INAUGURAL - DISSERTATION

zur Erlangung der Doktorwürde
der Naturwissenschaftlich-Mathematischen Gesamtfakultät
der Ruprecht-Karls-Universität Heidelberg

Vorgelegt von Diplom-Mathematiker

Dominic Kempf

aus Freiburg i. Br.

Tag der mündlichen Prüfung:

Code Generation for High Performance PDE Solvers on Modern Architectures

Betreuer: Prof. Dr. Peter Bastian
Prof. Dr. Artur Andrzejak

Abstract

Numerical simulation with partial differential equations is an important discipline in high performance computing. Notable application areas include geosciences, fluid dynamics, solid mechanics and electromagnetics. Recent hardware developments have made it increasingly hard to achieve very good performance. This is both due to a lack of numerical algorithms suited for the hardware and efficient implementations of these algorithms not being available.

Modern CPUs require a sufficiently high arithmetic intensity in order to unfold their full potential. In this thesis, we use a numerical scheme that is well-suited for this scenario: The Discontinuous Galerkin Finite Element Method on cuboid meshes can be implemented with optimal complexity exploiting the tensor product structure of basis functions and quadrature formulae using a technique called sum factorization. A matrix-free implementation of this scheme significantly lowers the memory footprint of the method and delivers a fully compute-bound algorithm.

An efficient implementation of this scheme for a modern CPU requires maximum use of the processor's SIMD units. General purpose compilers are not capable of autovectorizing traditional PDE simulation codes, requiring high performance implementations to explicitly spell out SIMD instructions. With the SIMD width increasing in the last years (reaching its current peak at 512 bits in the Intel Skylake architecture) and programming languages not providing tools to directly target SIMD units, such code suffers from a performance portability issue. This work proposes generative programming as a solution to this issue.

To this end, we develop a toolchain that translates a PDE problem expressed in a domain specific language into a piece of machine-dependent, optimized C++ code. This toolchain is embedded into the existing user workflow of the DUNE project, an open source framework for the numerical solution of PDEs. Compared to other such toolchains, special emphasis is put on an intermediate representation that enables performance-oriented transformations. Furthermore, this thesis defines a new class of SIMD vectorization strategies that operate on batches of subkernels within one integration kernel. The space of these vectorization strategies is explored systematically from within the code generator in an autotuning procedure.

We demonstrate the performance of our vectorization strategies and their implementation by providing measurements on the Intel Haswell and Intel Skylake architectures. We present numbers for the diffusion-reaction equation, the Stokes equations and Maxwell's equations, achieving up to 40% of the machine's theoretical floating point performance for an application of the DG operator.

German Abstract
—Zusammenfassung—

Die numerische Simulation mit partiellen Differentialgleichungen ist eine wichtige Teildisziplin des Höchstleistungsrechnens. Ihre Anwendungsgebiete umfassen beispielsweise Geowissenschaften, Fluiddynamik, Festkörpermechanik oder Elektromagnetismus. Durch die Entwicklungen der letzten Jahre im Hardwaresektor ist es zunehmend schwer geworden, sehr gute Performance zu erzielen. Gründe hierfür sind sowohl ein Mangel an numerischen Algorithmen, die gut für die Hardware geeignet sind, als auch ein Mangel an effizienten Implementierungen dieser Algorithmen.

In dieser Arbeit verwenden wir ein numerisches Verfahren, welches effizient auf moderner Hardware implementiert werden kann: Das unstetige Galerkinverfahren (Discontinuous Galerkin Finite Element Method) kann auf Hexaedergittern, unter Ausnutzung der Tensorproduktstruktur der Basisfunktionen und Quadraturformeln, mit optimaler Komplexität implementiert werden. Dies wird als Summenfaktorisierung bezeichnet. Die selben Algorithmen können verwendet werden um die Anwendung eines Operators zu implementieren, der in herkömmlichen Finite Elemente Methoden in eine Datenstruktur für dünnbesetzte Matrizen assembliert wird. Im Gegensatz zu assemblierten Matrizen erlaubt dies einen Algorithmus, dessen Performance durch die Rechenleistung des Prozessors und nicht durch seine Speicherbandbreite limitiert ist.

