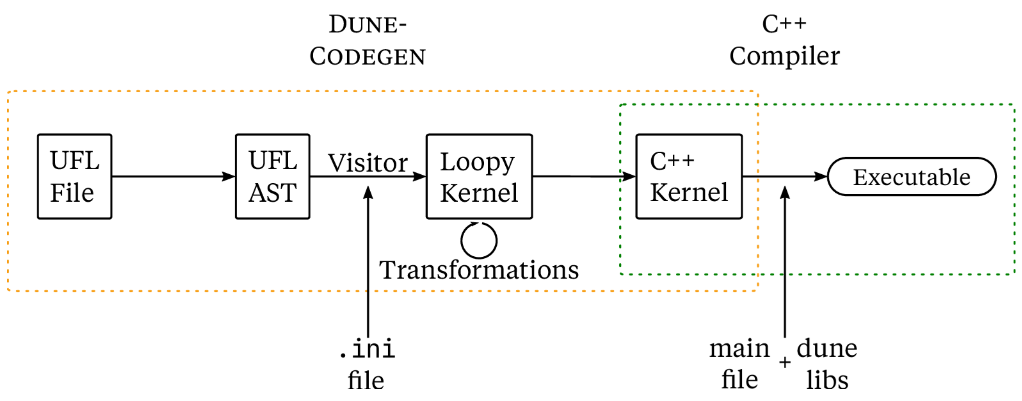

Figure 5.3: Schematic view of the DUNE-CODEGEN generation pipeline.

evolved applications additional user input is necessary. Lastly, the C++ compiler of choice, supporting at least C++–17, compiles the driver and `LocalOperator` class together with the DUNE libraries into the final executable. Thereby, the user effort for creating local kernels is significantly reduced.

5.2 Support for Block-Structured Grids

The following section outlines the handling of block-structured grids within DUNE-CODEGEN, detailing the mixin approach outlined in the previous section. With only minor adjustments, minimal support as described in section 3.1 can be achieved. It suffices to add the micro element loops, change the local data structure and the corresponding index mappings for basic support. Together with the optimized geometry computations from section 3.1.1 these changes are implemented as specialized mixin classes, while the vectorization mentioned in section 3.3 is realized as a Loopy kernel transformation, discussed in section 5.3.1. All four mixin classes, geometry, basis, accumulation, and quadrature, need to be specialized in order to implement the features that are crucial for block-structured grids. The specialized mixins are controlled by flags within the `.ini` file. Fig. 5.4 shows the domains and instructions of a finished Loopy kernel, without further optimizations, for the residual computation of the Poisson problem with $f = -4$ on a 2D equidistant grid with macro element size $k = 16$.

```
...
───────────────────────
DOMAINS:
{ [i_4_x, i_4_y] : 0 <= i_4_x <= 1 and 0 <= i_4_y <= 1 }
{ [q] : 0 <= q <= 3 }
{ [idim0] : 0 <= idim0 <= 1 }
{ [i_5_x, i_5_y] : 0 <= i_5_x <= 1 and 0 <= i_5_y <= 1 }
{ [e_x, e_y] : 0 <= e_x <= 15 and 0 <= e_y <= 15 }
{ [i_6_x, i_6_y] : 0 <= i_6_x <= 1 and 0 <= i_6_y <= 1 }
{ [i_7_x, i_7_y] : 0 <= i_7_x <= 1 and 0 <= i_7_y <= 1 }
───────────────────────
INSTRUCTIONS:
     for e_x, e_y, i_6_x, i_6_y
↱            r_lfsv_loc[i_6_x, i_6_y] = 0  {id=insn_0007}
│↱           x_lfsv_loc[i_6_x, i_6_y] = x_lfsv_alias[e_x, e_y, i_6_x,
 ↪  i_6_y]  {id=insn_0005}
││       end i_6_x, i_6_y
││       for q, idim0
││↱          acc_i_5_x_i_5_y = 0  {id=insn_0006_i_5_x_i_5_y_init}
│││          for i_5_y, i_5_x
│├└↱             acc_i_5_x_i_5_y = acc_i_5_x_i_5_y + x_lfsv_loc[i_5_x,
 ↪  i_5_y]*js_BCG1[q, i_5_x + 2*i_5_y, 0, idim0]
 ↪  {id=insn_0006_i_5_x_i_5_y_update}
││ │         end i_5_y, i_5_x
│└↱└         gradu_None[idim0] = acc_i_5_x_i_5_y  {id=insn_0006_0}
│ │        end idim0
│ │        for i_4_y, i_4_x
├↱├            r_lfsv_loc[i_4_x, i_4_y] = 0.00390625*(4*phi_BCG1[q, i_4_x
 ↪  + 2*i_4_y, 0] + 256*js_BCG1[q, i_4_x + 2*i_4_y, 0, 0]*jit[0,
 ↪  0]*gradu_None[0]*jit[0, 0] + 256*js_BCG1[q, i_4_x + 2*i_4_y, 0,
 ↪  1]*jit[1, 1]*gradu_None[1]*jit[1, 1])*qw_dim2_order2[q]*detjac +
 ↪  r_lfsv_loc[i_4_x, i_4_y]  {id=insn_0008}
│││      end q, i_4_y, i_4_x
│││      for i_7_y, i_7_x
└└└          r_lfsv_alias[e_x, e_y, i_7_x, i_7_y] = r_lfsv_alias[e_x, e_y,
 ↪  i_7_x, i_7_y] + r_lfsv_loc[i_7_x, i_7_y]*r.weight()  {id=insn_0009,
 ↪  tags=accum}
     end e_x, e_y, i_7_y, i_7_x
───────────────────────
...
```

Figure 5.4: The generated Loopy kernel for a 2D block-structured grid with $k = 16$ (abbreviated).

5.2.1 Visitor Specializations

In the following, the visitor specializations are discussed, starting with the quadrature mixin. This mixin realizes the characteristic micro element loops. More specifically, the default mixin defines the quadrature loop, its inames and extents, and the specialized version extents it by adding the micro element loop inames and their extents. Thereby, any instruction previously dependent on the quadrature loop is now also dependent on micro element loops. Besides the definition of the quadrature loops, the mixin implements the positions and weights of the quadrature points. On block-structured grids a distinction between a quadrature point within the micro element and the macro element is implemented. The position w.r.t. the micro element is used for local computations, such as the basis evaluations, while the position w.r.t. the macro element is mostly necessary during the evaluation of the geometry transformation.

As discussed in section 3.1.2, adjustments to the local data structure are necessary. These are addressed by the both the basis mixin and the accumulation mixin. As a first step in both mixins tensor product indices, $i_1, \dots, i_d \in \{0, \dots, p\}$, are introduced for the enumeration of the micro element DoFs belonging to the local $\mathbb{Q}_p$ basis, as mentioned in section 3.1.2. Using this local DoF indexing, and the micro element indexing discussed before, the local coefficient data and the local result data are defined as $2d$-dimensional arrays with shape and strides defined as

$$\text{shape} = (\overbrace{k, \dots, k}^{d\text{ times}}, \overbrace{p+1, \dots, p+1}^{d\text{ times}})$$
$$\text{stride}_j = \begin{cases} p(pk+1)^{j-1} & \text{if } j \le d, \\ (pk+1)^{j-d-1} & \text{if } j > d, \end{cases}$$

which are similar to the shape and strides from listing 5.1. By defining this non-unit stride, Loopy is able to compute the linearized index the same way as in section 3.1.2, since

$$\begin{aligned} \text{idx}(j_1, \dots, j_{2d}) &= \sum_{l=1}^{2d} j_l \cdot \text{stride}_l = \sum_{l=1}^{d} j_l p(pk+1)^{l-1} + \sum_{l=d+1}^{2d} j_l(pk+1)^{l-d-1} \\ &= \sum_{j=1}^{d} (pj_l + j_{d+l})(pk+1)^{l-1} \end{aligned}$$

with $j_l = e_l$ and $j_{d+l} = i_l$ for $l = 1, \dots, d$. As in chapter 3 $e_1, \dots, e_d$ refer to the micro element loop inames and $i_1, \dots, i_d$ denote inames of the local DoFs loop.

For a system of PDEs, an array for each finite element component is defined in that way. These arrays alias the DUNE-PDELAB local vectors passed into the generated kernel to enable direct accessing. Finally, the basis mixin adds the precomputation of the basis evaluation at each micro element quadrature point to the constructor of the generated `LocalOperator` class.

Similar to the default geometry mixin, there are multiple variants of geometry mixin classes for block-structured grids, specialized for different macro element geometry types. The most general variant handles multilinear geometries, which have to be reevaluated at each quadrature point. First, a temporary variable representing the Jacobian of the geometry transformation and the necessary instructions for its computation using the $\mathbb{Q}_1$ basis are cached. Then temporaries and their corresponding instructions, representing the inverse or the determinant, are cached, and the UFL geometry node is transformed into a pymbolic variable symbolizing the precomputed quantity. If the underlying macro grid consists only of axiparallel elements, the mixin adds instructions to evaluate the inverse or determinant only once per macro element, and scales them according to the micro element size. For equidistant macro grids, the mixin derives from the axiparallel mixin, with the optimization to compute the inverse and determinant only once for the whole grid. All specialized block-structured geometry mixins derive from the default mixin and use their implementation for intersection quantities.

Except for the geometry mixin, the block-structured mixins rely indirectly on DUNE's representations. The quadrature mixin assumes that a Gaussian quadrature rule is used in order to compute the number of quadrature points needed at compile time, simplifying the quadrature loop for the C++ compiler. The basis and accumulation mixin expect that the local data passed into the macro element kernel adheres to the layout specified in section 3.1.2. These assumptions are not enforced in any kind, leaving them fragile w.r.t. changes in upstream DUNE modules. Using python bindings for DUNE modules [27] could reduce this issue, by querying for example the quadrature rule for a specific degree from DUNE-GEOMETRY or the local layout interface defined in DUNE-LOCALFUNCTIONS. However, these are still in development.

5.2.2 Preconditioner Generation

The additions outlined above are sufficient to create block-structured Loopy kernels, but the two level preconditioners introduced in chapter 4 require some auxiliary kernels. First of all, local kernels for the coarse grid are needed. These are trivially generated using the default mixins. Since only continuous $\mathbb{Q}_p$ elements are considered, the macro element local restriction and prolongation operations can easily be generated. The required scaling for both the handling of the macro element interface DoFs and the Neumann-Neumann fine grid preconditioner is not generated. Instead, a generic local kernel that simply adds 1 to all macro element DoFs is sufficient. Through the additive assembly of the local contributions, this results in a global vector, containing the number of elements each DoF is contained in, and element-wise inversion results in the required scaling factors.

Both the Jacobi and Neumann-Neumann fine grid correction need some special kernels not shared between each other. The Jacobi preconditioner requires a kernel to precompute the diagonal of the stiffness matrix, since this cannot be computed on-the-fly. Although UFL does not directly support extracting the diagonal of the stiffness matrix, only minor adjustments are necessary. Replacing the first argument of the Jacobian bilinear form with the second one leads to a weak formulation that assembles $\operatorname{diag}(A)_i = a(\phi_i, \phi_i)$ directly.

As laid out in section 4.2, solving the local Neumann or Dirichlet problem depends only on already generated macro element kernels. These kernels require some additional pre- or post-processing. If a Dirichlet condition has to be applied, either artificial ones on internal faces or problem dependent ones on boundary faces, the interface values need to be zeroed out. In other cases, for example applying the local boundary-interface couplings, all interior DoFs need to be set to zero. For each of these operations, specialized copy kernels are generated. The scaling of the identity matrix, required by the Neumann solver, uses the approach outlined in 4.2, which requires the generation of a kernel for computing the macro element local stiffness matrix. This is part of the standard generation process, although for other matrix-free applications it is disabled.

5.3 Optimizations

This section describes the block-structuring specific Loopy kernel transformations implemented in DUNE-CODEGEN. The optimized geometry transformation computation mentioned in section 3.1.1 are part of the specialized mixin classes and, as such, already mentioned in 5.2. The other optimization discussed in chapter 3, vectorization, is implemented as a transformation on top of the Loopy kernel created after the UFL AST traversal and examined later on. Another optimization, mentioned only implicitly in chapter 3, is the precomputation of quantities independent of the micro element loop, like the scaling of gradients in the affine geometry case. In the code generator this is realized through loop invariant code motion (LICM), and will be described in the second part of this section.

5.3.1 Vectorization

Loopy supports vectorization for OpenCL or CUDA targets natively, and the support for C++ code using the VCL vector types has been added in DUNE-CODEGEN. Vectorized Loopy kernels are created by tagging loop inames and shape dimensions of arrays with an appropriate tag. Contrary to the vectorization mostly used in C++ compilers, the vectorized loop does not need to be the innermost loop, which is beneficial for the block-structured grids kernel, since the iname for the cross-element vectorization is never the innermost iname. The vectorization described in section 3.3 is achieved by first tagging the innermost micro element loop to be vectorized, creating vectorizable data aliases, and then adding explicit SIMD load and store expressions. Afterwards, the accumulation instruction is adjusted depending on the vectorization variant used (overlapping or non-overlapping).

Since only inames with loop size of exactly the SIMD vector width w may be tagged, the innermost micro element loop is split into an inner loop of size w and an outer loop of size k/w, where k is the macro element size, assuming that k is divisible by w. If the data layout of the tagged arrays is simple enough, i.e. standard strides are used, Loopy handles the vectorized accessing of the data by itself. This is not the case for the local data structures used in the block-structured kernels. Due to their non-trivial strides as discussed in 5.2.1, Loopy's code generation fails and the kernel has to be transformed to comply with Loopy's requirements for vectorizable code. The transformation adds new temporaries of size w, which alias the access of the $[i_1, \dots, i_d]$ micro element DoF of w adjacent elements, and new instructions to directly load or store these SIMD vectors using the VCL interface.

Depending on the vectorization variant, the accumulation into the local data structure is modified using the approaches mentioned in section 3.3. The non-overlapping variant requires unrolling the innermost micro element loop and merging the computations. For the overlapping variant the loop ordering of the micro element DoF loop is adjusted, such that the loop with the same direction as the vectorized macro element loop becomes the outermost loop. Additionally, some expressions require the value of vectorized iname directly and not as part of an indexing, which cannot be vectorized by Loopy. One instance is the computation of the quadrature position within the macro element. A workaround for that issue is to create an array, containing all values of the iname, and vectorize that array.

The previous discussion assumed that the macro element size is a multiple of the vector width, which is quite strict especially on AVX512 machines with $w = 8$. To accommodate for the cases where $k \bmod w \neq 0$, the innermost micro element loop is divided into two chunks, the first one of size $w\lfloor k/w \rfloor$ and the second one with $k \bmod w$. If desired, the second chunk can be vectorized with a lower SIMD vector width. In order to repeat the process described above, the chunk needs to be realized with duplicated instructions. Loopy does not duplicate these by itself, instead an additional transformation adds the necessary second chunk. The process of splitting the vectorization loop, realizing the tail chunk and vectorizing it may applied recursively until vectorization is not possible anymore. This ensures that as many vectorized instructions as possible are issued during the macro element kernel.

5.3.2 Loop Invariant Code Motion

The major contributor to the local kernel runtime are the number of FLOPs executed within the innermost loop, specifically the operations needed to compute the accumulation expression. One approach to reduce these FLOPs is to use *loop invariant code motion* (LICM), which shares similarities with common subexpression elimination (CSE). In both cases redundant computations are replaced by precomputed temporaries. While CSE only recognizes redundancies within a single instruction, LICM also considers subexpressions which are invariant with respect to the instructions loop nesting, and can thus be precomputed inside a smaller loop nesting. Since the block-structuring approach introduces additional loops and expression depending on these, LICM is better suited than plain CSE. Consider, for example, the kernel in Fig. 5.5. On a structured grid, the scaling of the basis function gradients is independent of the innermost loop and should be precomputed. In such simple cases, the C++ compiler might be able to apply the LICM transformation, but that chance rapidly

Figure 5.5: Example kernel with redundant computations. The kernel computes the local operator application. Since the geometry is assumed to be affine linear, the scaling of the gradients is redundant and should be precomputed outside of the micro element loop.

```
for ex in 0..k
  for (qp, qw) in quadrature
    grad_u = sum(j, grad_phi[j] *
                    u[ex, j])
    for i in 0..n_b
      r[ex, i] += qw / k *
        (1/k * J^-T * grad_u) *
        (1/k * J^-T * grad_phi[i]))
```

decreases for more complex weak formulations or kernels, and therefore this transformation is best applied at the loopy kernel level.

While algorithms implementing LICM are well known and widely used, see for example [26], the difficulty in this case arises from applying them within the Loopy kernel framework. Since an unscheduled Loopy kernel has no information about the nesting of its loop domains, it is not possible to determine loop invariant expressions at that moment. Thus, the kernel must be scheduled, using Loopy's capabilities, before the LICM algorithm can be applied. The algorithm consists of two parts, where the first part identifies subexpression suitable for precomputation, and in the second part these precomputation must be added to the kernel's schedule. It is possible to leave the scheduling of the new kernel to Loopy. However, it is more efficient to directly compute the new schedule, since all the necessary dependency information is already available from the first part. In the following, both parts of the algorithm are described in more detail.

Part I — Finding Subexpressions

The first step is to detect which subexpressions are worthwhile to precompute, by determining the number of redundant computations done for each subexpression. Computing this redundancy requires knowledge of the scope of each subexpression — the minimal loop domain, within which the subexpression is valid w.r.t. the current schedule — and removing that from the loop domain of the root expression containing the subexpression. Afterwards, the size of the remaining loop domain gives the redundancy cost. For example, the minimal loop domain of the expression `t+b[i]` in Fig. 5.6 is defined by the iname `i`, although `t` is also updated in the `k` loop. For the expression `c[j]` the minimal inames are `i,j`, since the `j` loop is nested within the `i` loop. From these minimal loop domains, it can be seen, that `n` additional computations are made for each computation of `d[i,j]`, which should be avoided. Both the detection of the minimal necessary loop domain and the computation of the redundant cost are implemented as AST visitors. Before applying the redundant cost visitor it is advantageous to group the child nodes of the `Sum` and `Product` nodes according to their loop domain to maximize the size of the possible precomputations.

Next, all subexpressions with cost greater than 1 are precomputation candidates, but it is generally not feasible to precompute every possible subexpression, since the high number of added temporaries would increase the pressure on the CPU registers. Instead, the precomputations are filtered into two categories. The first one contains the precomputations that can either be hoisted

Figure 5.6: Simple loop nesting illustrating the redundant cost definition. Although `t` is accessed in loop `i` and `k` its scope is only loop `i`. Thus, the minimal loop domain for the expression `t+b[i]` is `i`, and `i,j` for the expression `(t+b[i])*c[j]`.

```
for i in 0..n
  t = 0
  for k in -1,0,1
    t = t + a[k]
  for j in 0..n
    d[i,j] = (t + b[i]) * c[j]
```

outside the loop domain of their root expressions or have an arithmetic cost higher than a heuristic value. These precomputations are realized as part of the LICM transformation. All other precomputations fall into the second category, and are left for the C++ compiler to decide. Thus, these subexpressions are removed from the ASTs of each root expression.

Finally, the Loopy instructions and temporaries defining the precomputed subexpressions are added to the kernel. But this does not update the existing schedule automatically, which is addressed in the second part. Additionally, a dependency DAG for the newly added instructions is created, mirroring the reduced precomputation AST with the addition of dependencies on already existing instructions.

Part II — Updating the Schedule

Using the dependency DAG from the previous step, the existing schedule is updated. The new schedule must satisfy the following conditions: i) each new precomputation is scheduled after all precomputations or existing instruction it depends on, ii) all precomputations are scheduled before any instruction depending on it, iii) each precomputation is nested within its minimal loop domain. This problem is similar to finding a topological ordering, but condition iii) prohibits the formulation of this problem in that way, since the condition cannot be easily represented in the dependency DAG.

Nevertheless, starting from a topologically ordering of the precomputation subset of the DAG, the schedule index is determined for each precomputation, satisfying the conditions above. The schedule is not modified during this index computation, since this would lead to a $\mathcal{O}(m \cdot n)$ complexity. In this instance m is the number of new precomputations and n is the number of existing instructions. These new indices are used later on to create the updated schedule. This is accomplished by a stable insertion method 5.1, which adds the precomputation to the schedule, such that each precomputation is located at its corresponding index, if no other precomputation were added.

Subject to the dependency types of a precomputation, satisfying the conditions requires special care. If the precomputation does not depend on other precomputations, conditions i) and ii) can be easily satisfied by incrementing the index of the last instruction the new precomputation depends on. To satisfy condition iii) the currently active loop domain needs to be known at each index of the current schedule. Then the insertion index determined before can be increased until the loop domains match. If a precomputation also depends on other precomputations, these instructions are already scheduled due to the topologically ordering used in the beginning. Therefore, using the same index as the last scheduled precomputation it depends on suffices, as the insertion at the end guarantees that two instructions with the same index are inserted in the order they were handled during the DAG traversal. Algorithm 5.2 depicts the described approach.

One aim of using code generation is to transfer the handwritten optimizations applied to a specific problem to a broader class of problems, which requires as a first step that the code generation can replicate the known handwritten optimizations. The described algorithm can apply those optimizations to the examples described earlier. For example, applied to the example Loopy kernel in Fig. 5.4 from section 5.2 it results in the kernel shown in Fig. 5.7. This shows that the code generation, together with LICM, can create macro element kernels with similar computational complexity as the geometry optimizations discussed in section 3.1.1.

5.4 Benchmarks

In the following, the benchmarks for the handwritten code from section 3.4 are revisited to show that the generated kernels reach the performance of the handwritten ones. As before, all examples compute the matrix-free Laplace operator application with constant coefficients. In the generated case both a 2D grid with 1024×1024 fine grid elements and a 3D grid with 128×128×128 fine grid elements are used, while the handwritten kernels are only implemented for the 2D grid. In either cases, the size of the macro elements varies. The first example considers the generated vectorization, without further optimizations. Then, the second example examines the LICM optimization by considering the geometry computation again, and lastly, the overall performance of the operator application is discussed.