Effiziente Implementierungen dieses Verfahrens auf modernen CPUs müssen für eine bestmögliche Auslastung der SIMD-Einheiten des Prozessors sorgen. Da Standardcompiler für PDE-Probleme keinen zufriedenstellend vektorisierten Code generieren, muss SIMD Vektorisierung explizit im Quellcode vorgenommen werden. Die verfügbare SIMD Breite ist in den letzten Jahren stetig angestiegen (bis hin zu einer Breite von 512 bits in der Intel Skylake Architektur). Da Programmiersprachen jedoch kaum Sprachmittel für explizite SIMD Vektorisierung zur Verfügung stellen, ist es schwierig dies auf hardware-unabhängige Weise zu tun. Diese Arbeit schlägt generative Programmierung als Lösungsansatz für dieses Performanceportabilitätsproblem vor.

Zu diesem Zweck wird im Rahmen dieser Arbeit eine Toolchain entwickelt, welche ein in einer domänenspezifischen Sprache beschriebenes PDE Problem in hardware-spezifischen, optimierten C++ Code übersetzt. Diese Toolchain ist in den Userworkflow des DUNE-Projekts eingebettet, einem quelloffenen C++ Framework zur numerischen Lösung partieller Differentialgleichungen. Hierbei liegt das Hauptaugenmerk auf der Verwendung einer Zwischenrepräsentation, welche performanceorientierte Transformationen erlaubt. Desweiteren führt diese Arbeit eine neue Klasse von SIMD Vektorisierungsstrategien ein, welche Batches von Unterkerneln innerhalb eines Integrationskernels zusammenfasst. Der Codegenerator

traversiert die Menge dieser Vektorisierungsstrategien systematisch im Rahmen eines Autotuning-Prozesses.

Die Performance unserer Vektorisierungsstrategien und ihrer Implementierung wird durch Messungen auf den Intel Haswell und Intel Skylake Architekturen belegt. Dabei werden die Diffusions-Reaktions-Gleichung, die Stokes-Gleichungen sowie Maxwell's Gleichungen als Beispiele herangezogen. Für die Anwendung eines DG-Operators erzielen wir eine Maschinenauslastung von bis zu 40%.

Contents

List of Figures

List of Algorithms

List of Abbreviations and Acronyms

AD	Automatic Differentiation
AMG	Algebraic Multigrid
AoS	Array of Structures
API	Application Programming Interface
AST	Abstract Syntax Tree
AVX-512	Advanced Vector Extensions for 512 bits
AVX2	Advanced Vector Extensions 2
BLAS	Basic Linear Algebra Subprograms
CFD	Computational Fluid Dynamics
CG	Continuous Galerkin
CPU	Central Processing Unit
CSE	Common Subexpression Elimination
DAG	Directed Acyclic Graph
DG	Discontinuous Galerkin
DOF	Degree of Freedom
DSL	Domain-Specific Language
DUNE	Distributed and Unified Numerics Environment
FE	Finite Element
FMA	Fused Multiplication and Addition
FV	Finite Volume
FEM	Finite Element Method
FD	Finite Differences
FLOP	Floating Point Operation
FOSS	Free and Open Source Software
FSI	Fluid-Structure Interaction
GEMM	General Matrix Multiply
GPU	Graphics Processing Unit
HPC	High Performance Computing
ILP	Instruction Level Parallelism
IR	Intermediate Representation
JIT	Just-in-time
JSON	JavaScript Object Notation
MKL	Intel Math Kernel Library
MPI	Message Passing Interface
ODE	Ordinary Differential Equation