Algorithm 5.1: Stable Insertion

```
Function: StableInsertion($a$, $b$, $idx$)
    Data: $a$ list, $b$ list, $idx$ indices of $b$
    Result: $r$ merged list
    $r = []$
    $i = 0$
    $c = 0$
    while $a$ or $b$ not empty do
        if $b$ not empty and $i = idx[0] + c$ then
            $n =$ PopFront($b$)
            PopFront($idx$)
            $c = c + 1$
        else
            $n =$ PopFront($a$)
        Append($r, n$)
        $i = i + 1$
```

Algorithm 5.2: Schedule Update

```
Function: ScheduleUpdate($I$, $D$, $S$, $AI$, $G$, $T$, $MI$)
    Data: $I$ sorted set of scheduled existing instructions, $D$ set of all
          inames, $P(D)$ power set of $D$, $S : I \mapsto \mathbb{N}$ schedule,
          $AI : \mathbb{N} \mapsto P(D)$ maps schedule index to active inames,
          $G = (V, E)$ dependency DAG, $T$ set of precomputation
          instructions, $MI : V \mapsto P(D)$ maps subexpression to
          minimal inames
    Result: $R$ scheduled list of instructions satisfying conditions i)–iii)
    $Q =$ TopologicalSort($G$)
    while $Q \neq \emptyset$ do
        $v =$ Deque($Q$)
        if $v \in T$ then
            $S(v) = \max(\{S(d) + 1 | d \in \text{Children}(v) \cap I\} \cup \{S(d) | d \in$
              $\text{Children}(v) \cap T\})$
            while $AI(S(v)) \neq MI(v)$ do
                $S(v) = S(v) + 1$
    $T =$ StableSort($T$, $key$=$S$)
    $idx =$ StableSort($\{S(v) | v \in T\}$)
    $R =$ StableInsertion($I$, $T$, $idx$)
```

```
...
________________________

INSTRUCTIONS:
↱          t = r.weight()  {id=precompute_t}
│↱         t_0 = 256*jit[1, 1]*jit[1, 1]  {id=precompute_t_0}
││↱        t_2 = 256*jit[0, 0]*jit[0, 0]  {id=precompute_t_2}
│││↱       t_4 = 0.00390625*detjac  {id=precompute_t_4}
││││       for e_y, e_x, i_6_y, i_6_x
││││↱              r_lfsv_loc[i_6_x, i_6_y] = 0  {id=insn_0007}
│││││↱             x_lfsv_loc[i_6_x, i_6_y] = x_lfsv_alias[e_x, e_y,
 ↪  i_6_x, i_6_y]  {id=insn_0005}
││││││         end i_6_y, i_6_x
││││││         for q
│││├││↱          t_5 = t_4*qw_dim2_order2[q]  {id=precompute_t_5}
│││││││          for idim0
│││││││↱           acc_i_5_x_i_5_y = 0  {id=insn_0006_i_5_x_i_5_y_init}
││││││││           for i_5_x, i_5_y
│││││├│└↱              acc_i_5_x_i_5_y = acc_i_5_x_i_5_y +
 ↪  x_lfsv_loc[i_5_x, i_5_y]*js_BCG1[q, i_5_x + 2*i_5_y, 0, idim0]
 ↪  {id=insn_0006_i_5_x_i_5_y_update}
│││││││ │          end i_5_x, i_5_y
│││││└│↱└          gradu_None[idim0] = acc_i_5_x_i_5_y  {id=insn_0006_0}
│││││ ││         end idim0
│├│││↱│├         t_1 = t_0*gradu_None[1]  {id=precompute_t_1}
││├││││├↱        t_3 = t_2*gradu_None[0]  {id=precompute_t_3}
│││││││││        for i_4_x, i_4_y
│└└└├└└├└↱           r_lfsv_loc[i_4_x, i_4_y] = t_5*(4*phi_BCG1[q, i_4_x +
 ↪  2*i_4_y, 0] + t_3*js_BCG1[q, i_4_x + 2*i_4_y, 0, 0] + t_1*js_BCG1[q,
 ↪  i_4_x + 2*i_4_y, 0, 1]) + r_lfsv_loc[i_4_x, i_4_y]  {id=insn_0008}
│   │  │ │     end q, i_4_x, i_4_y
│   │  │ │     for i_7_x, i_7_y
└   └  └ └         r_lfsv_alias[e_x, e_y, i_7_x, i_7_y] =
 ↪  r_lfsv_alias[e_x, e_y, i_7_x, i_7_y] + r_lfsv_loc[i_7_x, i_7_y]*t
 ↪  {id=insn_0009, tags=accum}
           end e_y, e_x, i_7_x, i_7_y
________________________

...
```

Figure 5.7: Abbreviated Loopy kernel resulting from LICM application to the kernel presented in 5.4. The combination of the precomputations t_0,...,t_3 is nearly equivalent to scale the gradients beforehand.

5.4.1 Efficient Vectorization

To show the validity of the vectorization transformation, the speed-up gained by generating the vectorized code is examined, similar to section 3.4.3. The same setup is used, although only low and medium quadrature orders are considered. Fig. 5.8 shows the attained speed-up for the generated kernels, the corresponding values for the handwritten kernels in 2D can be found in Fig. 3.14.

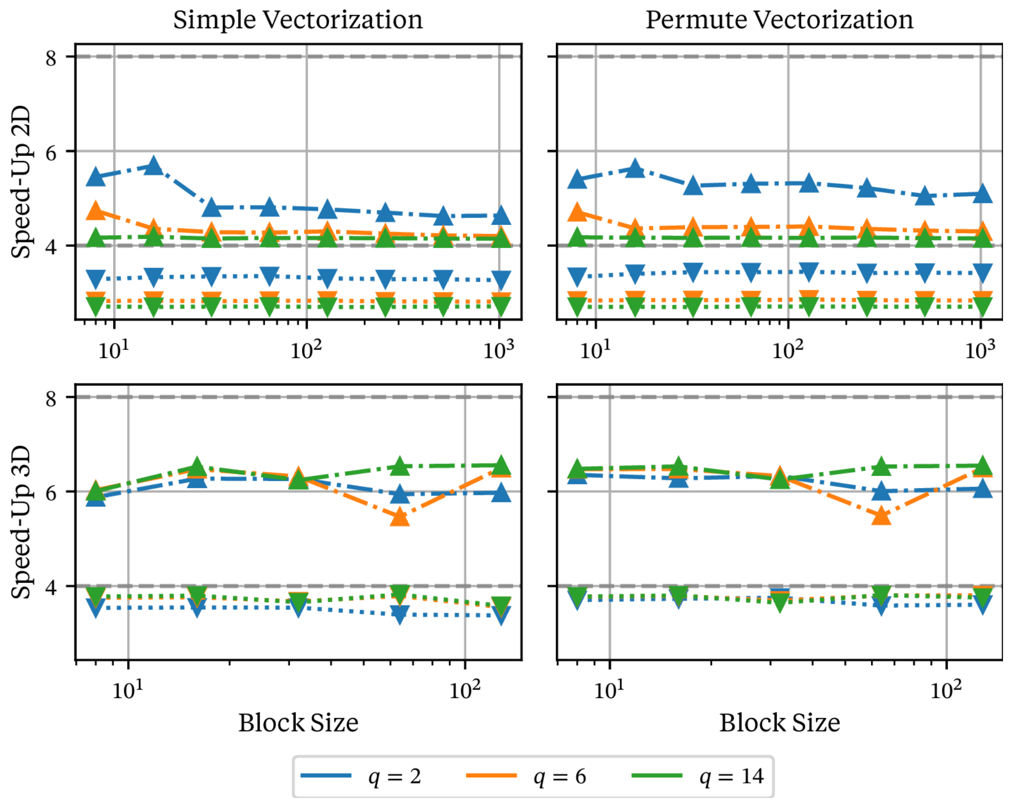

Figure 5.8: Speed-ups of the vectorized local kernels compared to unvectorized kernels for varying macro element sizes. The dotted lines correspond to AVX2 vectorized kernels (SIMD width 4) and the dashed-dotted lines to AVX-512 vectorized kernels (SIMD width 8). The speed-up is shown only for generated kernels, refer to Fig. 3.14 for the handwritten kernels. The 2D computations don't fully reach the optimal speed-up, especially the medium quadrature rule computations. In 3D, good and nearly ideal speed-ups are achieved, with little variation between the overlapping and non-overlapping variant. The poorer results in the 2D case might be explained by the better auto vectorization of the baseline kernel.

Compared to the handwritten speed-ups, the generated kernels achieve generally lower speed-ups in 2D. Especially for kernels with AVX-512 instructions and medium quadrature orders this difference is significant, while for $q = 2$ the gap is nearly closed. Again it should be noted that the baseline kernels still uses auto vectorization from the C++ compiler. Combined with the higher number of FLOP compared to the handwritten kernels in 3.4.3 due to omitting further optimizations, this could result in a higher performance of the baseline kernels, which would in turn reduce the scaling gained from cross-element vectorization. Furthermore, in 3D the auto vectorization fails, and therefore the generated kernels reach higher speed-ups, which resemble the expected speed-ups more closely. This shows that generating vectorized code can achieve nearly optimal speed-ups, which will be supported by the operator application benchmark in section 5.4.3.

5.4.2 Efficient Geometry Computation

The following benchmark examines the optimizations w.r.t. to the type of the geometry transformation, multilinear or affine, more closely and compares it with the handwritten code used in section 3.4.2. The setup is the same as in that section, and only the optimized handwritten kernels are compared to the generated kernels with the LICM optimization enabled. Additionally, the runtime reduction on a 3D grid is considered for the generated kernels.

The resulting performance, as illustrated in 5.9, is considered in the following The visitor optimizes the precomputations strictly constrained to the computation of the Jacobian determinant and the Jacobian's inverse. All other precomputations, such as scaling the gradients, are realized by the LICM transformation. These optimizations combined can replicate the runtime reduction of the handwritten kernels. In the affine case, the generated kernels are slightly slower, but for the multilinear case they achieve an additional reduction of 10% points, which are most likely caused by further precomputations realized by LICM that are not present in the handwritten case. The computations in 3D do not achieve the same reduction as in the 2D case, since the computation of the geometry quantities takes up less of the kernel runtime.

5.4.3 Operator Application

Finally, the benchmark from section 3.4.4 is replicated, examining the performance of the generated, matrix-free operator application. Both previously discussed optimizations are combined in this example. Fig. 5.10 shows the FLOP/s performance and Fig. 5.11 displays the DoF/s performance of one operator application, comparing the generated and handwritten kernels in 2D. Addition-

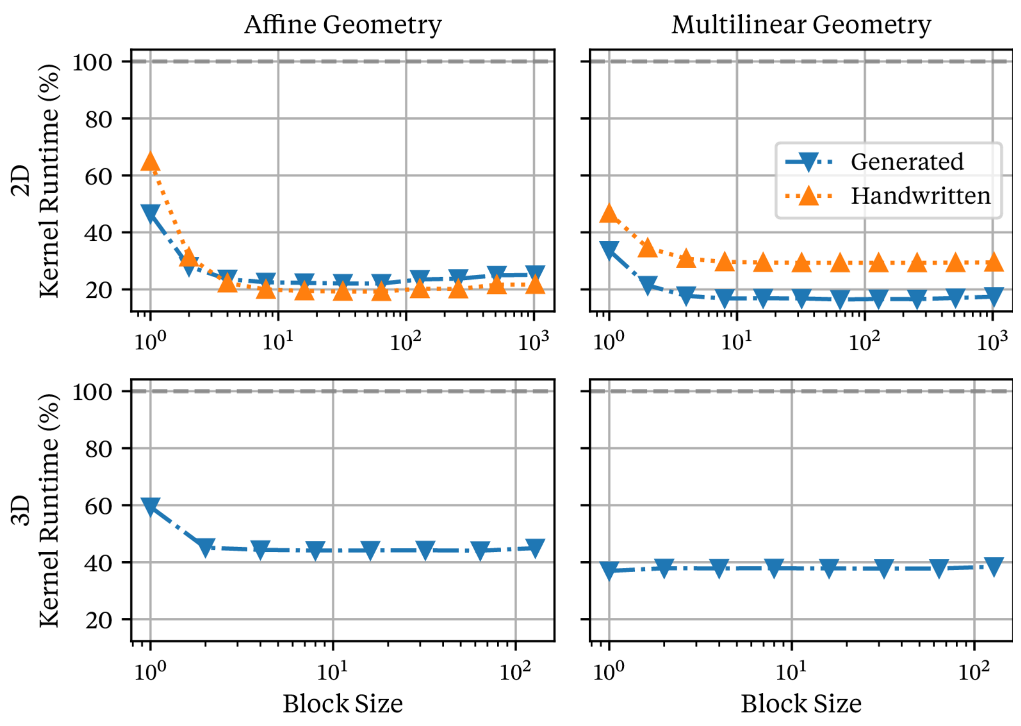

Figure 5.9: The local kernel runtime of block-structured kernels, both generated and handwritten, on grids with affine and multilinear geometries, reported as percentage of the non block-structured kernel. The handwritten kernel has been optimized by hand, while the generated kernel relies on LICM. Compared to the handwritten kernels, the generated kernels achieve nearly the same runtime reduction, even surpassing the handwritten performance for the multilinear case. The 3D behavior is similar to the 2D case, although the increased non geometry related work leads to a smaller reduction.

ally, the performance of the generated kernels in 3D is provided. In Fig. 5.11 the grayed-out line represents the theoretical performance of a matrix-based operator application for this problem, i.e. ∼35 MDoF/s in 2D and ∼11 MDoF/s in 3D. Across all benchmarks the generated kernels achieve nearly the same performance as the handwritten kernels, in some cases it even surpasses the handwritten performance. This comparison establishes generated kernels as a suitable replacement for kernels optimized by hand.

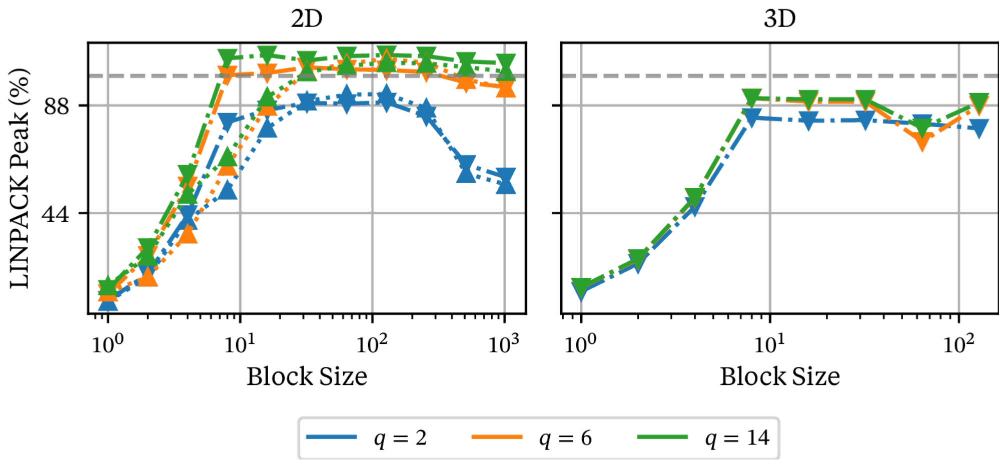

Figure 5.10: FLOP/s performance of the macro element kernels reported as % of LINPACK peak performance for the generated kernels (dashed-dotted lines) and the handwritten kernels (dotted lines). The generated kernels are also considered in 3D. 100% of LINPACK peak on one core corresponds to ~45.5 GFLOP/s. In 2D the generated kernels achieve their peak performance faster than the handwritten ones. Nevertheless, the peak performance of both variants is nearly indistinguishable. The 3D kernels exhibit the same qualitative behavior as the 2D cases, although the FLOP/s performance is slightly lower than the LINPACK peak.

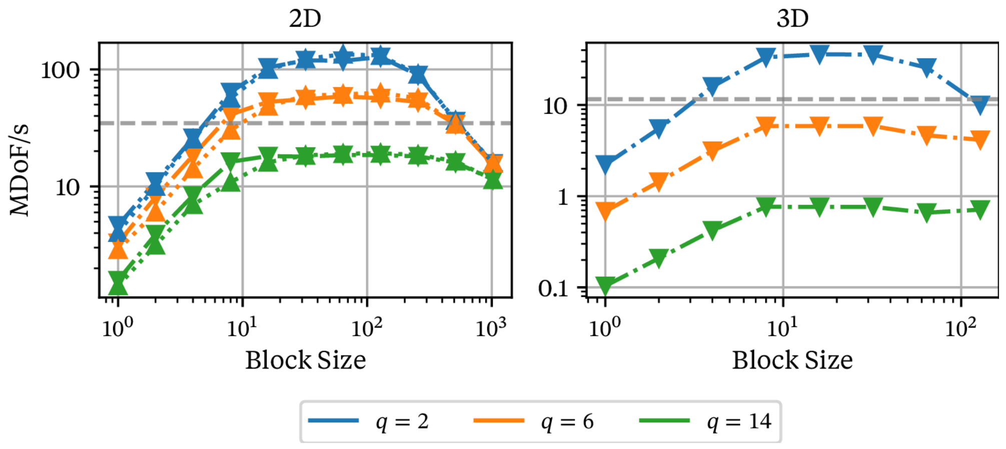

Figure 5.11: Operator application performance in MDoF/s for the generated kernels (dashed-dotted lines) and the handwritten kernels (dotted lines). The generated kernels are also considered in 3D. Additionally, the dashed, grayed-out line depicts the performance of the corresponding assembled matrix application. Similar to the FLOP/s discussion, the generated kernels achieve the same performance as the optimized, handwritten ones. Additionally, in 3D the matrix-free application with the $q = 2$ kernel is faster than the corresponding matrix vector multiplication.

5.5 Outlook

The code generation for block-structured grids, introduced in this chapter, can result in similar performance to handwritten codes. The discussed optimizations, realized either as specialized visitor or as Loopy transformations, are crucial to achieving this performance, but they are not the only possible optimizations. As seen in the numerical examples presented earlier as well as in chapter 3, the performance of block-structured kernels degenerates for large macro element sizes, starting in the hundreds for 2D grids and a bit earlier for 3D grids, which could be offset by tiling the micro element loop to increase cache locality. The benefits of this optimization are well understood and can be seen for example in [89]. Additionally, the LICM optimization discussed earlier is restricted to precomputing scalar expressions. But, in some cases the FLOP count of the kernel can be reduced further by precomputing array expressions, using the approach outlined in [71].

One issue in the current generation approach are the implicit dependencies on DUNE, mentioned in the introduction of section 5.1 and during the discussion of the block-structured specific generation. Some of these dependencies could be reduced, by extending the python bindings for DUNE modules [27, 11] and then querying the modules directly to gain the required information. Another way to minimize these dependencies and at the same time increase the flexibility of the code generator would be to extend the generation beyond local kernels. Out of necessity, this is already done for some parts of the block-structured preconditioner. Still, other components of the block-structured code generation may also benefit from it. For example, generating the gather and scatter functions depending on the local data structure would allow the use of the layout explained in 3.5, enabling vectorization for arbitrary local polynomial degrees.

5.5.1 Representation of Local Structure

The current code generation is restricted to continuous Lagrange elements on cubical grids, but more flexibility would be desirable, as already mentioned in 3.5. This flexibility could be gained by using a symbolic representation of local structuring patterns. Besides the discussed block-structured refinement with cube, these could describe a structured refinement with simplices, or a column wise extrusion of a 2D grid. Conceptually, the structuring refers to creating a new structured mesh $\mathcal{T}_e$ with N_e new micro elements e for each macro element E of the original macro Mesh $\mathcal{T}_0$. The micro elements may have a different type than the original element E. Considering the block-structured case with cubical elements, the meshes $\mathcal{T}_e$ are again cubical meshes, but in the extrusion case the elements e are prism elements, created by a tensor product of the original element type and an interval. The generated local kernel operates on each E, handling the iteration over the mesh $\mathcal{T}_e$, while the outside assembly engine handles the iteration over $\mathcal{T}_E$.

The new mesh is structured in the sense that the new elements are indexed by a multiindex α defined over an integral polyhedral set Φ, short index set, and there exists an iteration order $I : \Phi \mapsto [0, N_e)$. For example, the polyhedral set for a block-structured refinement in 3D is defined as

$$\Phi = \{(i, j, k) : 0 \leq i, j, k < N_c\},$$

with lexicographic iteration order given by $I(i, j, k) = i + jN_c + kN_c^2$, and for an extrusion of a 2D simplex grid as

$$\Phi = \{(i) : 0 \leq i < N_c\},$$

where N_c denotes the cells per direction in the first case and the number of extrusion levels in the second case. These sets cover only the volume of the micro elements, and additional index sets Φ_s for subentities s with different codimension are needed, as well as their corresponding iteration maps I_s. Furthermore, the full description of the local structure requires the topological connections between entities of different codimension, i.e. mapping between indices in different subentity set.

These topological connections are defined by the maps,

$$\begin{aligned} T_{s,s'} &: \Phi_s \times \beta \mapsto \Phi_{s'}, \\ T^*_{s',s} &: \Phi_{s'} \mapsto \Phi_s \times \beta, \end{aligned}$$

where s is an entity of codimension $0 \leq c \leq d$ and s' has codimension $c + 1$, and β denotes the subentity index on the reference element of s' w.r.t. s. The maps $T_{s,s'}$ and $T^*_{s,s'}$ are sometimes called cone and support operations respectively, for example in the PETSc unstructured grid interface DMPlex, see [60]. Concatenating these maps allows, for example, querying all vertex indices of a cell (i, j, k) within a block-structured refinement, which is necessary to identify the DoFs associated with that cell if a $\mathbb{Q}_1$ finite element is used.

In the following, the local extrusion case is examined in more detail to clarify the meaning of the previously described objects. Here, the cell index set is the same as above, and the face entities of codimension 1 are divided into two distinct classes, horizontal faces Φ_{hf} and vertical faces Φ_{vf}, defined by

$$\begin{aligned}\Phi_{hf} &= \{(i) \,:\, 0 \leq i < N_c + 1\},\\ \Phi_{vf} &= \{(i,j) \,:\, 0 \leq i < N_c,\, 0 \leq j < 3\}.\end{aligned}$$

In the same manner, there is a distinction between horizontal edges Φ_{he} and vertical edges Φ_{ve}, where the index sets are given by

$$\begin{aligned}\Phi_{he} &= \{(i,j) \,:\, 0 \leq i < N_c + 1\},\\ \Phi_{ve} &= \{(i,j) \,:\, 0 \leq i < N_c,\, 0 \leq j < 3\},\end{aligned}$$

and finally the vertex index set Φ_v is defined as

$$\Phi_v = \{(i,j) \,:\, 0 \leq i < N_c + 1,\, 0 \leq j < 3\}.$$

The corresponding cone maps are defined as

$$\begin{aligned}
T_{c,hf}(i,\beta) &= \begin{cases} i & \beta = 0 \text{ bottom face,}\\ i+1 & \beta = 1 \text{ top face,}\end{cases}\\
T_{c,vf}(i,\beta) &= (i) \quad \beta \in \{0,1,2\},\\
T_{hf,he}(i,\beta) &= (i) \quad \beta \in \{0,1,2\},\\
T_{vf,he}((i,j),\beta) &= \begin{cases} (i,j) & \beta = 0 \text{ bottom edge,}\\ (i+1,j) & \beta = 1 \text{ top edge,}\end{cases}\\
T_{vf,ve}((i,j),\beta) &= \begin{cases} (i,j) & \beta = 0 \text{ right edge,}\\ (i,j+1 \mod 3) & \beta = 1 \text{ left edge,}\end{cases}\\
T_{he,v}((i,j),\beta) &= \begin{cases} (i,j) & \beta = 0 \text{ right vertex,}\\ (i,j+1 \mod 3) & \beta = 1 \text{ left vertex,}\end{cases}\\
T_{ve,v}((i,j),\beta) &= \begin{cases} (i,j) & \beta = 0 \text{ bottom vertex,}\\ (i+1,j) & \beta = 1 \text{ top vertex.}\end{cases}
\end{aligned}$$

It should be noted that it is not guaranteed for every entity of codimension c to be connected to all types of entities with codimension $c+1$. These cone maps can now be used to gather all subentities topologically connected to a subentity of higher codimension. For example, iteratively applying these maps for each possible value of β, starting from an index i within Φ_c, results lastly in the indices $\{(i,0),(i,1),(i,2),(i+1,0),(i+1,1),(i+1,2)\} \subset \Phi_v$, which represents the indices of the vertices of the cell i.

For completion's sake, the support maps, used to gather topologically connected subentities in the other direction, are given by

$$\begin{aligned}
T^*_{v,ve}(i,j) &= \{(i-1,j),(i,j)\}, & T^*_{he,he}(i,j) &= \{i\},\\
T^*_{v,he}(i,j) &= \{(i,j-1 \mod 3),(i,j)\}, & T^*_{vf,c}(i,j) &= \{i\},\\
T^*_{ve,vf}(i,j) &= \{(i,j-1 \mod 3),(i,j)\}, & T^*_{hf,c}(i,j) &= \{i-1,i\},\\
T^*_{he,vf}(i,j) &= \{(i-1,j),(i,j)\},
\end{aligned}$$

where additional bound checking has to be applied in the case of boundary indices, e.g. when computing the support of the horizontal face with index 0.

The representation can simplify the code generation process for different types of local structuring by defining abstract interfaces, one for the mesh topology and one for its geometry. These can then be specialized for concrete local structure patterns. A preliminary implementation of both interfaces is available in DUNE-CODEGEN[1]. The topology interface is composed of the index sets for all subentities, and the cone and support maps as mentioned above. Furthermore, an index relation map is defined, which computes the topological relation between an index in one index set and any other index set regardless of their codimension, with the constraint that the target entity must be reachable from the starting entity. This new map has a generic implementation, which applies iteratively either the cone or support map, until the target index set is attained. As demonstrated above, these maps are useful to describe the iteration over subentities within the macro element and identify which DoFs, for example defined at the vertices, are connected to the current entity.

For certain integrals, the iteration domain specified by an index set is not suitable. During the integration over all interior faces of an extruded column, for instance, the first and last index of the index set should be excluded. This motivates the addition of interior and exterior sets to the index sets. These sets restrict the index set to the indices needed for an iteration over the interior or exterior micro element entities captured in the set, although they are not necessarily the interior or exterior of the entity set in a strict set topological sense. For instance, the index set of the horizontal edges of a block-structured

[1]Within the branch `feature/structured-refinement`:
`https://gitlab.dune-project.org/extensions/dune-codegen/-/tree/feature/structured-refinement`
Supported are a 2D block-structured refinement and a 1D extrusion.

refinement with macro element size k is given by

$$\Phi_{hf} = \{(i, j) : 0 \le i < k \land 0 \le j < k + 1\}.$$

The required indices for iterating over interior horizontal faces are defined by

$$\mathrm{I}_{hf} = \{(i, j) : 0 \le i < k \land 1 \le j < k\},$$

while the interior set in the strict sense

$$\mathrm{int}(\Phi_{hf}) = \{(i, j) : 1 \le i < k - 1 \land 1 \le j < k\}$$

misses the horizontal faces that are only touching the macro element vertical boundary, but are not entirely contained in the boundary.

The second part of the interface, the mesh geometry, connects the topological description of the locally structured mesh with the physical mesh. It defines a mapping from a given entity index to the physical corners of that entity. These are typically required to compute any geometric quantity and can be defined using the corners of the corresponding macro element entity. As part of the UFL preprocessing of the weak formulation, the spatial coordinates of an entity with codimension greater than 0 are reformulated in terms of the spatial coordinates of the cell corners and a transformation from the reference entity to the reference cell. Therefore, the geometry mapping reduces to computing the corners of a given cell entity and implementing transformation within the reference element.

Unfortunately, this approach leads to inefficient computations without careful considerations. For a 2D block-structured refinement, the computation of the global coordinates of a given quadrature point q in the cell entity (i, j) is given by

$$T(q) = T_E\left(T_{e[i,j]}(q)\right) = \sum_l c_l \phi_l((q + [i\ j]^T)/k),$$

where l iterates over the multilinear basis ϕ_l of the geometry transformation, and c_l denotes the macro element corners. The approach using the mesh geometry interface, as outlined above, would be written as

$$\begin{aligned} T(q) &= \sum_l v_l \phi_l(q), \\ v_l &= \sum_i c_i \phi_l(c_l + [i\ j]^T), \end{aligned}$$

where v_l are the physical coordinates of the micro element (i, j). The second variant involves significantly more operations than the former one. However, this is not an argument against this interface definition. Since the multilinear basis is known at generation time, the expression may be simplified, yielding the efficient formulation under the right choice of simplifications.

CHAPTER 6

Performance Evaluation

IN THIS CHAPTER the components of a matrix-free simulation using block-structured grid, discussed in the previous chapters, are combined and their performance as a whole is examined. But, as a first step, some theoretical performance estimates are developed. With these estimates, the usefulness of the block-structured matrix-free approach might be predicted, especially regarding if the operator application would be faster than a sparse matrix-vector multiplication. Afterwards, multiple benchmarks for different types of PDEs are considered, each highlighting specific strengths and weaknesses of the block-structured grids approach, and to validate the theoretical performance estimates.

6.1 Theoretical Performance

In the following, the theoretical performance of one full matrix-free operator application is investigated, with the intent to predict the runtime of a given matrix-free operator for a specific problem and to compare it with the corresponding matrix-based operator. The roofline model described in section 2.4 is a handy approach for simple loops, but not applicable in this case. Since the kernel of the matrix-free operator application use localized data instead of directly accessing global data, the main assumption of the roofline model, memory transfers and computations overlap, does not hold anymore, at least w.r.t. the main memory. Instead, the performance of two mostly separate segments, the global-to-local gather and scatter, and the local kernel application, has to be considered individually. Due to pipelining and out-of-order execution some overlap still exists, but the main chunk of both parts should be separated. Therefore, the theoretical runtime T^{MF} for the matrix-free application is modeled as

$$T^{MF} = T^{MEM} + T^{COMP}.$$

A downside to this approach is that determining the grid size independent performance in DoF/s is not directly possible anymore, whereas that is the case for the roofline model used in section 2.4.

The memory bound runtime T^{MEM} is easily approximated by

$$T^{MEM} = \frac{\#\text{Byte}}{BW},$$

similar to the roofline model, where #Byte is the total number of transferred bytes and BW is the main memory bandwidth. Applying the matrix-free operator requires three streams in total to compute $y = y + Ax$. One read-only stream is necessary for the vector it is applied to, and one read-write stream for the result. On a block-structured grid in $\mathbb{R}^d$ with N_E macro elements and k micro elements per direction, one stream transfers $N_E \cdot M_{loc}$ values over the course of one operator application. For a scalar, CG finite element with local basis degree p, the local data size is given by

$$M_{loc} = (k \cdot p + 1)^d.$$

In the case of a finite element with multiple components M_{loc} adds up accordingly. Additional n_c streams may be required, which could represent interpolated coefficients of the PDE or a linearization point if a non-linear PDE is considered. Therefore, the total number of transferred bytes (for a double precision data type) is given by

$$\#\text{Byte} = 8 \cdot (3 + n_c) \cdot N_E \cdot M_{loc}.$$

For a comparison with the runtime of a matrix-based operator application, it is beneficial to write this number in terms of the problem size N. As a first step, the transferred data is split based on codimension resulting in

$$\begin{aligned} N_E \cdot M_{loc} &= N_E \cdot (k \cdot p + 1)^d \\ &= N_E \cdot (k \cdot p - 1)^d + 2 \cdot N_{Fi} \cdot (k \cdot p - 1)^{d-1} + N_{Fe} \cdot (k \cdot p - 1)^{d-1} \\ &\quad + \mathcal{O}(N_E (k \cdot p)^{d-2}) \end{aligned}$$

with N_{Fi} denoting the number of interior macro faces, and N_{Fe} the number of exterior faces. The data associated with the macro element's volume is transferred only once, while the data attached to entities with higher codimension is transferred multiple times, as these are shared between multiple macro elements. Therefore, more than N values are handled in total, and more specifically

$$N_E \cdot M_{loc} = N + \beta,$$

with $\beta > 0$ and $\beta = \mathcal{O}\left(N_F(kp)^{d-1}\right) = \mathcal{O}\left(N_E(kp)^{d-1}\right)$. Relative to the total amount of data $N = \mathcal{O}\left(N_E(kp)^{d-1}\right)$, the shared data is negligible for larger block sizes due to $\beta/N = \mathcal{O}((kp)^{-1})$. Thus, leaving it out in the approximation leads to

$$T^{MEM} = \frac{\#\text{Byte}}{BW} \approx \frac{8 \cdot (3 + n_c) \cdot N}{BW},$$

which resembles the estimate for the matrix-based approach. If the simplification is too broad, which might be the case for lower block sizes, the shared data volume can be taken into consideration by increasing the coefficient parameter n_c.

Deriving a precise estimate for the runtime of the local kernels is more complicated and depends more tightly on the discretized PDE. The runtime is simply modeled as

$$T^{COMP} = \frac{\#\text{FLOP}}{P},$$

where #FLOP is the total number of floating point operations executed during one operator application, and P the achieved performance as FLOP/s. According to the roofline model, the FLOP/s are easily computed as $\min(\text{AI}{\cdot}BW, P_{\text{peak}})$, assuming the data resides in the main memory. It is possible to use the same approach defining an AI w.r.t. the L1 cache, but this requires careful inspection of assembly code and ignores other in-core effects, such as dependency chains.

Therefore, the roofline model is not a suitable estimator for the performance of the local kernels and more fitting models need be found. A more nuanced model is the execution-cache-memory (ECM) model [52], which provides accurate results for relatively simple loops. This model requires the number of transfers between the core registers and the L1 cache, as well as the number of arithmetic operations, ideally grouped by their corresponding instruction throughput. However, these are hard to predict accurately based on the source code even for simple local kernels, due to the C++ compiler's optimizations. Therefore, this model is currently not considered further, instead the FLOP/s performance from the LINPACK benchmark is used later on. An interesting approach for future investigations would be to use the FLOP counting capabilities of Loopy, although these still don't account for all optimizations, or to run the compiled kernel on a dummy macro element and use hardware counters to determine the actual number of executed operations.

Similarly to the previous discussion for the total number of transferred bytes, the number of executed FLOP can be approximated by estimating the macro-element local computations. Starting from

$$\#\text{FLOP} = N_E \cdot F_{loc}$$

the macro element local work F_{loc} can be further factorized as

$$F_{loc} = k^d \cdot (n_q \cdot (n_b \cdot F_{dof} + \gamma) + \varepsilon_0) + \varepsilon_1,$$

where k is the block size as before, n_q the number of quadrature points, $n_b = (p+1)^d$ the number of micro element local basis functions, and F_{dof} the number of FLOP executed at one DoF of one micro element for one quadrature point. The parameters γ, ε_0, and ε_1 denote additional FLOP independent of the local basis or quadrature points. Since the scaling of ε_0 and ε_1 is negligible compared to the remaining terms, they are dropped in the following discussions.

To reformulate this in terms of the number of DoFs, it would be tempting to use the approximation $(k \cdot (p+1))^d \approx (k \cdot p+1)^d$, but for small p this is widely inaccurate. Instead, a scaling factor s is introduced as

$$(k \cdot (p+1))^d = (k \cdot p+1)^d \cdot \left(\frac{k \cdot (p+1)}{k \cdot p+1}\right)^d \approx (k \cdot p+1)^d \cdot s^d, \quad \text{with } s = \frac{p+1}{p},$$

which can be interpreted as how often the same DoF will be considered during one macro element computation. For example, an interior DoF of a $\mathbb{Q}_1$ discretization on a 2D grid will be handled during the computation of $s = 4$ micro elements. Now, the number of DoFs can be introduced as before, leading to

$$\begin{aligned}\#\text{FLOP} &= N_E \cdot F_{loc} \\ &\approx N_E \cdot (k \cdot p+1)^d \cdot s^d \cdot n_q \cdot (F_{dof} + \gamma') \quad \text{with } \gamma' = \frac{\gamma}{n_b} \\ &\approx N \cdot s^d \cdot n_q \cdot (F_{dof} + \gamma'),\end{aligned}$$

and finally

$$T^{COMP} = \frac{\#\text{FLOP}}{P} = \frac{N \cdot s^d \cdot n_q \cdot (F_{dof} + \gamma')}{P}.$$

Due to the difficulties of determining both P and F_{loc} as outlined earlier, this approach is currently not capable of estimating the performance beforehand, but it can be used to formulate conditions under which the matrix-free application is faster than the corresponding matrix based one. It needs to be stressed that due to the imprecise approximations this can only give a rough estimate.

Recalling the roofline model for the matrix-based application from section 2.4 leads to the runtime

$$T^{MB} = \frac{8 \cdot (2 + \alpha) \cdot nnz}{BW}.$$

For nearly every relevant number of additional coefficients, the runtime for the memory transfer part T^{MEM} is faster than T^{MB}, more precisely

$$T^{MEM} < T^{MB} \quad \text{if} \quad 3 + n_c' < (2 + \alpha) \cdot nzr,$$

where $nzr = nnz/N$ is the average number of non-zeros per row. Therefore, the matrix-free application can only be faster than the matrix-based application if

$$T^{COMP} < T^{MB} - T^{MEM},$$

$$\Leftrightarrow \frac{N \cdot s^d \cdot n_q \cdot (F_{dof} + \gamma')}{P} < \frac{8 \cdot ((2 + \alpha) \cdot nzr - 3 - n_c') \cdot N}{BW}.$$

This results in the following upper bound for the amount of work at one DoF

$$n_q \cdot (F_{dof} + \gamma') < s^{-d} \cdot ((2 + \alpha) \cdot nzr - 3 - n_c') \cdot \frac{8 \cdot P}{BW}.$$

The previous estimate shows that if a high quadrature order is required or the evaluation at one DoF is costly, the matrix-free application may be slower than the matrix-based version, despite using significantly less memory transfers. The first part of the right-hand-side is PDE and discretization dependent through the number of non-zeros per row and the number of coefficients, while the second part is only machine dependent, encoding how many floating point operations can be executed during the transfer of one double precision value (8 byte), from main memory. Due to the necessary approximations, this estimate is not strict in the sense that even if the estimate holds, the matrix-free application may still be slower. Instead, this bound guarantees that after a specific number of floating point operations per DoF the matrix-based approach is faster than the matrix-free one.

To get a more practical understanding of this estimate, some examples are considered. As discussed above, P is estimated using the LINPACK benchmark and with BW provided by the STREAM benchmark this results in a machine dependent factor of $P/BW \approx 8$ for the Skylake architecture described in appendix A.1. For the discretized Poisson equation the stiffness matrix has $nzr \approx (2 \cdot p + 1)^d$ on a structured grid in $\mathbb{R}^d$, and more specifically $nzr \approx 3^d$ for $\mathbb{Q}_1$ basis functions. Assuming the best cache reuse during the matrix application, i.e. $\alpha = 0$, and no additional coefficients are necessary, the upper bound for the total computation per DoF is given by

$$n_q \cdot (F_{dof} + \gamma') < \begin{cases} 30 \cdot 8 & \text{if } d = 2, \\ 51 \cdot 8 & \text{if } d = 3, \end{cases}$$

where the scaling factor $s = (p+1)/p = 2$ was used. In the same setting, $n_q = 2^d$ quadrature points are necessary to accurately compute the micro element integrals resulting in the following FLOP bound per quadrature

$$F_{dof} + \gamma' \lesssim \begin{cases} 8 \cdot 8 & \text{if } d = 2, \\ 3 \cdot 8 & \text{if } d = 3. \end{cases}$$

On a structured grid and without coefficients $F_{dof} \approx 10$, and thus the simple estimate shows that under idealized assumptions the matrix-free application should be faster, even if the matrix vector multiplication has the best cache reuse. The estimate is supported by the benchmark discussed in section 6.2.1.

In less favorable settings, the allowed number of FLOP before the matrix-based application is guaranteed to be faster might be significantly lower. If, for example, an unstructured grid is used, the number of quadrature points should be increased to $n_q = 3^d$, which reduces the previous bound noticeably. As the matrix-based performance is not dependent on the quadrature order, the bound is lowered to

$$F_{dof} + \gamma' \lesssim \begin{cases} 10/6 \cdot 8 & \text{if } d = 2, \\ 8/9 \cdot 8 & \text{if } d = 3. \end{cases}$$

Additionally, if the floating point performance P is overestimated, then the bound is reduced even further. Assuming, for example, that only half of the LINPACK performance can be achieved then the estimate reduces to $\sim$ 120 FLOP per DoF in total in the 2D case and $\sim$ 200 FLOP in the 3D case, on structured grids. The benchmarks in section 6.2.3 illustrate this problem.

6.2 Measured Performance

In the following, three distinct benchmarks are considered to evaluate the performance of the matrix-free block-structured grid approach with generated local kernels. For comparison, a matrix-based implementation without block-structuring is used as a baseline. In contrast to the block-structured variants, the matrix-based local kernels are handwritten, utilizing, if possible, existing DUNE-PDELAB implementations, and using a blocked sparse matrix if the PDE has multiple components. The performance discussion is divided into two parts, the first one focused on a single operator application, and the second one considering the full simulation process, i.e. solving a linear or non-linear system. Due to the immature nature of the block-structured preconditioner the main focus lies on the operator application, while the full solution acts only as an accessory.

The first benchmark is a Poisson equation with an additional coefficient on a structured grid. Due to the simplicity of the PDE and the subsequent simplicity of the local kernels, this example is well suited to illustrate the advantages the block-structured matrix-free approach can provide compared to the matrix-based one. Next, a non-linear extension to this model is considered, which increases the micro element local computations immensely, thereby reducing the efficiency of the matrix-free operator application. But the missing matrix-assembly may improve the overall performance, making it comparable to the matrix-based approach. And finally, the performance of solving the Navier-Stokes equation with the Chorin projection method is examined. Besides the new characteristics stemming from an explicit time stepping method, this benchmark explores the effects of a multilinear grid on the overall performance.

Some implementation details of the benchmarks are discussed in the following. The first two benchmark are discretized with $\mathbb{Q}_1$ elements, and the last benchmark with both a Taylor-Hood element pair and an unstable element pair. For the first benchmark, the matrix-based variant uses preexisting local kernels from DUNE-PDELAB, while for the remaining benchmarks the kernels had to be implemented manually. In the block-structured case, each local kernel is generated with the optimizations discussed in the previous chapters enabled, namely specialized geometry transformations, loop invariant code motion, and vectorization using the permutation approach. The first benchmark requires the restriction of the piece-wise constant coefficient for matrix-free preconditioner. As the restriction and prolongation operations are currently only implemented for continuous functions, this is implemented by hand. The driver for all benchmarks are also handwritten both in the matrix-free case and the matrix-based case.

The linear systems appearing in each model are solved with the BiCGStab method, using the AMG from DUNE-ISTL in the matrix-based case and the two-grid, matrix-free preconditioner with a Jacobi iteration on the fine grid otherwise. For the Poisson-type systems appearing for these models the AMG provides good convergence behavior. Additionally, it is easy to use with an implementation effort similar to the generated variant, while other preconditioners, such as geometric multigrid methods, would require more elaborate user implementations. Therefore, the AMG is a suitable choice for the tests considered here. Furthermore, most of the observations remain valid if other preconditioner or solution techniques are compared, as long as they are still matrix-based, since the operator results are not altered by this change, while the solver results mostly impact the requirements on the matrix-free preconditioner. The comparison with non-iterative methods, for instance the fast multipole method or the fast Fourier transformation, which are especially suitable for the Poisson problem on structured grids (see [44]), is out of the scope of this work.

6.2.1 Poisson Model

This benchmark considers the Poisson equation with a scalar coefficient

$$-\nabla \cdot (c(x)\nabla u(x)) = f(x)$$

in the domain $\Omega = [0,1]^d, d = 2,3$, with homogenous Dirichlet conditions. The coefficient is piece-wise constant on the micro element grid with values randomly chosen from $[0.5, 2]$ using a uniform distribution, and the right-hand-side is chosen as $f = 1$. Different block sizes $k = 8 \cdot i$ with $i = 1, \dots, 8$ are considered to examine the performance of small and medium block sizes more detailed. A structured mesh with n_e^d micro elements is used, where for each block size multiple values for n_e are chosen as $n_e = 2^i \cdot k$ such that the total number of DoFs $N = (n_e + 1)^d$ is between 10^3 and 10^7. For larger grid sizes, the solution on one core becomes quite expensive and therefore distributing the grid across more cores would be more beneficial, while for smaller sizes the full performance of the codes is usually not achieved yet. Furthermore, this range contains problem sizes, where the whole problem inclusive the full matrix can be cached, which is extremely favorable for the matrix-based application, as well as problems, where the data needs to be transferred from the main memory.

Operator Application Measurements

Concerning the operator application, the most significant performance metric is DoF/s, which represents a grid independent measure on the efficiency of the implementation. Assuming that the runtime grows linearly in the number of DoFs this metric gives the proportion factor in that relation, allowing to estimate the runtime of the operator application on any grid size. Fig. 6.1 and 6.2 depict the DoF/s for the matrix-based operator application, i.e. a sparse matrix vector multiplication, as well as for the matrix-free operator applications with different block sizes, both in 2D and in 3D.