PDE	Partial Differential Equation
POD	Plain Old Data Structure
SIMD	Single Instruction Multiple Data
SWIP	Symmetric Weighted Interior Penalty
TSC	Time Stamp Counter
UFL	Unified Form Language
UQ	Uncertainty Quantification
SoA	Structure of Arrays
SSE	Streaming SIMD Extensions
VCFV	Vertex-centered Finite Volumes
XFEM	Extended Finite Element Method

List of Symbols

Symbol	Explanation
$\mathcal{B}_h$	Set of boundary facets in a mesh $\mathcal{T}_h$
$\mathbb{C}$	The field of complex numbers
δ_{ij}	Kronecker Delta
ϵ_{ijk}	Levi-Civita Symbol
F	Single facet from $\mathcal{B}_h \cup \mathcal{F}_h$
$\mathcal{F}_h$	Set of interior facets in a mesh $\mathcal{T}_h$
h	Mesh size
$H^1(\Omega)$	Sobolev space
$\mathbb{I}$	Identity matrix
$\mathcal{I}_h$	Interpolation operator
μ_T	Geometry mapping $\mu_T : \hat{T} \to T$
$\mathcal{O}$	Landau symbol
Ω	Spatial domain
$\mathcal{P}_k^d(\hat{T})$	Space of polynomials of degree up to k on simplicial $\hat{T} \subseteq \mathbb{R}^d$
$\mathcal{Q}_k^d(\hat{T})$	Space of polynomials of degree up to k on cuboid $\hat{T} \subseteq \mathbb{R}^d$
$\mathbb{R}$	The field of real numbers
T	Single cell from $\mathcal{T}_h$
$\hat{T}$	Reference element of cell T
$\mathcal{T}_h$	Triangulation of the domain Ω
tr	Trace operator
w	SIMD width in lanes

CHAPTER 1

Introduction

1.1 Motivation and Scope

Partial Differential Equations (PDEs) are an important tool in the modelling of many physical processes of societal interest. Notable application areas include geosciences, fluid dynamics, solid mechanics and electromagnetics. In the absence of analytical solutions, numerical techniques are mandatory for the treatment of PDEs. In fact, numerical simulation is often regarded as an emerging fundamental pillar of scientific work, complementing theory and experiment.

Numerical solution techniques for PDEs approximate a continuous model with a discrete model. Real world applications might require large computational domains and/or resolution of small scale features, leading to discrete model sizes of billions of unknowns and beyond. Numerical simulation with PDEs has therefore always been a natural subject for High Performance Computing (HPC). Indeed, the biggest supercomputing facilities in the world are used to perform numerical simulations with PDEs.

Modern hardware has started to hit the limits of traditional scaling, forcing hardware designers to introduce new levels of parallelism. As a consequence, HPC software developers need to address many complex tasks: Leverage multicore processors via multi threading, saturate the increased floating point capabilities of the hardware or deal with heterogeneous clusters including Graphics Processing Units (GPUs) and accelerator chips. Adapting numerical simulation codes to these new architectures involves two separate challenges: Development of algorithms suited for the hardware and efficient implementation of these algorithms.

The task of developing efficient algorithms requires revisiting existing numerical techniques in the light of the available hardware and employ numerical schemes that

enable HPC implementations. In the field of numerical simulation with PDEs this comprises e.g. the ongoing endeavor to develop communication-minimal domain decomposition schemes for Message Passing Interface (MPI) parallelism or the search for algorithmic alternatives with high arithmetic intensity (Floating Point Operations (FLOPs) per byte loaded from main memory) that allow compute-bound implementations on current Central Processing Units (CPUs). The latter motivates our use of the Discontinuous Galerkin (DG) Finite Element Method (FEM) on cuboid meshes in this work, as it allows a reduction of the algorithmic complexity of finite element assembly by exploiting the tensor product structure of finite elements through a technique called sum factorization. A matrix-free implementation of the DG scheme significantly lowers the memory footprint of the method and delivers a fully compute-bound algorithm.