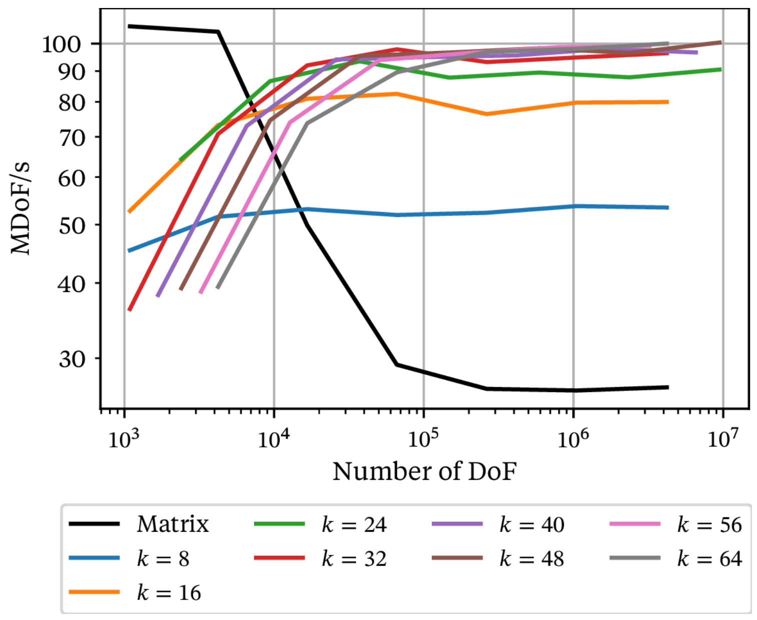

Figure 6.1: Performance of one operator application measured as MDoF/s (million DoF/s) on a 2D structured grid w.r.t. one core. For problem sizes, which exceed the L3 cache, the matrix-free application clearly surpasses the matrix-based one. After a short start-up period depending on the block size, the lowest block size kernel reaches a speed-up of 2×, while kernels with larger block sizes ($k \geq 32$) achieve up to 100 MDoF/s, which results in a speed-up of 3×.

Both plots illustrate the diametrical behavior of the matrix-free and matrix-based operator application. In the former case the DoF/s either increases with the number of DoFs or stagnates, while it decreases in the latter case. Because the matrix vector multiplication is memory bound, its performance depends strongly on the possible bandwidth for the data transfer. For tiny problems with #DoF $\sim 10^3$ the whole operator matrix and the necessary vectors fit into the L2 or L3 cache, which have a significant higher bandwidth than the main memory. Therefore, the matrix vector multiplication attains a high performance, even the highest measured performance. As the problem size increases, the caches cannot hold the entire problem anymore and the data needs to be transferred from the main memory instead, resulting in a decreasing bandwidth, until the final main memory bandwidth is hit at problem size of $\sim$ 50 000 DoFs. The final matrix-based performance after this point is around 30 MDoF/s in 2D and around 10 MDoF/s in 3D.

The behavior for matrix-free application significantly differs, since it first increases in most cases with the number of DoFs. For higher block sizes this is most pronounced and correlates to going from a macro grid with only one macro element to grids with multiple macro elements. The macro grid used in the later Navier-Stokes section has always more than one macro element and those benchmarks do not show this behavior, indicating that there is some significant assembly setup cost. Further investigations in that area seem reasonable. In 3D this becomes especially clear, since for the higher block sizes only 2 or 4 macro elements per direction are possible, if the total number of DoFs should be kept under 10^7. In those cases with more macro elements, the performance does not degenerate even for larger problems with more than 10^6 DoFs demonstrating that the operator applications are indeed not memory bound anymore. It is also worthwhile to point out that, at least in 2D and for lower block sizes in 3D, relatively small problem sizes are sufficient to achieve the DoF/s saturation point, around 10^4 DoFs. This could be beneficial in further studies concerning parallelism and strong scaling.

Besides the concrete realized performance, the qualitative behavior of the block-structured variants in 2D and 3D differ. The highest achieved performance in 2D corresponds to the highest block sizes $k = 56, 64$ with around 100 MDoF/s, although 90 MDoF/s are already reached with $k = 24$, which represents a speedup of ~3× compared to the matrix-based operator application. In contrast, the lower block sizes, $k = 16, 24$ realize the highest performance in 3D, around 25 MDoF/s. This corresponds to the increased amount of work per DoF by a factor ~2–3× from 2D to 3D, as the $k = 8$ variant achieves 50 MDoF/s in 2D. However, for higher block sizes the observed reduction seems to be too harsh. One reason could be the already mentioned assembly setup cost, but another potential reason could be inefficient caching. In 3D, it is necessary to keep one 2D plane of each local vector in the L1 cache to minimize the cache miss rate, if straight forward loop nests are used, which corresponds to $(k \cdot p + 1)^2$ values. For $k \geq 40$ this becomes problematic. These cache misses could be further reduced by introducing tiling, as mentioned in the outlook of chapter 5.

The next segment compares the performance of the block-structured variants with the theoretical estimates provided earlier. In particular, the program is split into a local kernel part and a transfer part, and the individual perfor-

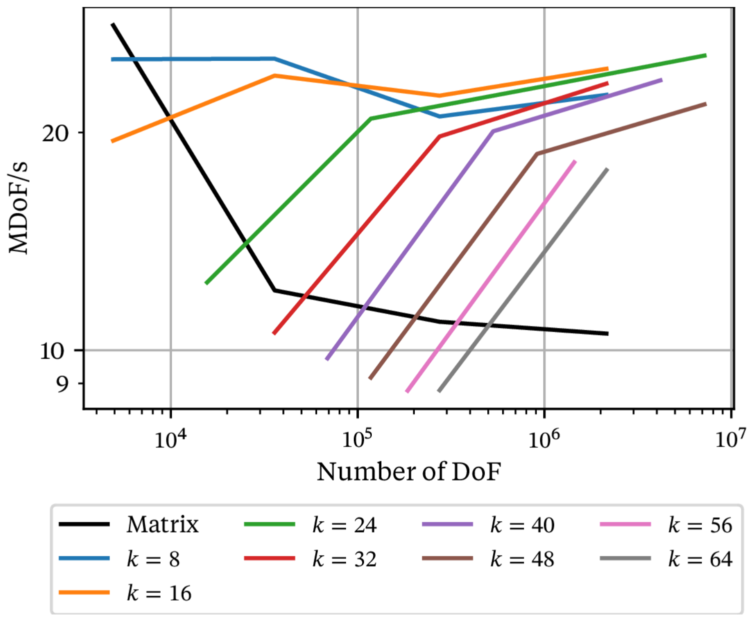

Figure 6.2: Performance of one operator application measured as MDoF/s on a 3D structured grid w.r.t. one core. Again, all matrix-free application surpass the matrix-based on for medium and large grid sizes. The ramp-up of the DoF/s performance is more prominent than in 2D, since for large block sizes the macro grid contains only one or four macro elements. Therefore, the kernels with smaller block sizes ($k \leq 32$) achieve the best performance of ~25 MDoF/s, which is ~2× higher than the matrix-based performance.

mance of each part is considered in more detail, i.e. the FLOP/s for the kernel, shown in Fig. 6.3, and the bandwidth, depicted in Fig. 6.4, for the transfer part. It should be noted that only the local kernel is measured separately, and the runtime of the transfer part is given by the difference of the total runtime and the local kernel runtime. Therefore, it is not possible to directly support the claim that the total runtime may be split as $T^{MF} = T^{MEM} + T^{COMP}$. If estimates for T^{COMP} were available, as suggested earlier, it would be possible to compare these estimate with the measured runtime to verify the claim.

In Fig. 6.3 the attained FLOP/s for each block size is plotted against the total number of DoFs both in 2D and in 3D. While the generated kernels in section 5.4.3 could reach the full LINPACK peak, this is not the case anymore. Still, the block-structured kernels reach a significant portion of the LINPACK peak, at least 50%, and usually more than 60%, which translates to between 30% and 40% of the theoretical FLOP/s limit. The difference to the previous benchmark might be caused by the loading of the coefficient, which is included in the local kernel timings. Compared to the DoF/s behavior, the performance does not

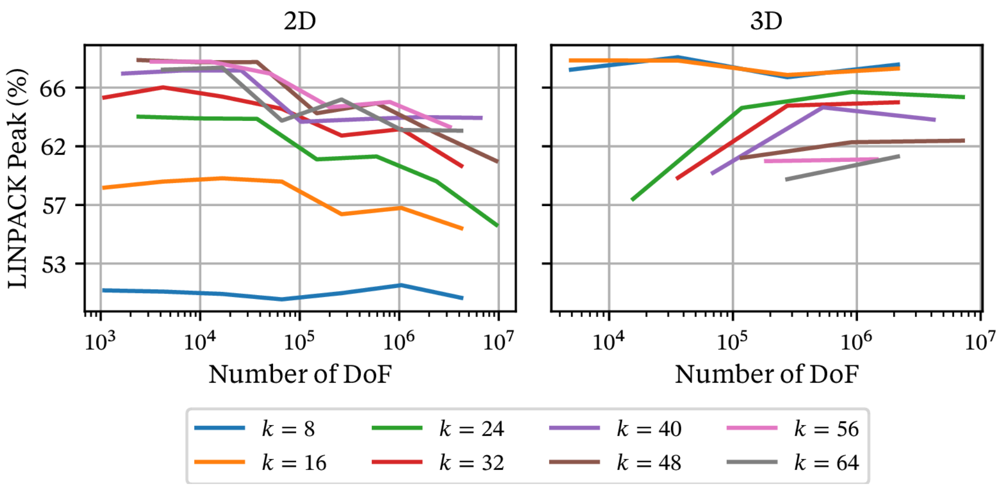

Figure 6.3: Performance of the compute-part of one operator application reported as % of LINPACK peak performance w.r.t. one core. 60% of LINPACK peak on one core corresponds to ~27.3 GFLOP/s. Kernels with higher block sizes ($k \geq 32$) achieve over 60% of the LINPACK peak performance both in 2D and 3D, while in 2D lower block size kernel exceed consistently the 50% mark. In 3D the higher performance of kernels with lower block size the DoF/s results is mirrored here.

increase significantly with higher problem sizes as it did for the DoF/s metric, which supports the assumption that there are some unoptimized setup costs. However, the performance degenerates slightly, about 10%, for higher problem sizes, but only in 2D. Since the decline is not noticeable in the DoF/s performance, other parts of the program seem to compensate for it.

The comparatively higher performance of the lower block sizes in 3D from the DoF/s plot is mirrored here. As the local computation uses more quadrature points, which increases the amount of work on data usually placed in registers, it is understandable that the FLOP/s performance increases for the block sizes $k = 8, 16$. Similar to the DoF/s case, the FLOP/s for higher block sizes in 3D could be increased by tiling the micro element loop to improve the data locality.

The achieved main memory bandwidth outside the local kernel is depicted in Fig. 6.4. As a reference, the bandwidth of the matrix-based operator application is also supplied, which is mostly identical to the STREAM peak performance. Two clearly distinct phases can be detected, first the bandwidth increases for each block size until a saturation point is reached, after which the full bandwidth is attained. These different phases stem from the measuring approach. By using performance counters, which measure only the transferred data volume from main memory, the bulk of the data used for smaller

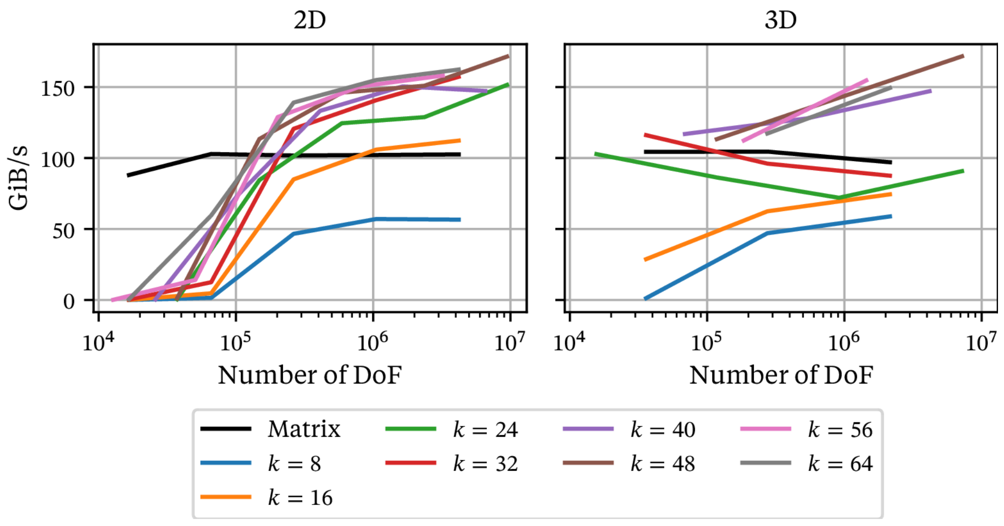

Figure 6.4: Performance of the memory-part of one operator application measured in GiB/s. In contrast to the other measurements in this section, the bandwidth is reported per NUMA domain and not per core, i.e. for 20 parallel programs, due to the measuring approach as explained in appendix A. The matrix-based application reaches consistently the STREAM peak of 100 GiB/s. After a start-up phase, which is less pronounced in 3D, the matrix-free methods attain their peak bandwidth. Only the lowest block size variant achieves significantly less than the STREAM peak, while higher block size variants even surpass this peak.

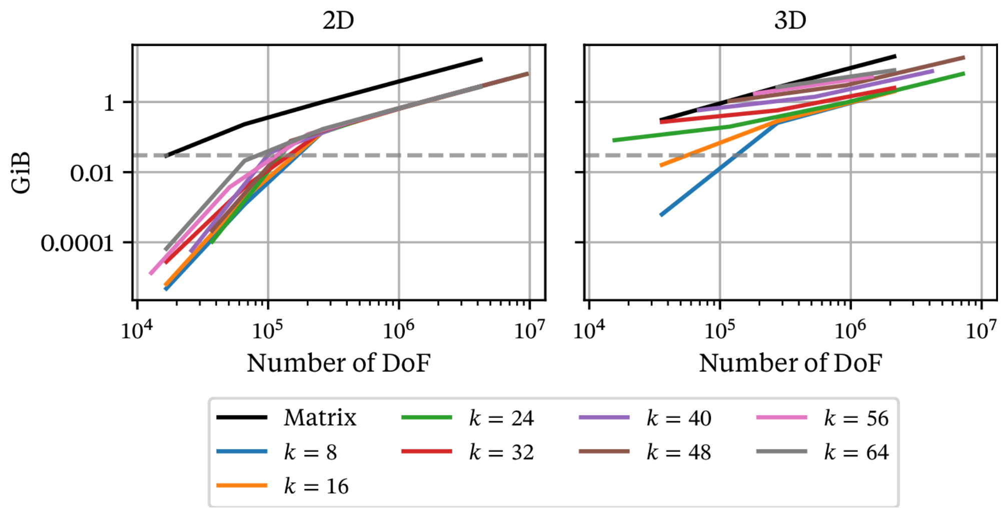

Figure 6.5: Total data volume transferred to and from the main memory over one operator application measured in GiB. The dotted line marks the size of the L3 cache (30 MiB), indicating after which point the whole problem does not fit into the caches anymore. The intersection with the dotted line marks the grid size after which the matrix-free applications reach their peak bandwidth.

problem sizes is ignored, since it resides within the caches and not the main memory, as can be seen in Fig. 6.5. Therefore, the saturation point corresponds to the smallest problem size for each block size at which the full problem needs to be moved from the main memory.

After the saturation point, a constant bandwidth is attained, where a higher block size results in a higher bandwidth. This can be explained by recalling that the gather and scatter becomes more efficient for higher block sizes, since the number of values per subentity with codimension less than d increases with the block size, and therefore more values can be transfer with a streaming access pattern in these cases, implying a higher efficiency. It should be noted that in multiple cases bandwidth higher than the STREAM benchmark are reached. Since this is an upper limit on the bandwidth, it is likely that some measured transfers happen during the local kernel timings. This indicates that the strict distinction in memory part and compute part in the theoretical estimates does not hold. Instead, some overlap has to modeled into the estimate. Nevertheless, this shows that the memory part of the block-structured variants can achieve optimal performance at least for higher block sizes, although there is still for improvement, especially in 3D.

Combining the results for the memory and computational part shows that the assumptions of the theoretical estimate are partially supported. Although using the LINPACK benchmark as part of the runtime estimate for the computational part is too optimistic, as already suggested, for the memory part the optimistic assumptions seem to hold. Furthermore, the generated code computes around 10–14 FLOP per DoF and per quadrature point, which is well below the established upper bound. Therefore, the estimate suggests that the matrix-free computation should be faster. The DoF/s measurements support the estimate, especially for larger problem sizes. All of these reinforce the theoretical estimate with some necessary adjustments, namely adapting the performance of the computational part and apply some overlap between both parts.

Solver Measurements

In the following, the performance of the solution process as a whole is considered, where at first, the most important measure in this regard, the time-to-solution, is shown in Fig. 6.6 and in Fig. 6.7. Later on, this is dissected into the necessary iterations until convergence, Fig. 6.8, and into the time per iteration, Fig. 6.9.

Fig. 6.6 shows the time-to-solution on one core, for the matrix-based vari-

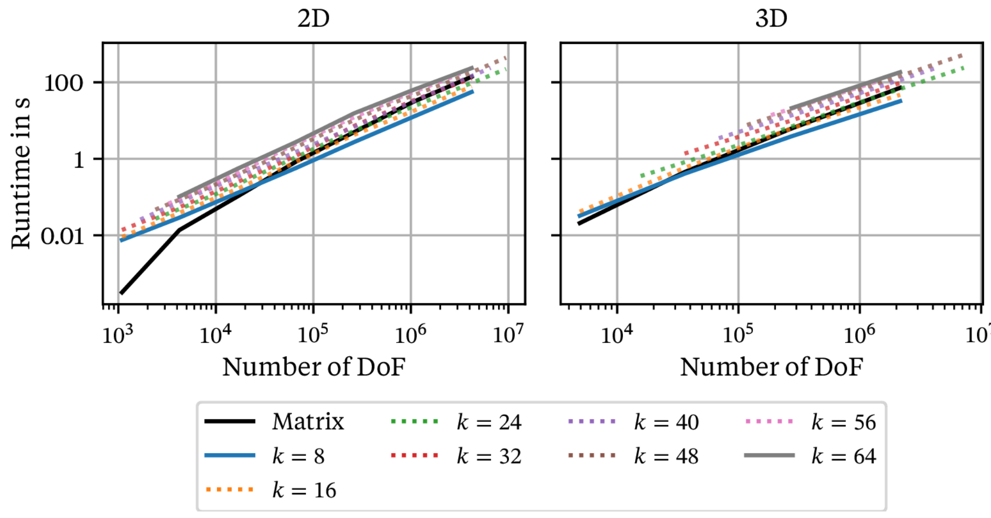

Figure 6.6: Time until linear solver convergence with a residual reduction of 10^{-10}. Some lines are dotted solely to improve visibility. For smaller grid sizes the matrix-based solver is faster than any matrix-free variant. On larger grids some matrix-free solvers surpass the matrix-based one. The significantly faster operator application can compensate for the slower convergence rate, which will be discussed later, at least for variant with low and medium block sizes ($k \leq 24$). There is no difference in the qualitative behavior of the 2D and 3D case.

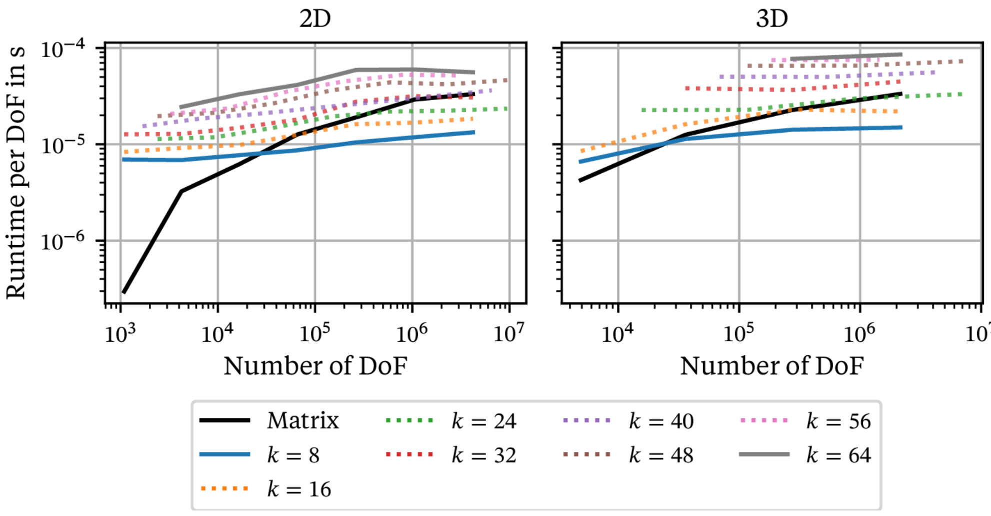

Figure 6.7: Time per DoF until linear solver convergence with a residual reduction of 10^{-10}. Again, some lines are dotted solely to improve visibility. This plot is a rescaling of the previous results to clarify the different convergence speeds for each block size. Thus, the same conclusions as before apply.

ants, as well as each matrix-free variant, and Fig. 6.7 shows the rescaled performance using DoF/s to increase the clarity. Compared to the matrix-based application, only the matrix-free solvers corresponding to the smaller block sizes are faster than the matrix-based solver. Additionally, the advantage only holds after a certain problem size, where that intersection point depends on the block size. The highest speedup of ∼2–2.5x is achieved by the $k = 8$ variant, while the worst slowdown of ∼0.5x is realized by $k = 64$. Additionally, a nearly linear increase in runtime can be observed, even in those cases where the DoF/s metric increased for larger block sizes, which will be examined in more detail as part of the iteration number and time per iteration discussions. All in all, the solution process benchmark illustrates that a faster operator application does not translate directly into a faster solution time. In the following, the reasons for this failure are examined.

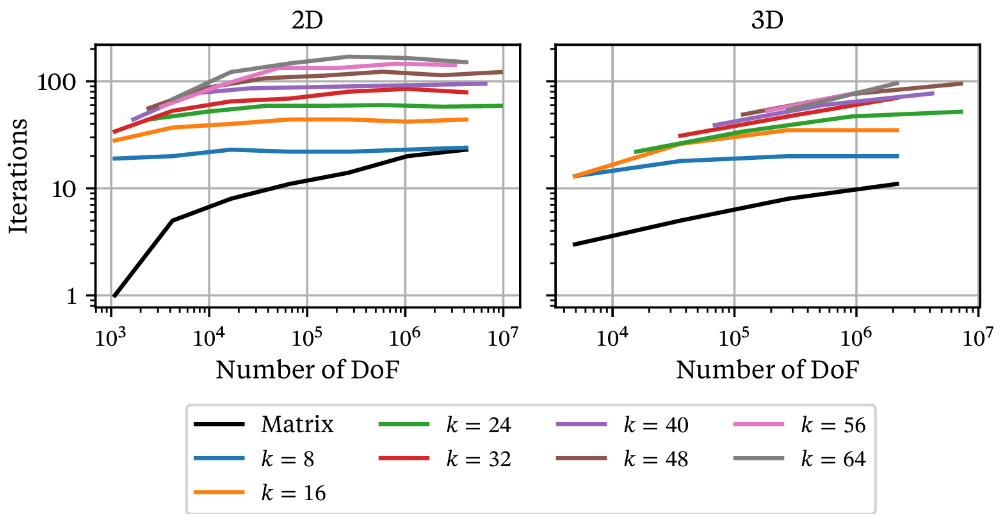

Figure 6.8: Number of iterations until convergence of the linear solver. All matrix-free solver converge slower than the matrix-based solver, where the difference increases with the block size. The largest block sizes require about 10× more iterations until convergence. Only the convergence speed of the variant with $k = 8$ on the largest grid comes close to matrix-based one.

The main culprit for the high time-to-solution is the number of iterations necessary for convergence, presented in Fig. 6.8. Both in 2D and in 3D, the matrix-free block-structured solvers require more iterations than the conventional solver with the matrix-based AMG preconditioner. Furthermore, the linear dependency of the iteration number on the block size, as mentioned in section 4.3, is clearly noticeable. This dependency results in 7–9× higher iteration number for the highest block sizes compared to the matrix-based computation.

Since the convergence rate is mainly determined by the preconditioner, this underlines the necessity for better preconditioners in the matrix-free case to improve upon the state-of-the-art matrix based approach. Nevertheless, it is remarkable that the time-to-solution difference is significantly lower than the iteration count difference, which can be explained by examining one iteration in detail next up.

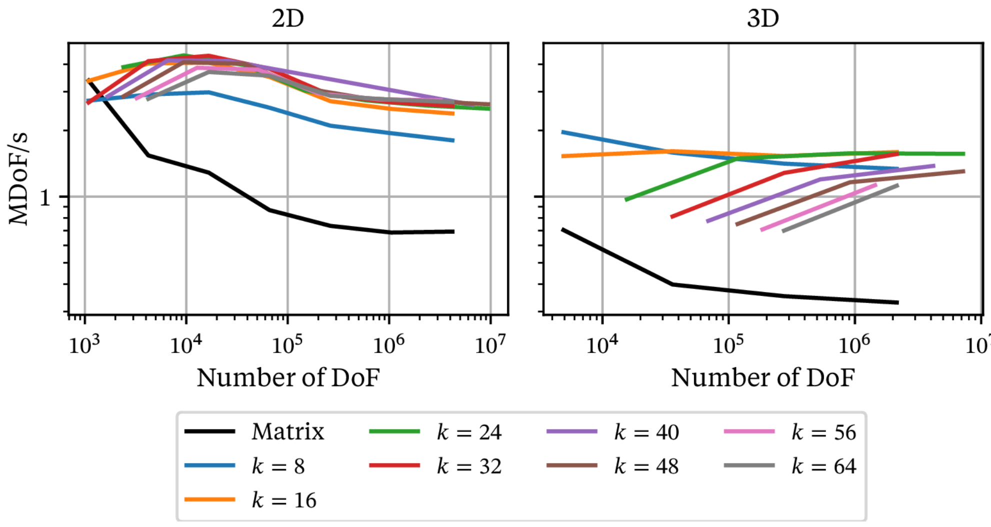

Figure 6.9: Performance of one full linear solver iteration, measured in MDoF/s. The DoF/s performance behavior for a full iteration closely resembles the DoF/s performance of the operator application. The benefit of the matrix-free variants is more noticeable than in the operator application, since the preconditioner application is also incorporated.

The performance of one solver iteration is illustrated in Fig. 6.9 using again the DoF/s metric to reveal similarities to the operator application. Since one BiCGStab iteration contains two preconditioner applications, multiple axpy operations, and scalar products, besides two operator applications, the total DoF/s is lower than in the operator application case. Nevertheless, the qualitative behavior is clearly mirrored here. The AMG preconditioner employed in the matrix-based case is quite expensive, such that the matrix-free variants achieve higher DoF/s, around 2–5x the matrix-based DoF/s, even where that is not the case for the operator application. This indicates that improving the preconditioner to reach the convergence rate of the AMG method will significantly speed-up the time-to-solution up to 5× in the best case.

6.2.2 p-Laplacian

In the following section, the p-Laplace equation is considered in order to determine the advantages or disadvantages the matrix-free block-structured grids approach would convey in a non-linear setting. This simple non-linear extension of the Poisson equation defined as

$$\begin{aligned} -\nabla \cdot \left(|u|^{p-2}\nabla u\right) &= f \text{ in } \Omega, \\ u &= g \text{ on } \partial\Omega, \end{aligned}$$

with $1 < p < \infty$. The parameter $p = 3/2$ is chosen here. The right-hand-side is defined as

$$f(x, y) = 1 + \cos(2\pi x)\sin(2\pi y),$$

and homogenous Dirichlet conditions are used. The domain and the element geometry is the same as in the Poisson benchmark, i.e. $\Omega = [0, 1]^2$ with a structured mesh, which allows focusing on the new aspects brought in by the non-linearity.

The discretization, using $\mathbb{Q}_1$ finite elements, leads to the non-linear functional

$$r(u, v) = \int_\Omega \left(\varepsilon + |u|^2\right)^{-\frac{1}{4}} \nabla u \cdot \nabla v - f v \,\mathrm{dx},$$

where the p-Laplacian is regularized with $\varepsilon = 10^{-10}$ to handle the singularity at $u = 0$. Both the handwritten matrix-based variant and the generated matrix-free variants solve the corresponding non-linear equation $R(\mathbf{z}) = 0$ with Newton's method, where the linearized equation $DR(\mathbf{z})\delta\mathbf{z} = -R(\mathbf{z})$ is solved using the setup described at the beginning of this section. The implementation of the Newton method available in DUNE-PDELAB automatically chooses a target reduction for the linear solver such that quadratic convergence can be guaranteed. Therefore, the linear systems are usually solved with a lower precision than in the previous benchmark. Unfortunately, the Newton implementation in DUNE-PDELAB 2.7 does not support matrix-free application, but a patch for it is provided in the custom branch mentioned in appendix A.

Due to the evaluation of the non-linear coefficient at each matrix-free operator application, the local amount of work is significantly higher than during the linear Poisson benchmark. To be more precise, around 300 FLOP per DoF are executed during one operator application, 7.5× the work of the linear Laplace operator. Additionally, the theoretical FLOP bound is even a bit lower for the non-linear Laplace operator due to the additional linearization point coefficient $n_c = 1$. Assuming best cache reuse of the matrix-vector multiplication, the upper FLOP bound is 224 FLOP per DoF, and for no cache reuse, which is unlikely on structured grids, the FLOP bound is 368. Therefore, even if the LINPACK performance can be achieved in the local kernels, the matrix-free operator application will be slower than the matrix-based one. The DoF/s performance in Fig. 6.10 clearly supports this theoretical finding.

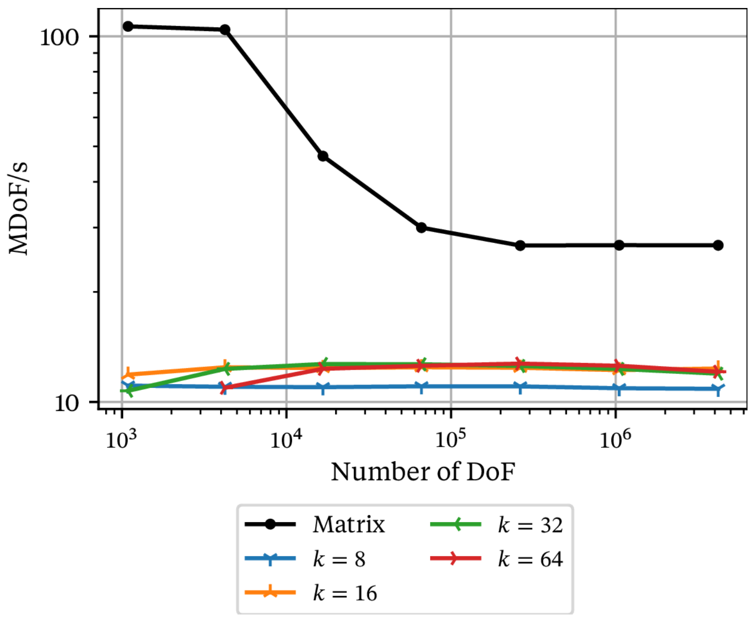

Figure 6.10: Performance of one operator application measured as MDoF/s (million DoF/s) on a 2D structured grid w.r.t. one core. The matrix-based operator performance is the same as in the previous benchmark. Due to the evaluation of the non-linearity, the matrix-free performance is considerably lower than previously, and also lower than the matrix-based variant.

Although the matrix-free variants cannot be faster than the matrix-based variants comparing only the operator application, this is not necessarily the case for the whole solution process. During each Newton step, a new linear system has to be solved, which requires assembling a new matrix in each step. It should be noted that this is only required to achieve the best convergence, otherwise techniques to reduce the number of matrix assemblies could be employed, outlined for example in [28]. However, to guarantee the same convergence behavior as in the matrix-free case, these techniques are not considered here. This matrix assembly step takes notably longer than the matrix application and when taking this setup cost into account the matrix-based operator

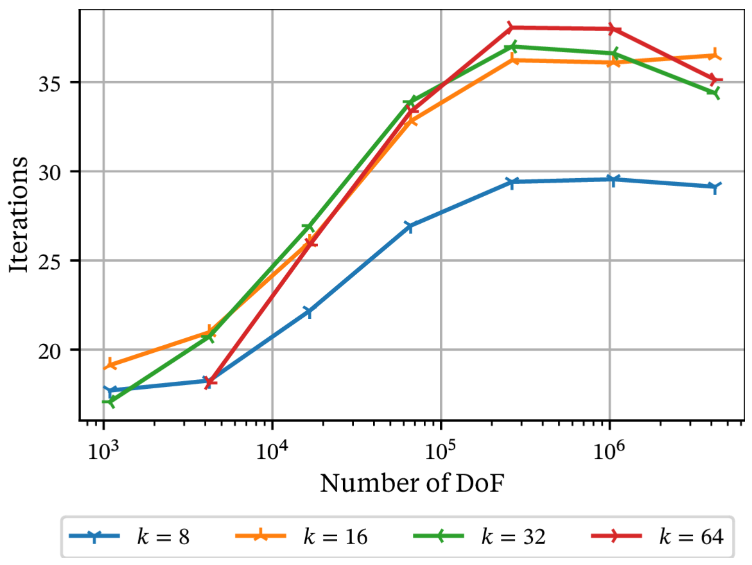

Figure 6.11: The number of operator applications until the matrix-based variant with the matrix assembly setup cost included becomes faster than the matrix-free variants. For medium and larger grid sizes, the assembly of the stiffness matrix is as expansive as 30–35 applications of the matrix-free operator.

application is slower than the matrix-free variants for the first 30–35 applications, as can be seen in Fig. 6.11.

However, this cannot be directly transferred to the linear solver iterations, since the preconditioner cost is not considered. If the two-level preconditioner described before with three Jacobi iterations on the fine grid is used, the solver iterations advantage for the matrix-free variants reduce to 5–6 iterations in the worst case, i.e. the matrix-based preconditioner is just as fast as the matrix application. Nevertheless, there is a certain number of iteration, which could be considered 'free' for the matrix-free variants. As long as the matrix-free variants converge within this iteration number, it does not matter if the matrix-based operator application is significantly faster or if the matrix-based linear solver converges earlier, the matrix-free solver will be faster.

This effect of the large setup cost is barely notable in this benchmark due to the slow convergence of the matrix-free linear solver, which can be seen in Fig. 6.12. After a slight increase in the linear solver iterations per Newton step, the matrix-free variants with $k \geq 16$ require more than 40 iterations, or even more than 100 iterations, on average, surpassing the 10 iterations of matrix-based linear solver by a significant amount. The setup cost is amortized already after the first 5–6 combined matrix-free operator and preconditioner applica-

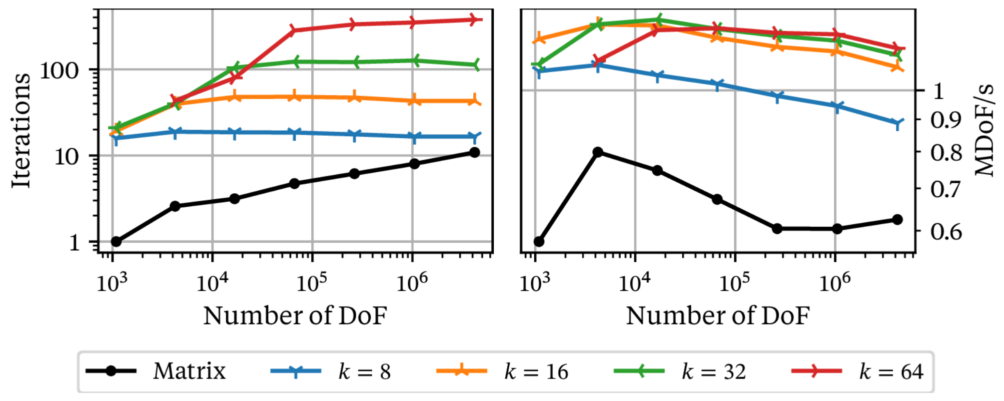

Figure 6.12: Left, average number of iterations until the linear solver converges with the accuracy prescribed by the Newton solver within one Newton step. Right, the performance for one linear solver iteration as MDoF/s, including the assembly cost. The convergence behavior of the linear solver is nearly identical to the behavior observed during the Poisson benchmark. Although the operator application of the matrix-free variant is slower, considering one linear solver iteration they are faster than the matrix-based variant, due to the reduced setup cost.

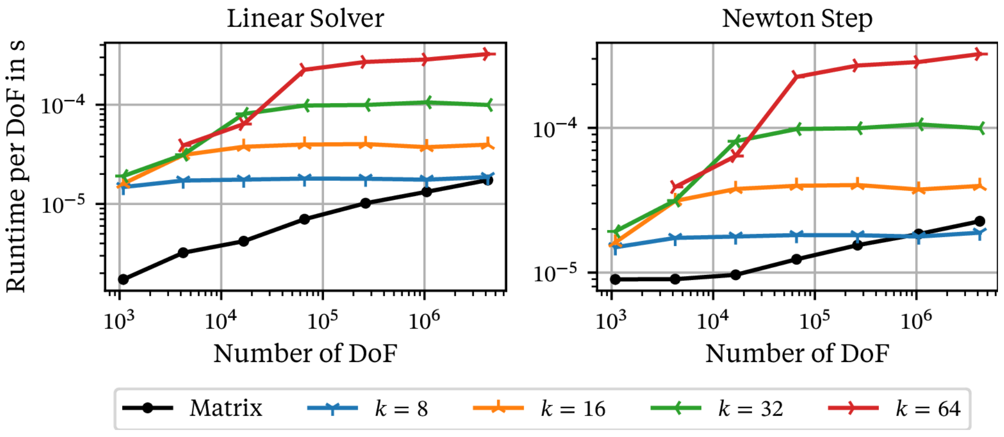

Figure 6.13: Left, the runtime per DoF of both one full linear solve. Right, the runtime per DoF for one single Newton step, including additional setup cost and the line search besides the linear solve. The cost of the assembly is also included in the runtime of the linear solve for the matrix-based variant. The matrix-free linear solver is never faster than the matrix-based one due to the higher number of required iterations and the slow operator application. Nevertheless, in few cases the runtime per Newton step is lower in the matrix-free case. The full Newton step contains additional costs, which impact the matrix-based computation more than the matrix-free ones.

tions, which translates to 2–3 BiCGStab iterations, since two applications are executed per iterations. Therefore, the gap in the remaining linear solver iteration cannot be closed, even with a faster runtime per iteration, shown in Fig. 6.12. In the instances where the matrix-free linear solver comes close, $k = 8$ on larger grids, see Fig. 6.13, some influence of the setup cost can be recognized. After removing the 'free' iterations, gained from skipping the matrix assembly, the matrix-free linear solver converges after ~15 additional iterations. Therefore, the difference to the matrix-based variant is reduced, but the reduction is still not enough to surpass the matrix-based performance. If the additional cost of a Newton step besides the linear solve, such as line search, that require multiple residual evaluation, the matrix-free variant becomes slightly faster, see Fig. 6.13. This shows that removing the assembly cost can give the required edge to outperform the matrix-based computation.

6.2.3 Navier-Stokes Model

The final benchmark considers the incompressible Navier-Stokes equations, defined as

$$\begin{aligned} \rho\left(\frac{\partial u}{\partial t} + u \cdot \nabla u\right) &= \mu \Delta u - \nabla p + f, \\ \nabla \cdot u &= 0, \end{aligned}$$

to simulate a channel flow around a cylinder in 2D. The problem parameters are chosen according to the DFG 2D-3 benchmark [84], leading to the domain $\Omega = [0, 2.2] \times [0, 0.41] \setminus B_r(0.2, 0.2)$ with $r = 0.05$, a density of $\rho = 1$, a dynamic viscosity of $\mu = 0.001$, and no source term, i.e. $f = 0$. On the top and bottom walls $\Gamma_B = [0, 2.2] \times \{0\}$, $\Gamma_T = [0, 2.2] \times \{0.41\}$, as well as the boundary of the cylinder $\Gamma_C = \partial B_r(0.2, 0.2)$ no-slip boundary conditions are prescribed, i.e.

$$u|_{\Gamma_T} = u|_{\Gamma_B} = u|_{\Gamma_C} = 0.$$

The left wall $\Gamma_I = \{0\} \times [0, 0.41]$ acts as an inflow boundary with the following inflow profile

$$\begin{aligned} u(0, y, t) &= \begin{pmatrix} \frac{4U(t)y(0.41 - y)}{0.41^2} \end{pmatrix}, \quad y \in [0, 0.41] \\ U(t) &= 1.5 \sin\left(\frac{\pi t}{8}\right), \end{aligned}$$

and a do-nothing outflow together with a homogenous Dirichlet condition on the pressure is defined on the right wall $\Gamma_O = \{2.2\} \times [0, 0.41]$ leading to

$$\begin{aligned}\partial_\eta u &= 0 \quad \text{on } \Gamma_O,\\ p &= 0 \quad \text{on } \Gamma_O.\end{aligned}$$

On the other boundary parts, homogenous Neumann conditions are applied to the pressure.

These equations are solved with the Chorin projection using the explicit Euler time stepping method, which allows splitting the initial equation into three separate steps. An overview of the Chorin projection, as well as other projection methods, can be found in the review [49]. Let u^n be the current approximation of the velocity, p^n the corresponding pressure approximation, and τ the time step size. Then, the next step is computed by first determining an intermediate velocity u^* with

$$\begin{aligned}\rho\frac{u^* - u^n}{\tau} &= -\rho(u^n \cdot \nabla)u^n + \mu\Delta u^* \quad \text{in } \Omega,\\ u^* &= g \quad \text{on } \Gamma_B \cup \Gamma_T \cup \Gamma_C \cup \Gamma_I,\\ \partial_\eta u^* - p^n\eta &= 0 \quad \text{on } \Gamma_O,\end{aligned}$$

where η is the unit outer normal on Γ_O. Afterwards a pressure correction is computed by solving for the new pressure

$$\begin{aligned}\Delta p^{n+1} &= -\frac{\rho}{\tau}\nabla \cdot u^* \quad \text{in } \Omega,\\ p^{n+1} &= 0 \quad \text{on } \Gamma_O,\\ \partial_n p^{n+1} &= 0 \quad \text{on } \Gamma_B \cup \Gamma_T \cup \Gamma_C \cup \Gamma_I,\end{aligned}$$

and then correcting the intermediate velocity by

$$\rho u^{n+1} = \rho u^* - \tau\nabla p^{n+1}.$$

Using a finite element discretization leads to the following three problems, the intermediate problem

$$\frac{\rho}{\tau}\langle u^*, v\rangle + \mu\langle\nabla u^*, \nabla v\rangle = \frac{\rho}{\tau}\langle u^n, v\rangle - \rho\langle(u^n \cdot \nabla)u^n, v\rangle \quad \forall v \in V_D^u,$$

the pressure problem

$$\langle \nabla p^{n+1}, \nabla q \rangle = \frac{\rho}{\tau} \langle \nabla \cdot u^*, q \rangle \quad \forall q \in V_D^p,$$

and the projection problem

$$\rho \langle u^{n+1}, v \rangle = \rho \langle u^*, v \rangle - \tau \langle \nabla p^{n+1}, v \rangle \quad \forall v \in V^u,$$

where V^u is the finite element space for the velocity, without constraints, V_D^u the same space with the specified Dirichlet constraints, and V_D^p the finite element space for the pressure with Dirichlet constraints.

The handwritten and generated implementations use both a stable finite element, the $\mathbb{Q}_2$-$\mathbb{Q}_1$ Taylor-Hood pair, and an unstable $\mathbb{Q}_1$-$\mathbb{Q}_1$ element. The sparse matrix from the discretized intermediate problem uses blocking of the velocity unknown at each grid node, resulting in a matrix with 2×2 blocks as entries. Blocked matrices have more efficient the operator application, as the arithmetic intensity increases to 1/4 in the limit. Since the vectorization of the $\mathbb{Q}_2$ element is not supported, only the local kernels using the unstable element are vectorized. In either case, both the handwritten matrix-based and the generated matrix-free variant apply mass lumping to the projection problem, reducing it to assembling the right-hand-side and scaling it. Therefore, the projection step is also effectively matrix-free for the handwritten variant.

These settings for the Navier-Stokes benchmark explore several aspects not covered in the previous benchmarks. First, the particular choice of the domain necessitates the usage of an unstructured grid to account for the interior circular boundary. In contrast, the other benchmarks used an axiparallel, equidistant grid, which is more favorable for any matrix-free computation. The usage of multilinear geometries automatically increases the total amount of work per DoF, through an increased quadrature order to stay accurate and the additional FLOPs to compute the geometry quantities. Furthermore, the projection step closely resembles one step of an explicit time stepping method, which allows estimating how the block-structured approach would compare against a non block-structured approach for solving a time dependent PDE with an explicit method.

The performance discussion is analogous to the previous sections, i.e. the performance of one single operator for each step, and the performance of the full intermediate and pressure solver are considered separately. As a first step, some predictions on the following performance are provided, based on the theoretical estimates from the beginning of the chapter. Table 6.1 presents the

	Intermediate	Pressure	B^i	B^p
$\mathbb{Q}_2$-$\mathbb{Q}_1$	323.75	222.19	600.89	384.00
$\mathbb{Q}_1$-$\mathbb{Q}_1$	198.73	206.05	456.00	384.00

Table 6.1: Comparing the measured work (FLOP) per micro element for the intermediate and pressure operator for both element types on the left with the corresponding theoretical upper bounds $B^\bullet$ on the right side.

worst-case FLOP bound estimates ($\alpha = 1$) and the measured data. Due to the higher quadrature order and the higher local polynomial degree for the velocity part of the stable element, the measured work per DoF is significantly higher than in the Poisson benchmark, which uses similar local kernels. Since the estimated FLOP bounds are reported w.r.t. the LINPACK peak performance, the matrix-free approach cannot be faster than the matrix-based operator application if less than 50% of that peak performance is achieved, except in the unstable case for the intermediate operator application where $\sim$40% are sufficient. The FLOP bounds are worsened further by the fact that the local kernels are not vectorized for the stable element. In that case, only 1/8th of the LINPACK performance can be realized, indicating that the matrix-based variant will be faster.

Operator Performance

In Fig. 6.14 the DoF/s of the operator application is presented for each step and each element type. Considering the pressure and intermediate step, the performance behaves as expected. Compared to the initial Poisson benchmark no variance w.r.t. the grid size is notable, as the coarse grid always consists of multiple macro elements and only lower block sizes are used. Due to the missing vectorization with the stable element, the matrix-free variants fails to reach a significant portion of the LINPACK performance. Thus, it falls well below the matrix-based operator application performance. By switching to an unstable element and utilizing vectorization for larger block sizes, the matrix-free operator application can catch up. The matrix-free performance does not surpass the matrix-based one by a large amount, which suggests that the achieved FLOP/s performance of the local kernels is close to the 50% LINPACK bound mentioned earlier. Since the projection step in the handwritten non block-structured variant is also computed matrix-free, the qualitative comparison differs from the other steps. In this case, the block-structured right-hand-side assembly is vastly faster. Without vectorization speed-ups between 4× for the

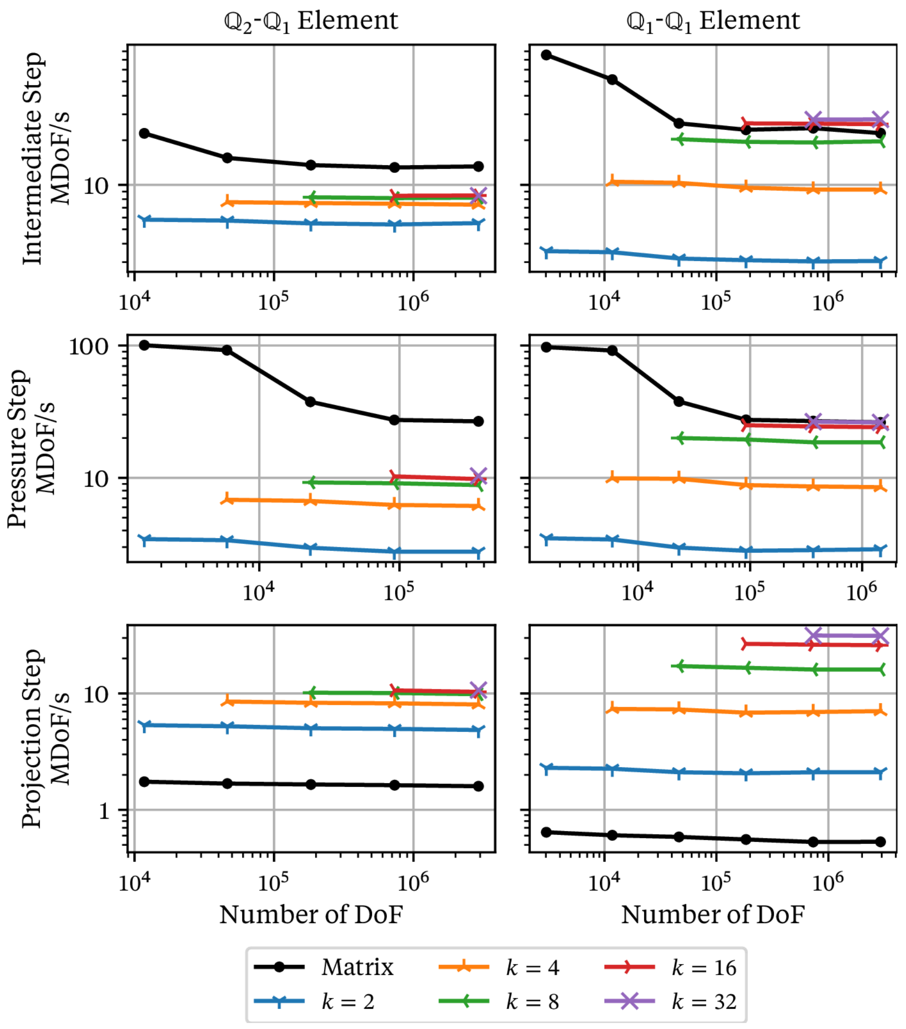

Figure 6.14: Performance of one operator application measured as MDoF/s for each of the three steps defined above and for both element types. Although the handwritten variant is denoted as 'Matrix', the projection step is computed matrix-free also in that case, since only the assembly of the right-hand-side is considered. Due to missing vectorization, the matrix-free performance of the intermediate and pressure step is significantly lower than the matrix-based one for the stable element. If vectorization is possible with the unstable element, the matrix-free variants with high block sizes reach or even surpass the matrix-based performance. The matrix-based computation during the projection step is identical to a matrix-free computation with $k = 1$. Therefore, in this step each matrix-free variant clearly exceeds the matrix-based performance.

smaller block size ($k = 4$) and 5× for larger block sizes can be achieved, while with vectorization speed-ups between 10–50× can be achieved.

In the following, the performance of the data transfer and the compute-part are examined, with similar findings as in the previous benchmarks. First off, Fig. 6.15 shows FLOP/s performance relative to the LINPACK peak w.r.t. one core. Again, due to the higher amount work per macro element and the increased number of macro elements, there is no variation due to the grid sizes. Without vectorization only 20% of the LINPACK benchmark is reached, illustrating the cause of failing to reach the DoF/s of the matrix-based operator application. If vectorization is enabled by using the unstable finite element pair and AVX-512 instructions are possible, between 35%–55% of the peak are achieved for block sized $k \geq 8$. For multilinear geometries, the local kernels becomes more complicated than in the previous models with longer dependency chains, which results in lower performance compared to previous benchmarks. Curiously, the performance of the projection step is lower than the intermediate step, although both steps have the same loop nests. The projection step requires loading additional data for the coefficients to compute the residual of the projection step, which are transferred as part of the local kernel and are not excluded form the timings, thereby increasing the local kernel time without adding any FLOPs.

The attained bandwidth for the data transfer part, or the whole operator application in the matrix-based case, is presented in Fig. 6.16. The assembled matrix application continues to achieve the STREAM benchmark bandwidth. Matrix-free variants reach the benchmark's bandwidth only in cases where the amount of DoFs associated with the macro element interior is high enough, i.e. $k \geq 8$ for the stable elements and $k \geq 16$ for the unstable elements. Since the stable elements use the $\mathbb{Q}_2$ element with a higher volume to surface DoFs ratio, these variants achieve higher bandwidths compared to the unstable element pair. As in the benchmarks before, some matrix-free measurements are higher than the STREAM benchmark. This supports the previous conclusions, namely that matrix-free variants with many interior DoFs can fully exploit the main memory bandwidth and the overlap between memory and computation part is larger than initially assumed.

Solver Performance

Next up, the whole solution process is examined following the procedure of the previous benchmarks. Since this model is an instationary problem, the solution process would normally refer to computing a solution for each time

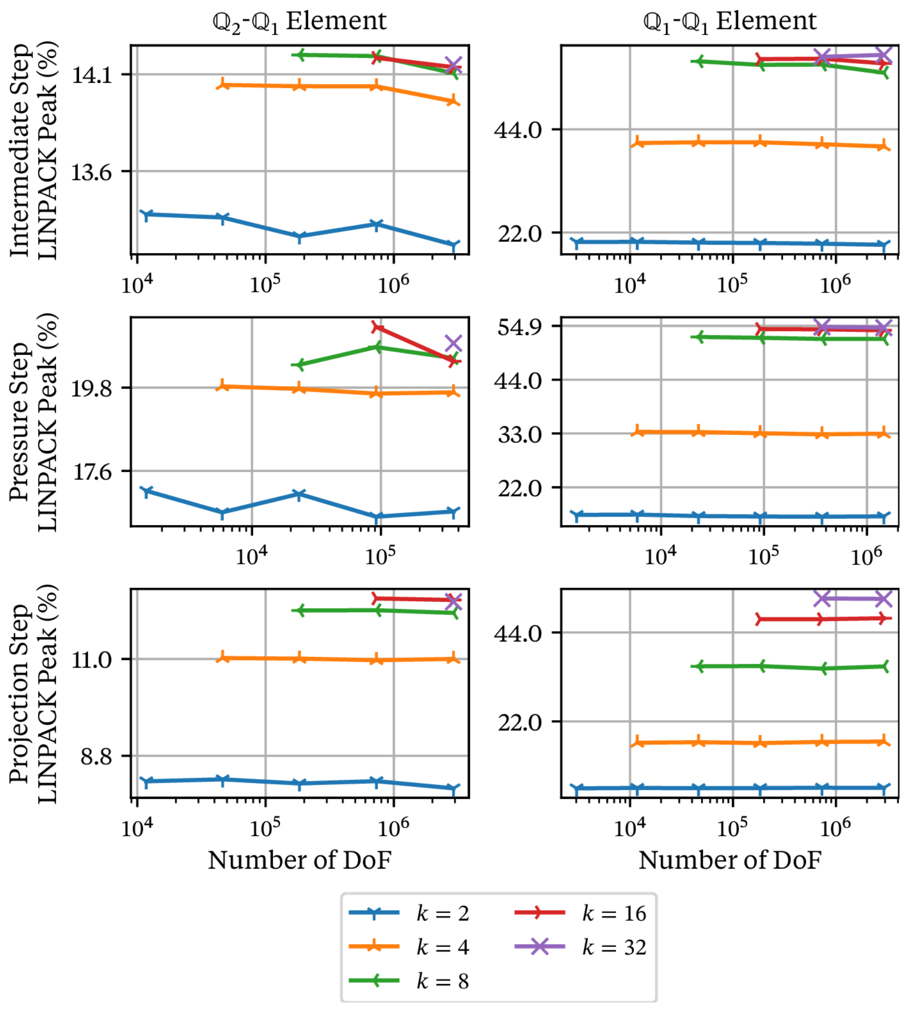

Figure 6.15: Performance of the compute-part of one operator application reported as % of LINPACK peak performance w.r.t. one core. 55% of LINPACK peak on one core corresponds to ~25 GFLOP/s. Due to the current restrictions of the code generator, only the local kernels for the unstable element can be vectorized. Without vectorization, the matrix-free kernels cannot achieve any significantly portion of the LINPACK peak. Using the unstable element with vectorized local kernels, 35–55% of LINPACK peak are attained.

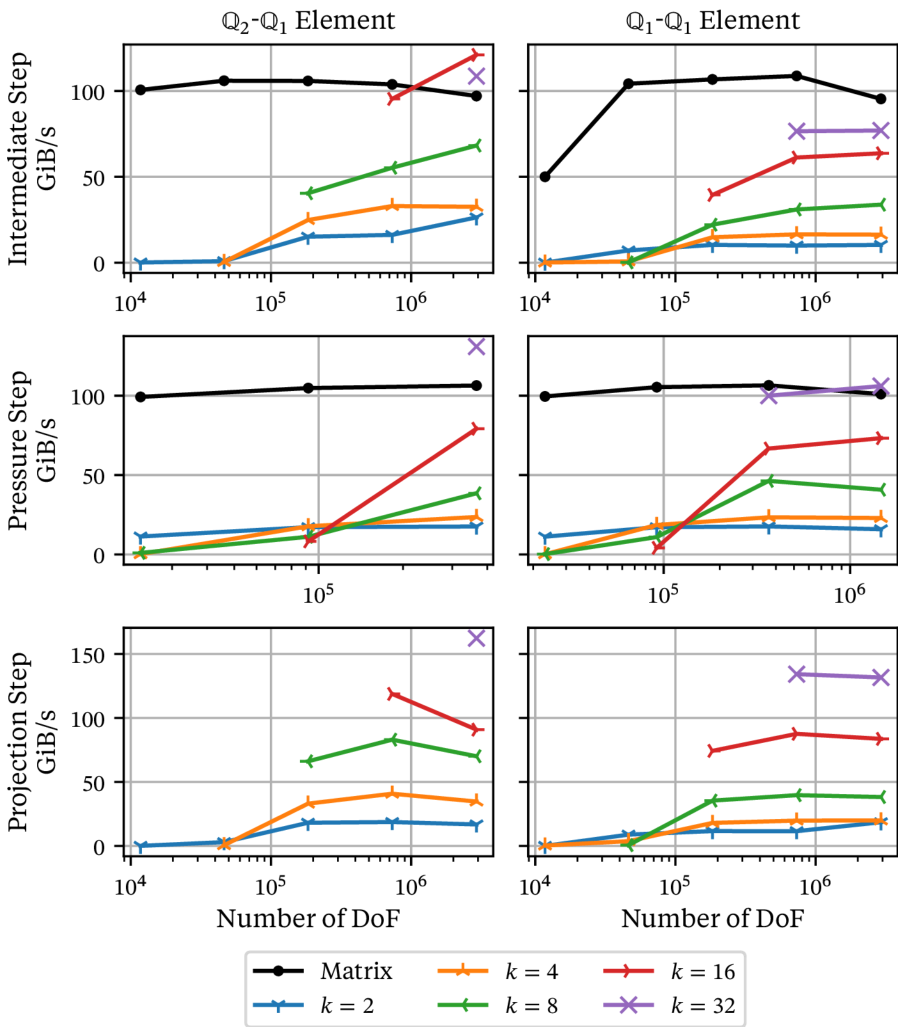

Figure 6.16: Performance of the memory-part of one operator application measured in GiB/s. As before, the measurements are w.r.t. one NUMA domain. Since the otherwise matrix-based computation is matrix-free for the projection step and there are no separate kernel timings available in this case, the measurements for that case are dropped. The computations using the $\mathbb{Q}_2$ velocity element achieve a higher bandwidth than their $\mathbb{Q}_1$ element counterparts. Similar to the bandwidth results for the Poisson model, the variants with higher block sizes reach higher bandwidths, again surpassing the STREAM bandwidth.

step on a fixed time interval, possibly using adaptive time stepping. For performance considerations this is not useful. Instead, it is sufficient to investigate the performance of one time step, which includes the solution of two linear system and one lumped system. Since solving the lumped system is equal to the, already discussed, assembly of the residual, this system is not considered further. Only the system for the intermediate velocity and the new pressure are considered. As for all measurements, one time step is repeated until the total runtime of each solution is larger than 0.5s.

The total runtime for one time step is given in Fig. 6.17 and the runtime per DoF is shown in Fig. 6.18. As expected from the mostly slower operator application, the matrix-free solution process is generally slower than the matrix-based variants, only in one case, the intermediate solver using unstable elements and $k = 8$ without any additional grid refinement, the matrix-free computation is slightly faster. Considering the runtime per DoF, the runtime does not scale linearly with the number of DoFs anymore, or at least the runtime per DoF did not level off during the used grid sizes. Examining the number of solver iterations in Fig. 6.19 and the performance per solver iteration in Fig. 6.20 illuminates this behavior.

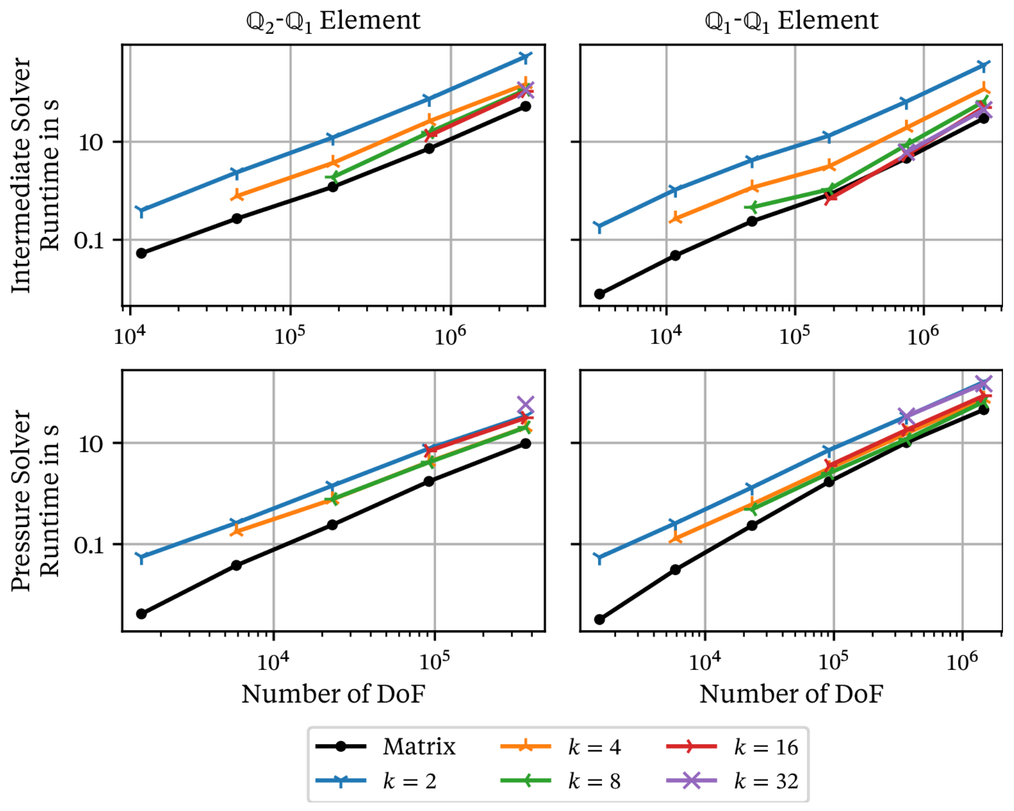

Figure 6.17: Runtime until linear solver convergence with a residual reduction of 10^{-10} for the intermediate and pressure step and both element types. The matrix-based variant is faster for virtually all grid sizes. The marginally faster operator application for some matrix-free variants is not enough to compensate for the slower convergence, discussed later on.

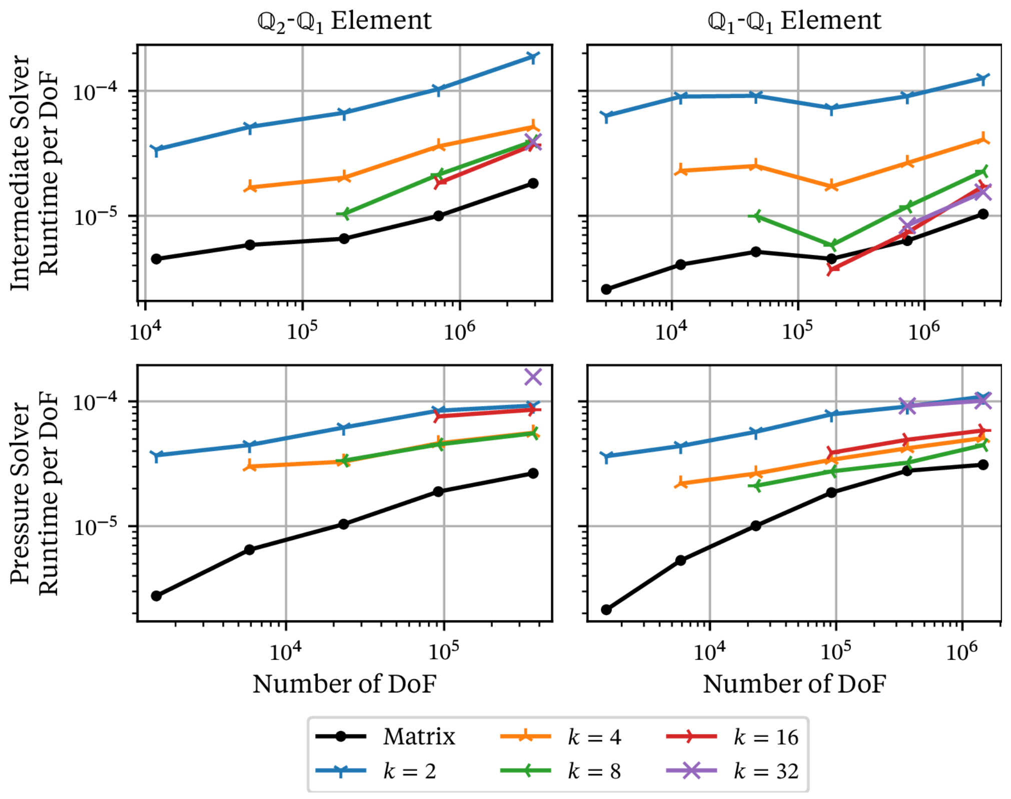

Figure 6.18: Runtime per DoF until linear solver convergence with a residual reduction of 10^{-10} for the intermediate and pressure step and both element types. This is a rescaling of the results from Fig. 6.17. It becomes clear that in only one case, $k = 8$ and a grid size of $\sim 2\times 10^5$, the matrix-free solver exceeds the matrix-based.

The iteration number, as shown in Fig. 6.19, grows slightly with the grid size for the pressure solver, which is in agreement with the Poisson benchmark, since in both instances the same assembled matrix is considered. Besides the grid size scaling, the matrix-free variants scale again linearly with the block size, leading to high iteration numbers for high local refinement cases, roughly 10× more than the matrix-based iteration number. For the intermediate system, both the absolute iteration numbers and the block size scaling are significantly reduced, most likely due to the mass matrix. As a result, the difference in iterations between matrix-free and matrix-based variants reduces to a factor of ∼5–7×. Additionally, the scaling due to the grid size seems to be increased. Considering the lowest block size, which also uses the most macro elements, the iteration numbers seem to level off, suggesting that the iteration numbers for the higher block sizes might also flatten for larger grid sizes.

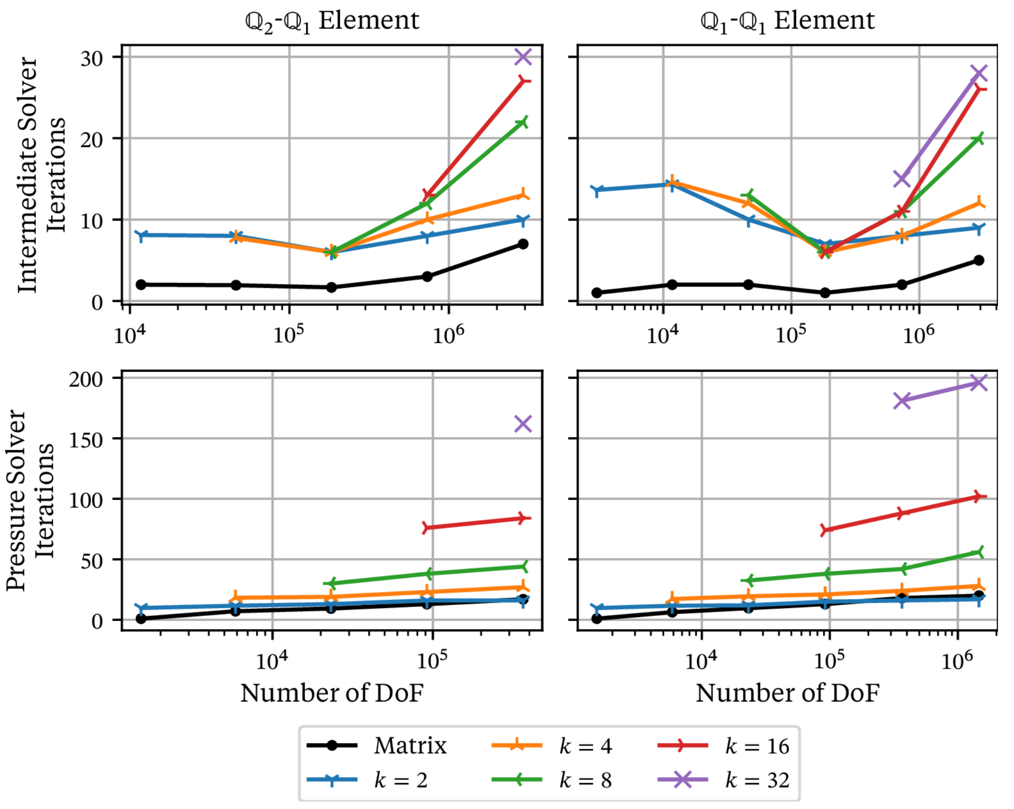

Figure 6.19: Average number of iterations of each linear solver for one time step. For the intermediate step some ramp-up of the required iteration number can be seen for higher block sizes. Similar to the earlier iteration number discussions, the matrix-based solver converges faster than most matrix-free solver. Only very small block sizes ($k = 2, 4$) can lead to fewer required iterations. In the worst case, the slowest matrix-free variant ($k = 32$) requires ∼10× as many iterations as the matrix-based variant for the pressure step, and ∼5–7× more iterations for the intermediate step.

Similar to the previous benchmarks, the DoF/s of one linear solver iteration for the matrix-free variants ($k \geq 8$) exceeds the matrix-based performance, due to the less expansive preconditioner application. Fig. 6.20 depicts this performance. A roughly 2× higher performance than the matrix-based variant is achieved by the matrix-free variants during the intermediate step in the best case. During the pressure step, this increases slightly to a speed-up of ∼3×. There are some clear dips in the matrix-based performance, because the linear solver converges immediately after the first preconditioner application, resulting in a runtime dominated by setup costs. The performance of the matrix-free variants degenerates with increasing grid sizes, especially for lower block-

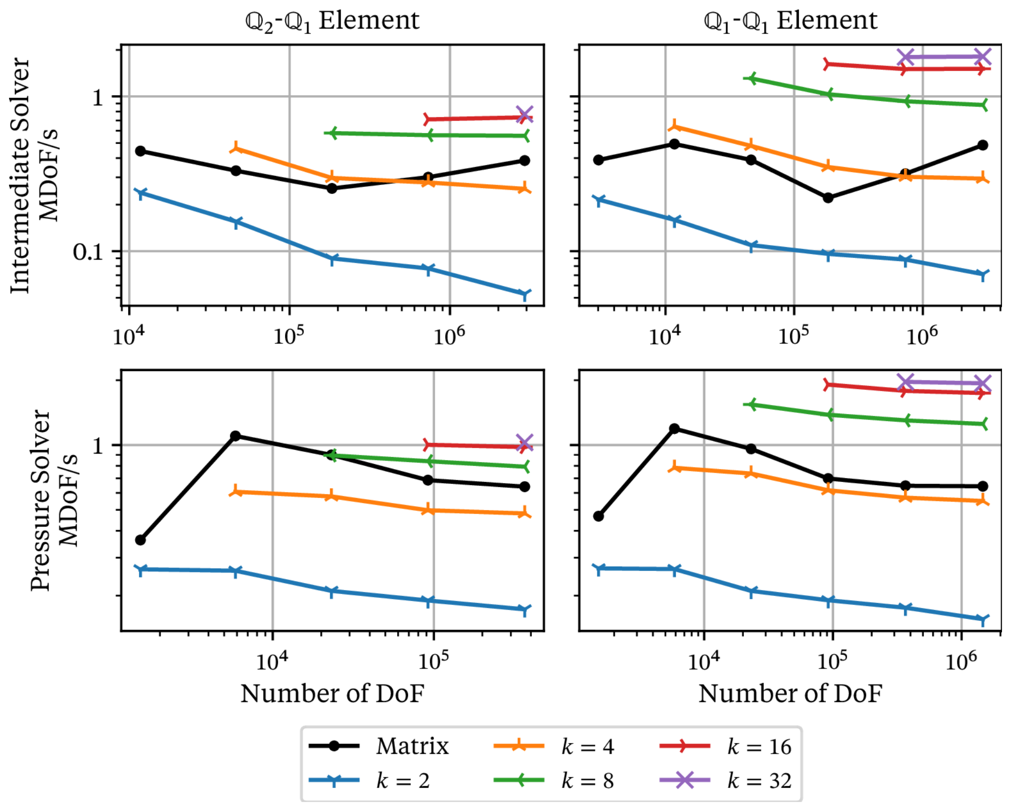

Figure 6.20: Performance of one iteration of each linear solver measured in MDoF/s. Most matrix-free variants achieve a higher performance for a single iteration than the matrix-based variant. In the best case, during the intermediate step the matrix-free variants attain ∼2× more DoF/s than the matrix-based variant, and in the pressure step even up to 3× more DoF/s.

sizes, which is most likely caused by the direct solver used for the coarse problem as part of the coarse grid correction. Since the coarse problem is considerably larger for smaller block sizes, the reduction is more pronounced in these cases.

6.3 Concluding Remarks

The performance analysis revealed two key characteristics of the matrix-free block-structured grid approach presented in this thesis. One concerning the single operator application and one concerning the whole solution process. If the amount of work per DoF is well beneath the established theoretical bound, the matrix-free operator application is faster than the corresponding sparse matrix-vector multiplication. To sharpen this bound and give a more reliable estimate the FLOP/s performance of the local kernel needs to be determined more accurately. Ideally it should depend directly on the generated code, using the ideas mentioned in section 6.1. In those instances where the bound is violated, the matrix-free approach might still be advantageous, especially for non-linear systems, as it does not require to assemble the system matrix. Although in many cases considered above the matrix-free operator application is faster, this does not translate to the solution process. The matrix-free preconditioner used here, i.e. the two-level method with Jacobi iterations on the fine grid and a coarse gird correction, showed a significantly lower convergence rate than the AMG preconditioner for the matrix-based variants. This leads to high iteration number that lessen the advantage per iteration. Therefore, the common theme of these benchmarks is the requirement for better preconditioners to fully utilize the benefits from the matrix-free block-structured grid approach.

CHAPTER 7
Summary & Conclusion

THIS THESIS SHOWED that the efficiency of matrix-free methods for low order discretizations can be significantly increased by using the block-structured grids approach. The groundwork for that is laid out in chapter 3, detailing the block-structuring approach and its implementation. Different optimizations that apply to block-structured grids were discussed, such as cross-element vectorization and efficient computation of geometry quantities, targeting the local kernels, or using streaming access into the global data structure during the gather and scatter operations, which reduces the global assembly overhead. Additionally, the problem of consistent assembly was brought up referencing known solutions without runtime overhead. With the previously mentioned optimizations up to 50–70% of the theoretical compute-bound peak performance were reached for handwritten local kernels.

By generating the local kernels, this performance can be reached for different PDEs without substantially more effort than deducing the weak formulation. Chapter 5 illustrates the generation process for block-structured kernels implemented in DUNE-CODEGEN. First, the weak formulation using the popular DSL UFL is transformed into an intermediate representation using Loopy, upon which transformations are applied to recreate the aforementioned handwritten optimizations. Besides the vectorization and efficient geometry computations, the code generator employs a simple loop invariant code motion method to further reduce the number of FLOP per local kernel, getting closer to ideal handwritten kernels. After these transformations, the intermediate representation is then translated into a C++ class, which can be used within a DUNE-PDELAB program.

A simple two-level preconditioner, based on non-overlapping domain decomposition, was adapted to the block-structuring approach in chapter 4. In contrast to the most widely available preconditioners, the application of the preconditioner does not require an assembled matrix on the fine grid and is therefore useful in the matrix-free setting considered in this thesis. By also solving the local problems appearing during the fine grid computations matrix free, the preconditioner can directly use the already generated local kernels. Thereby, it benefits immediately from the optimizations mentioned before.

Finally, the performance of the matrix-free block-structured grid approach has been thoroughly evaluated in chapter 6, highlighting the advantages of this approach as well as illustrating about its shortcomings. The performance of one single operator application is discussed, showing the high efficiency under appropriate circumstances. Its performance is also considered in the context of one full solution. Furthermore, a bound on the micro element local amount of work has been developed to specify these circumstances. This can help to estimate if the matrix-free operator application will be faster than multiplying the corresponding assembled matrix with a vector, although better estimates for the FLOP/s performance of the local kernels are necessary to supply sharper bounds.

As can be seen from the discussion in chapter 6, the block-structured grid approach is not suitable for all problems. Above all, if the FLOP bound is violated, perhaps due to an expansive coefficient function or non-linearity, the matrix-free approach might be slower than the matrix-based one. But there are other limiting factors than the FLOP bound. Since block-structuring is equivalent to uniform grid refinement, it should only be applied to problems where this grid refinement strategy is appropriate. Otherwise, for problems requiring highly adaptive meshes, it is not suitable conceptually. In these cases either the matrix-based approach should be chosen or other optimization techniques for matrix-free computations have to be considered. However, at least in those instances where the FLOP bound holds and the macro grid is suitable for uniform refinement, block-structuring is a strong choice to increase the operator application performance, especially if it is developed further.

Most of the further research direction were already mentioned in the corresponding chapters, nevertheless, the most important and promising areas are highlighted in the following. The current local kernel performance is already substantial, especially for low order methods, but there are still some opportunities for further enhancements, most pressing the vectorization for elements with degree greater than 1 and tiling for higher block sizes. These would lead to a more consistently high performance. In order to widen the possible application of the block-structured optimization, the generalized interface for local structuring, discussed in section 5.5.1, could be pursued further. In particular, generating locally structured simplex meshes would be helpful for 3D applications with unstructured grids, as the mesh generation for 3D purely cubical meshes is still in its early stages. Additionally, using affine linear simplex elements implies that the geometry transformation is also affine linear, and therefore needs to be computed only once per macro elements even for unstructured meshes, which benefits more from the optimization developed in chapter 5. As seen during the benchmark evaluation in chapter 6, in particular during the solver performance discussion, better preconditioners are necessary. In cases where the matrix-free operator application out-performs the conventional sparse matrix-vector multiplication, the matrix-free solver could not achieve this performance due to slower convergence. With more efficient preconditioners, for example those alluded to in the outlook of chapter 4, this deficiency would be fixed. It needs to be stressed that investigating more efficient preconditioners that harness the efficiency of the generated, optimized local kernels is imperative to find practical use for the block-structured grid approach discussed here.

In conclusion, this thesis shows that generating block-structured kernels improves the efficiency of matrix-free computations for low order FEM, without requiring elaborate user implementations. The resulting matrix-free operator application is faster than the corresponding matrix-vector multiplication, if the local amount of work stays below a theoretical limit, which depends on the sparsity of the assembled matrix. Three key factors are responsible for the attained high performance. First, the reduction of the global assembly overhead by allowing streaming access into the global data and decreasing the number of global index computations significant. The second factor, reducing the macro-element local work, is quite general and obvious, but the block-structuring reveals more options for removing redundant operations than otherwise possible, especially regarding the geometry computations. And lastly, cross-element vectorization allows to efficiently utilize modern CPU features. Therefore, the block-structured, matrix-free approach is a viable candidate to replace the conventional matrix-based computations for a specific set of problems, but for practical use efficient preconditioner need to be developed.

APPENDIX A
Performance Measuring Settings

All performance measurements in this thesis were done on the same hardware, using the same software stack with the same methodology, described in the following. First, the hardware stats of the used CPU are provided and then the versions and options of the necessary software packages are described. Finally, the considered performance measures as well as the measuring approach are explained.

A.1 Hardware

Supported core freqs:	1.2–3.6 GHz
Supported uncore freqs:	1.2–2.4 GHz
Cores/Threads:	20/40
SIMD extensions:	AVX-512
L1 cache capacity:	20×32 KiB
L2 cache capacity:	20×1 MiB
L1 cache capacity:	27.5 MiB
Memory configuration:	6 ch. DDR4-2666
Theor. mem. bandwidth:	128.0 GiB/s

Table A.1: CPU specification

The benchmarks were run on a dual-socket machine with an Intel Skylake server CPU, the exact model is a *Xeon Gold 6148*. In table A.1, borrowed from [52], the most important features of this particular CPU are gathered.

Judging the efficiency of a particular program requires a baseline for comparison. In particular, two hardware characteristics are widely used in the HPC community, the peak bandwidth in Byte/s for memory-bound codes and the peak FLOP/s for compute-bound codes. Both values can be determined theoretically or empirically, where the empiric peak is measured by running an appropriate benchmark. These benchmarks are chosen such that they reflect the typical work load of an either memory-bound or compute-bound and are therefore a more realistic target than the theoretical peaks.

The common benchmark for peak bandwidth is the STREAM benchmark[1], which computes

$$a_i = b_i * c + d_i.$$

This benchmark can be used as reference by any code that has the same memory access pattern, in this case one write/write stream and two read streams.

[1]`http://www.cs.virginia.edu/stream/`

Another popular benchmark is the triad benchmark, using one additional read stream by computing

$$a_i = b_i * c_i + d_i.$$

The achieved bandwidth directly depends on the size of these vectors, which dictates in which memory layer they fit in. Table A.2 displays the measured bandwidth for both benchmark run on 20 cores, using vector sizes characteristic for each memory layer.

Level	Size	Stream	Triad
L1	20×16 KiB	4358.98 GiB/s	4207.79 GiB/s
L2	20×256 KiB	1338.86 GiB/s	1522.34 GiB/s
L3	27 MiB	362.04 GiB/s	375.96 GiB/s
Main memory	1 GiB	100.91 GiB/s	100.75 GiB/s

Table A.2: Empirical peak bandwidth in GiB/s depending on the work size.

Nowadays the theoretical peak FLOP/s of a CPU is usually not advertised and has to be determined through combining lower level specifications. Dolbeau's paper [30] describes exactly how this computation works and also supplies the peak FLOP/s for the Skylake server architecture. According to Dolbeau, the peak FLOP/s directly depends on the used SIMD width and the achieved clock frequency. On Skylake CPUs the clock frequency is linked to the SIMD width of the instructions to be executed, leading to lower frequencies when AVX-512 heavy code is run, although the reduction can only be measured empirically. In table A.3 the first rows display the theoretical FLOP/s depending on the used SIMD width assuming a clock frequency of 2.4 GHz for non AVX-512 code and between 1.8–2.2 GHz for AVX-512 codes. The last row in table A.3 shows the measured peak FLOP/s and achieved clock frequency for the LINPACK benchmark[2], which is the standard benchmark for compute-bound codes and also used as part of the TOP500[3] supercomputer list.

Since the CPU core frequency directly influences the achieved FLOP/s, accurately evaluating the performance of a program requires to control the actual frequency at which it runs. Modern CPUs operate on a range of possible frequencies, which is controlled by the operating system according to the current CPU governor. The default 'ondemand' governor lowers or raises the frequency depending on the current work load, and the 'performance' governor uses always the highest possible frequency. Additionally, to the restrictions depending on the used SIMD instructions, the maximal frequency of a single core scales inversely with the number of currently active cores if the 'turbo mode' is activated, which are captured in table A.4. Therefore, to ensure that

[2] `https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-benchmarks-suite.html`

[3] `https://top500.org/`

SIMD Instructions	GFLOP/s
None	124
AVX2	416
AVX2 + FMA	832
AVX-512	576 (1.8 Ghz) – 704 (2.2 GHz)
AVX-512 + FMA	1152 (1.8 GHz) – 1408 (2.2 GHz)
LINPACK	910.62 (1.8 GHz)

Table A.3: Theoretical and empirical peak GFLOP/s depending on the used SIMD instructions.

Active CPUs	Normal	AVX2	AVX512
Base	2.4	1.9	1.6
1	3.7	3.6	3.5
2	3.7	3.6	3.5
3	3.5	3.4	3.3
4	3.5	3.4	3.3
5	3.4	3.3	3.1
6	3.4	3.3	3.1
7	3.4	3.3	3.1
8	3.4	3.3	3.1
9	3.4	3.1	2.6
10	3.4	3.1	2.6
11	3.4	3.1	2.6
12	3.4	3.1	2.6
13	3.3	2.8	2.3
14	3.3	2.8	2.3
15	3.3	2.8	2.3
16	3.3	2.8	2.3
17	3.1	2.6	2.2
18	3.1	2.6	2.2
19	3.1	2.6	2.2
20	3.1	2.6	2.2

Table A.4: CPU frequencies in GHz with ‘turbo mode’ activated.

the frequency is as stable as possible the ‘turbo mode’ is deactivated and the ‘performance’ CPU governor is activated for every benchmark. An additional benefit of these settings is that the socket GFLOP/s performances can be directly translated into single core performance, i.e. dividing the values in table A.3 by 20 results in the achievable floating point performance for one core.

A.2 Software

All benchmarks appearing in this thesis are written as DUNE projects, which, together with all required DUNE modules, can be found in the following git super project:

```
https://gitlab.com/MarcelKoch/thesis-superproject
```

The modules use the 2.7 release branch, if that is available, otherwise they are fixed to a commit compatible with the 2.7 releases. Both DUNE-PDELAB and DUNE-CODEGEN use custom branches, which incorporate some changes that were not merged into the master branch, in particular the DUNE-PDELAB custom branch contains the optimized gather and scatter operations discussed in 3.1.3. The modules are build using the `dunecontrol` command with the supplied `dune.opts` file, containing the C++ compiler and CMake options. Since this thesis is only concerned with single core performance and does not support parallelization in its current state, DUNE is build without MPI support, although enabling it would not change any outcomes.

The machine described earlier operates on Ubuntu 18.04 with kernel version 4.15.0-117-generic. As C++ compiler the GNU compiler `g++` version 9.3.0 was used with the following flags:

```
            -O3 -DRELEASE -DNDEBUG -march=native
-ftemplate-backtrace-limit=0 -Wno-deprecated-declarations
```

During the benchmarks in section 3.4.3 and 5.3.1 the auto-vectorization of `g++` is disables with the additional ‘`-fno-tree-vectorize`’ option for the programs with explicit vectorization. The coarse grid correction from chapter 4 uses the SuperLU library to solve the coarse problem directly, where version 5.2.1 is provided by the `apt` package `libsuperlu`. The BLAS and LAPACK dependencies of SuperLU are provided by the packages `libblas3` and `liblapack3` each implementing version 3.7.1.

The tool suite LIKWID version 4.3.4 is used for pinning threads to cores and monitoring performance counters as described in the following section.

A.3 Methodology

This section discusses what exactly is measured and, perhaps more importantly, how it is measured. As stated in the beginning of this chapter, both the achieved FLOP/s and bandwidth (BW), computed as

$$\text{FLOP/s} = \frac{\text{number of executed FLOP}}{\text{runtime in s}}, \quad \text{BW} = \frac{\text{number of transferred Byte}}{\text{runtime in s}},$$

of a program are common measures in the HPC community. They allow a quick comparison with the best possible values, which gives an easy-to-understand ranking of different implementations of the same algorithm. But these measures can be deceiving. For example, executing redundant operations on already loaded data will most likely increase the FLOP/s without increasing the overall efficiency. Therefore, measures more closely related to the targeted domain should be examined in addition. One such measure for FEM is DoF/s, computed as

$$\text{DoF/s} = \frac{\text{number of DoF}}{\text{runtime in s}},$$

which is especially useful for evaluating a single operator application, and another crucial measure is time-to-solution (tts), which is used when examining the whole simulation process, from assembling the (non-)linear system to computing its solution. In some cases more niche measures such as the ratio between vectorized and scalar instructions appear to be useful.

Many measures require the runtime of the program, which can be easily determined either through source code additions or a command-line wrapper. Determining the other part of the measurement, especially hardware related ones, is usually trickier. The number of FLOP a program executes, or the number of bytes it transfers, may be counted by hand for simplistic local kernels, but for complicated ones, such as the ones appearing in chapter 6, this is nearly impossible. At least for the FLOP number a C++ based approach is possible. For this approach the underlying data type, `double` or `float`, is replaced by a class, which implements all the arithmetic operations and static counters for each operation, allowing afterwards to inspect the number of FLOP executed during the programs run. Since this approach may interfere with the compiler's optimization and it is not extensible to measure memory transfers, it is not considered further.

An alternative approach monitors hardware performance counters through a special tool, where a fixed number of hardware events are counted automatically by the core. Depending on the CPU architecture, these events can contain, for example, the number of all retired instructions, the number of retired 256 bit SIMD instructions, or the number of L3 cache hits or misses. The quality of this approach directly depends on the quality of the available hardware events. On Skylake these events are fine-grained and reliable enough such that accurate measurements are possible, while on other architectures this is not the case, for instance, on Intel's Haswell the FLOP related events were too inaccurate, and on AMD's Zen does not distinguish between arithmetic operation of different SIMD sizes. The tool used for measuring the hardware events is `likwid-perfctr` from the LIKWID tool suite.

Although the current implementation of the block-structured grids is not optimized for parallelization, the effect of running on multiple cores is at least simulated. To this end, for each benchmark, multiple instances of that benchmark are executed at the same time filling up one socket of the system, which corresponds to one NUMA domain on this architecture. The `likwid-perfctr` tool allows to pin the currently executed program to a specific core, thereby making it is possible to guarantee that every core of a socket is busy, while no hyperthreading is used. Each benchmark is accompanied by a python script available from the superproject. These scripts manage the execution of multiple instances with a correctly parameterized `likwid-perfctr` wrapper, and additionally ensure that the CPU frequency behavior is set as defined at the end of section A.1.

Every benchmark is repeated multiple times within one run, until the total runtime is larger than 0.5s to stabilize the measurements for smaller problems, where the number of repetitions is determined adaptively, depending on the runtime of one iteration. Thus, the reported measurements is an average of multiple iterations of the considered benchmark. Due to the specified frequency behavior and guaranteed minimal runtime, the measurements are quite stable with low variance, usually less than 5%. Therefore, every benchmark is run only once on 20 cores simultaneously, unless otherwise stated. From these 20 separate results, the 10% quantile, or 90% quantile if lower numbers are better, is chosen as the representative for that benchmark with the rational that the lower bound on a particular performance measure is more relevant than the upper bound for real-world applications. The last thing to note is that each measurement, e.g. the achieved FLOP/s or DoF/s, is w.r.t. one core, except for the bandwidth measurements, since that cannot be measured per-core.

Bibliography

[1] Rainer Agelek et al. "On Orienting Edges of Unstructured Two- and Three-Dimensional Meshes". In: *ACM Trans. Math. Softw.* 44.1 (July 2017), 5:1–5:22. ISSN: 0098-3500. DOI: `10.1145/3061708`. URL: `http://doi.acm.org/10.1145/3061708`.

[2] Martin Alnæs et al. "The FEniCS Project Version 1.5". en. In: *Archive of Numerical Software* 3.100 (2015). DOI: `10.11588/ANS.2015.100.20553`.

[3] Martin S. Alnæs et al. "Unified form language". In: *ACM Transactions on Mathematical Software* 40.2 (Mar. 2014), pp. 1–37. DOI: `10.1145/2566630`.

[4] Martin Sandve Alnæs. "UFL: a finite element form language". In: *Automated Solution of Differential Equations by the Finite Element Method*. Springer Berlin Heidelberg, 2012, pp. 303–338. DOI: `10.1007/978-3-642-23099-8_17`.

[5] Robert Anderson et al. "MFEM: A modular finite element methods library". In: *Computers & Mathematics with Applications* (July 2020). DOI: `10.1016/j.camwa.2020.06.009`.

[6] Olof Widlund Andrea Toselli. *Domain Decomposition Methods - Algorithms and Theory*. Springer Berlin Heidelberg, Oct. 18, 2004. 468 pp. ISBN: 3540206965. URL: `https://www.ebook.de/de/product/2807579/andrea_toselli_olof_widlund_domain_decomposition_methods_algorithms_and_theory.html`.

[7] Santiago Badia, Alberto F. Martı   n, and Javier Principe. "Implementation and Scalability Analysis of Balancing Domain Decomposition Methods". In: *Archives of Computational Methods in Engineering* 20.3 (July 2013), pp. 239–262. DOI: `10.1007/s11831-013-9086-4`.

[8] P. Bastian et al. "A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework". In: *Computing* 82.2-3 (Apr. 2008), pp. 103–119. DOI: `10.1007/s00607-008-0003-x`.

[9] P. Bastian et al. "A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE". In: *Computing* 82.2-3 (June 2008), pp. 121–138. DOI: `10.1007/s00607-008-0004-9`.

[10] Peter Bastian et al. "Hardware-Based Efficiency Advances in the EXA-DUNE Project". In: *Software for Exascale Computing - SPPEXA 2013-2015*. Ed. by Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel. Cham: Springer International Publishing, 2016, pp. 3–23. ISBN: 978-3-319-40528-5. DOI: `10.1007/978-3-319-40528-5_1`. URL: `http://dx.doi.org/10.1007/978-3-319-40528-5_1`.

[11] Peter Bastian et al. "The Dune framework: Basic concepts and recent developments". In: *Computers & Mathematics with Applications* (July 2020). DOI: `10.1016/j.camwa.2020.06.007`.

[12] S. Bauer et al. “A Stencil Scaling Approach for Accelerating Matrix-Free Finite Element Implementations”. In: *SIAM Journal on Scientific Computing* 40.6 (Jan. 2018), pp. C748–C778. DOI: `10.1137/17m1148384`.

[13] Benjamin Karl Bergen and Frank Hülsemann. “Hierarchical hybrid grids: data structures and core algorithms for multigrid”. In: *Numerical Linear Algebra with Applications* 11.23 (Mar. 2004), pp. 279–291. DOI: `10.1002/nla.382`.

[14] Petter E. Bjørstad and Olof B. Widlund. “Iterative Methods for the Solution of Elliptic Problems on Regions Partitioned into Substructures”. In: *SIAM Journal on Numerical Analysis* 23.6 (Dec. 1986), pp. 1097–1120. DOI: `10.1137/0723075`.

[15] David Bommes et al. “Quad-Mesh Generation and Processing: A Survey”. In: *Computer Graphics Forum* 32.6 (Mar. 2013), pp. 51–76. DOI: `10.1111/cgf.12014`.

[16] Dietrich Braess. *Finite Elements*. Cambridge University Press, 2007. DOI: `10.1017/cbo9780511618635`.

[17] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Springer New York, 2008. DOI: `10.1007/978-0-387-75934-0`.

[18] CEED. *Center for Efficient Exascale Discretizations in the U.S. Department of Energy’s Exascale Computing Project*. URL: `https://ceed.exascaleproject.org/`.

[19] Philippe Ciarlet. *The finite element method for elliptic problems*. Amsterdam New York New York: North-Holland Pub. Co. Sole distributors for the U.S.A. and Canada, Elsevier North-Holland, 1978. ISBN: 9780080875255. eprint: `https://www.elsevier.com/books/the-finite-element-method-for-elliptic-problems/ciarlet/978-0-444-85028-7`.

[20] B. Cockburn. “Discontinuous Galerkin methods”. In: *ZAMM* 83.11 (Nov. 2003), pp. 731–754. DOI: `10.1002/zamm.200310088`.

[21] Bernardo Cockburn, Suchung Hou, and Chi-Wang Shu. “The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. IV. The multidimensional case”. In: *Mathematics of Computation* 54.190 (May 1990), pp. 545–545. DOI: `10.1090/s0025-5718-1990-1010597-0`.

[22] Bernardo Cockburn, San-Yih Lin, and Chi-Wang Shu. “TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-dimensional systems”. In: *Journal of Computational Physics* 84.1 (Sept. 1989), pp. 90–113. DOI: `10.1016/0021-9991(89)90183-6`.

[23] Bernardo Cockburn and Chi-Wang Shu. “The Runge-Kutta Discontinuous Galerkin Method for Conservation Laws V: Multidimensional Systems”. In: *Journal of Computational Physics* 141.2 (Apr. 1998), pp. 199–224. DOI: 10.1006/jcph.1998.5892.

[24] Bernardo Cockburn and Chi-Wang Shu. “The Runge-Kutta local projection P^1-discontinuous-Galerkin finite element method for scalar conservation laws”. In: *ESAIM: Mathematical Modelling and Numerical Analysis* 25.3 (1991), pp. 337–361.

[25] Bernardo Cockburn and Chi-Wang Shu. “TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework”. In: *Mathematics of Computation* 52.186 (May 1989), pp. 411–411. DOI: 10.1090/s0025-5718-1989-0983311-4.

[26] Keith D. Cooper and Linda Torczon. “Chapter 10 - Scalar Optimizations”. In: *Engineering a Compiler (Second Edition)*. Ed. by Keith D. Cooper and Linda Torczon. Second Edition. Boston: Morgan Kaufmann, 2012, pp. 539–596. ISBN: 978-0-12-088478-0. DOI: https://doi.org/10.1016/B978-0-12-088478-0.00010-4. URL: http://www.sciencedirect.com/science/article/pii/B9780120884780000104.

[27] Andreas Dedner and Martin Nolte. “The Dune Python Module”. In: (July 13, 2018). arXiv: 1807.05252v1 [cs.MS].

[28] Peter Deuflhard. *Newton Methods for Nonlinear Problems*. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-23899-4.

[29] Clark R. Dohrmann. “A Preconditioner for Substructuring Based on Constrained Energy Minimization”. In: *SIAM Journal on Scientific Computing* 25.1 (Jan. 2003), pp. 246–258. DOI: 10.1137/s1064827502412887.

[30] Romain Dolbeau. “Theoretical peak FLOPS per instruction set: a tutorial”. In: *The Journal of Supercomputing* 74.3 (Nov. 2017), pp. 1341–1377. DOI: 10.1007/s11227-017-2177-5.

[31] Victorita Dolean, Pierre Jolivet, and Frederic Nataf. *An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation*. Society for Industrial and Applied Mathematics, Apr. 1, 2017. ISBN: 1611974054. DOI: 10.1137/1.9781611974065. URL: https://www.ebook.de/de/product/25835455/victorita_dolean_pierre_jolivet_frederic_nataf_an_introduction_to_domain_decomposition_methods_algorithms_theory_and_parallel_implementation.html.

[32] Daniel Drzisga, Brendan Keith, and Barbara Wohlmuth. “The surrogate matrix methodology: Low-cost assembly for isogeometric analysis”. In: *Computer Methods in Applied Mechanics and Engineering* 361 (Apr. 2020), p. 112776. DOI: 10.1016/j.cma.2019.112776.

[33] Alexandre Ern and Jean-Luc Guermond. *Theory and Practice of Finite Elements*. Springer New York, 2004. DOI: 10.1007/978-1-4757-4355-5.

[34] Lawrence Evans. *Partial differential equations*. Providence, R.I: American Mathematical Society, 2010. ISBN: 9780821849743.

[35] Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. "The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners". In: *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, 2006, pp. 267–294. DOI: 10.1007/3-540-31619-1_8.

[36] Charbel Farhat and Francois-Xavier Roux. "A method of finite element tearing and interconnecting and its parallel solution algorithm". In: *International Journal for Numerical Methods in Engineering* 32.6 (Oct. 1991), pp. 1205–1227. DOI: 10.1002/nme.1620320604.

[37] Charbel Farhat et al. "FETI-DP: a dual-primal unified FETI method?part I: A faster alternative to the two-level FETI method". In: *International Journal for Numerical Methods in Engineering* 50.7 (2001), pp. 1523–1544. DOI: 10.1002/nme.76.

[38] C. Feichtinger et al. "WaLBerla: HPC software design for computational engineering simulations". In: *Journal of Computational Science* 2.2 (May 2011), pp. 105–112. DOI: 10.1016/j.jocs.2011.01.004.

[39] Paul Fischer et al. "Scalability of High-Performance PDE Solvers". In: (Apr. 14, 2020). arXiv: 2004.06722v1 [cs.PF].

[40] Agner Fog. *Vector Class Library*. c++. Version 2.01.00. Nov. 23, 2019. URL: https://github.com/vectorclass/version2/releases.

[41] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass: Addison-Wesley, Dec. 1, 1995. ISBN: 0201633612. URL: https://www.ebook.de/de/product/3236753/erich_gamma_richard_helm_ralph_e_johnson_john_vlissides_design_patterns.html.

[42] Xifeng Gao et al. "Robust hex-dominant mesh generation using field-guided polyhedral agglomeration". In: *ACM Transactions on Graphics* 36.4 (July 2017), pp. 1–13. DOI: 10.1145/3072959.3073676.

[43] Christophe Geuzaine and Jean-Francois Remacle. "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 79.11 (May 2009), pp. 1309–1331. DOI: 10.1002/nme.2579.

[44] Amir Gholami et al. "FFT, FMM, or Multigrid? A comparative Study of State-Of-the-Art Poisson Solvers for Uniform and Nonuniform Grids in the Unit Cube". In: *SIAM Journal on Scientific Computing* 38.3 (Jan. 2016), pp. C280–C306. DOI: 10.1137/15m1010798.

[45] Björn Gmeiner et al. “Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters”. In: *Concurrency and Computation: Practice and Experience* 26.1 (Dec. 2012), pp. 217–240. DOI: 10.1002/cpe.2968.

[46] Björn Gmeiner et al. “Performance and Scalability of Hierarchical Hybrid Multigrid Solvers for Stokes Systems”. In: *SIAM Journal on Scientific Computing* 37.2 (Jan. 2015), pp. C143–C168. DOI: 10.1137/130941353.

[47] Christian Godenschwager et al. “A framework for hybrid parallel flow simulations with a trillion cells in complex geometries”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. ACM Press, 2013. DOI: 10.1145/2503210.2503273.

[48] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. “POLLY — PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION”. In: *Parallel Processing Letters* 22.04 (Dec. 2012), p. 1250010. DOI: 10.1142/s0129626412500107.

[49] J.L. Guermond, P. Minev, and Jie Shen. “An overview of projection methods for incompressible flows”. In: *Computer Methods in Applied Mechanics and Engineering* 195.44-47 (Sept. 2006), pp. 6011–6045. DOI: 10.1016/j.cma.2005.10.010.

[50] F. Hecht. “New development in freefem++”. In: *Journal of Numerical Mathematics* 20.3-4 (Jan. 2012). DOI: 10.1515/jnum-2012-0013.

[51] Jan S. Hesthaven and Tim Warburton. *Nodal Discontinuous Galerkin Methods*. Springer New York, 2008. DOI: 10.1007/978-0-387-72067-8.

[52] Johannes Hofmann et al. “Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors”. In: *Supercomputing Frontiers and Innovations* 7.2 (July 1, 2019). DOI: 10.14529/jsfi200204.

[53] M. Homolya and D. A. Ham. “A Parallel Edge Orientation Algorithm for Quadrilateral Meshes”. In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), S48–S61. DOI: 10.1137/15m1021325.

[54] Dominic Kempf. “Code Generation for High Performance PDE Solvers on Modern Architectures”. PhD thesis. 2019. DOI: 10.11588/HEIDOK.00027360.

[55] Dominic Kempf et al. “Automatic Code Generation for High-Performance Discontinuous Galerkin Methods on Modern Architectures”. In: (Dec. 19, 2018). arXiv: 1812.08075v1 [math.NA].

[56] David I. Ketcheson et al. “PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems”. In: *SIAM Journal on Scientific Computing* 34.4 (Jan. 2012), pp. C210–C231. DOI: 10.1137/110856976.

[57] Axel Klawonn and Oliver Rheinbach. “Inexact FETI-DP methods”. In: *International Journal for Numerical Methods in Engineering* 69.2 (2006), pp. 284–307. DOI: 10.1002/nme.1758.

[58] Andreas Klöckner. “Loo.py”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY’14*. ACM Press, 2014. DOI: 10.1145/2627373.2627387.

[59] Andreas Klöckner. *pymbolic*. python. Version 2020.1. Mar. 29, 2020. URL: https://github.com/inducer/pymbolic.

[60] Matthew G. Knepley and Dmitry A. Karpeev. “Mesh algorithms for PDE with Sieve I: Mesh distribution”. In: *Scientific Programming* 17.3 (2009), pp. 215–230. ISSN: 1058-9244. DOI: 10.3233/SPR-2009-0249.

[61] Nils Kohl et al. “The HyTeG finite-element software framework for scalable multigrid solvers”. In: *International Journal of Parallel, Emergent and Distributed Systems* 34.5 (Aug. 2018), pp. 477–496. DOI: 10.1080/17445760.2018.1506453.

[62] Moritz Kreutzer et al. “A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units”. In: *SIAM Journal on Scientific Computing* 36.5 (Jan. 2014), pp. C401–C423. DOI: 10.1137/130930352.

[63] Martin Kronbichler and Katharina Kormann. “A generic interface for parallel cell-based finite element operator application”. In: *Computers & Fluids* 63 (June 2012), pp. 135–147. DOI: 10.1016/j.compfluid.2012.04.012.

[64] Martin Kronbichler and Katharina Kormann. “Fast Matrix-Free Evaluation of Discontinuous Galerkin Finite Element Operators”. In: *ACM Transactions on Mathematical Software* 45.3 (Aug. 2019), pp. 1–40. DOI: 10.1145/3325864.

[65] Christian Lengauer et al. “ExaStencils: Advanced Stencil-Code Engineering”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 553–564. DOI: 10.1007/978-3-319-14313-2_47.

[66] Randall J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002. DOI: 10.1017/cbo9780511791253.

[67] Jing Li and Olof B. Widlund. “On the use of inexact subdomain solvers for BDDC algorithms”. In: *Computer Methods in Applied Mechanics and Engineering* 196.8 (Jan. 2007), pp. 1415–1428. DOI: 10.1016/j.cma.2006.03.011.

[68] Xiaoye S. Li. “An overview of SuperLU”. In: *ACM Transactions on Mathematical Software* 31.3 (Sept. 2005), pp. 302–325. DOI: 10.1145/1089014.1089017.

[69] Anders Logg, Kent-Andre Mardal, and Garth Wells, eds. *Automated Solution of Differential Equations by the Finite Element Method*. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-23099-8.

[70] Anders Logg and Garth N. Wells. “DOLFIN”. In: *ACM Transactions on Mathematical Software* 37.2 (Apr. 2010), pp. 1–28. DOI: 10.1145/1731022.1731030.

[71] Fabio Luporini, David A. Ham, and Paul H. J. Kelly. “An Algorithm for the Optimization of Finite Element Integration Loops”. In: *ACM Transactions on Mathematical Software* 44.1 (Mar. 2017), pp. 1–26. DOI: 10.1145/3054944.

[72] Jan Mandel. “Balancing domain decomposition”. In: *Communications in Numerical Methods in Engineering* 9.3 (Mar. 1993), pp. 233–241. DOI: 10.1002/cnm.1640090307.

[73] Jan Mandel and Clark R. Dohrmann. “Convergence of a balancing domain decomposition by constraints and energy minimization”. In: *Numerical Linear Algebra with Applications* 10.7 (2003), pp. 639–659. DOI: 10.1002/nla.341.

[74] Sally A. McKee. “Reflections on the memory wall”. In: *Proceedings of the first conference on computing frontiers on Computing frontiers - CF’04*. ACM Press, 2004. DOI: 10.1145/977091.977115.

[75] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. DOI: 10.7717/peerj-cs.103.

[76] Steffen Müthing, Marian Piatkowski, and Peter Bastian. “High-performance Implementation of Matrix-free High-order Discontinuous Galerkin Methods”. In: (Nov. 29, 2017). arXiv: 1711.10885v1 [math.NA].

[77] Daniele Antonio Di Pietro and Alexandre Ern. *Mathematical Aspects of Discontinuous Galerkin Methods*. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-22980-0.

[78] Alfio Quarteroni and Alberto Valli. *Numerical Approximation of Partial Differential Equations*. Springer Berlin Heidelberg, 1994. DOI: 10.1007/978-3-540-85268-1.

[79] Ralf Rannacher. “Numerical Linear Algebra”. de. In: (2018). DOI: 10.17885/HEIUP.407.

[80] Florian Rathgeber et al. “Firedrake”. In: *ACM Transactions on Mathematical Software* 43.3 (Jan. 2017), pp. 1–27. DOI: 10.1145/2998441.

[81] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Jan. 2003. DOI: 10.1137/1.9780898718003.

[82] Oliver Sander. *DUNE — The Distributed and Unified Numerics Environment*. Springer International Publishing, 2020. DOI: 10.1007/978-3-030-59702-3.

[83] Alberto Sartori et al. “deal2lkit: A toolkit library for high performance programming in deal.II”. In: *SoftwareX* 7 (Jan. 2018), pp. 318–327. DOI: 10.1016/j.softx.2018.09.004.

[84] M. Schäfer et al. "Benchmark Computations of Laminar Flow Around a Cylinder". In: *Notes on Numerical Fluid Mechanics (NNFM)*. Vieweg+Teubner Verlag, 1996, pp. 547–566. DOI: 10.1007/978-3-322-89849-4_39.

[85] Christian Schmitt et al. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers". In: *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. IEEE, Nov. 2014. DOI: 10.1109/wolfhpc.2014.11.

[86] Jonathan R Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. USA, 1994. URL: http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

[87] Jan Sjödin et al. "Design of graphite and the polyhedral compilation package". In: 2009.

[88] Barry Smith, Petter Bjorstad, and William Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge university press, 2004.

[89] Holger Stengel et al. "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model". In: *Proceedings of the 29th ACM on International Conference on Supercomputing - ICS '15*. ACM Press, 2015. DOI: 10.1145/2751205.2751240.

[90] S. Turek, Ch. Becker, and A. Runge. "The FEAST indices—realistic evaluation of modern software components and processor technologies". In: *Computers & Mathematics with Applications* 41.10-11 (May 2001), pp. 1431–1464. DOI: 10.1016/s0898-1221(01)00108-0.

[91] Stefan Turek et al. "FEAST-realization of hardware-oriented numerics for HPC simulations with finite elements". In: *Concurrency and Computation: Practice and Experience* 22.16 (May 2010), pp. 2247–2265. DOI: 10.1002/cpe.1584.

[92] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[93] Wm. A. Wulf and Sally A. McKee. "Hitting the memory wall". In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. DOI: 10.1145/216585.216588.

[94] Jinchao Xu and Jun Zou. "Some Nonoverlapping Domain Decomposition Methods". In: *SIAM Review* 40.4 (Jan. 1998), pp. 857–914. DOI: 10.1137/s0036144596306800.

[95] Sabine Zaglmayr. "High Order Finite Element Methods for Electromagnetic Field Computation". PhD thesis. Johannes Kepler University Linz, July 2006.

[96] Stefano Zampini. “PCBDDC: A Class of Robust Dual-Primal Methods in PETSc”. In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), S282–S306. DOI: `10.1137/15m1025785`.