The development speed in programming languages cannot hold up to the speed in hardware development. As a result, a lack of support for new hardware features in general purpose programming languages and tools becomes apparent. This makes performance portability across heterogeneous architectures a major challenge that requires dedicated programming models. In fact, the same portability issue already arises between CPU generations due to e.g. the increasing width of Single Instruction Multiple Data (SIMD) units. General purpose C(++) compilers are not capable of sufficiently vectorizing PDE codes and the C++ language does not provide a portable way of explicitly expressing SIMD computations. Bridging this gap between performance and portability is necessary in order to ensure the sustainability of HPC software.

Generative programming is a potential solution to this performance portability issue. Instead of manually implementing the performance-critical loops of finite element assembly in a general purpose programming language, the mathematical problem is expressed in a Domain-Specific Language (DSL). From this DSL, a source code generator produces hardware-specific code that is integrated with the rest of the simulation code. Beyond enabling performance portability, this approach facilitates separation of concern between computational scientists and performance engineers.

In the field of numerical simulation with PDEs, generative programming has been pioneered by the FEniCS project [80] in the last decade. They provide a DSL for finite element problems in the form of the Unified Form Language (UFL). Their code generation toolchain is fully embedded into the Python language and geared towards rapid prototyping of new models. In order to generate HPC-enabled code from the UFL DSL, new algorithms and data structures need to be developed and used in the code generation process. This work introduces such a toolchain that integrates into the Distributed and Unified Numerics Environment (DUNE) framework, a Free and Open Source Software (FOSS) simulation toolbox for PDE problems.

1.2 Contribution

The contribution of this thesis is two-fold: Firstly, we develop an HPC-enabled toolchain that employs generative programming in the field of numerical simulation with PDEs. Secondly, we formulate a new class of SIMD parallelization strategies for finite element assembly and study it using a variety of example problems.

The developed code generation toolchain automates the generation of C++ code for the innermost loops of finite element integration kernels. The user expresses the PDE problem in the popular UFL DSL which is also used in other PDE software. However, our toolchain translates UFL into an Intermediate Representation (IR) that is more suited to performance engineering than earlier approaches. This IR uses `loopy`, a Python project that expresses and transforms computational kernels in a hardware-independent fashion. We embed our toolchain into the user workflow of DUNE, greatly enriching the framework's capabilities.

SIMD vectorization of the finite element assembly algorithm is often realized by considering batches of local integration kernels, e.g. by grouping together multiple grid cells. This approach however increases the memory footprint of the kernel by a factor of the SIMD width and requires costly interleaving of the kernel input data. We instead pursue and extend an idea from [91], where batches of subkernels within a single integration kernel are used for SIMD vectorization. In contrast to [91], we consider many more opportunities to batch subkernels allowing us to target wider SIMD widths like in the Advanced Vector Extensions for 512 bits (AVX-512) instruction set. The class of vectorization strategies is explored systematically from within the code generator in an autotuning procedure. We show performance measurements for our strategies using the examples of the diffusion-reaction equation, the Stokes equations and Maxwell's equations. On the Intel Haswell and Intel Skylake architectures (with the AVX-512 instruction set), we achieve up to 40% of the machine's theoretical peak performance for an application of the DG operator.

The contributions of this thesis have been previously published in the following articles:

- D. Kempf, R. Heß, S. Müthing, and P. Bastian. "Automatic Code Generation for High-Performance Discontinuous Galerkin Methods on Modern Architectures". In: *arXiv preprint arXiv:1812.08075* (2018)
- D. Kempf and P. Bastian. "An HPC perspective on generative programming". In: *Proceedings of the Software Engineering for Science workshop*. 2019
- D. Kempf and T. Koch. "System testing in scientific numerical software frameworks using the example of DUNE". in: *Archive of Numerical Software* 5.1 (2017), pp. 151–168

This thesis reuses some of these articles verbatim, but indicates such use in the introduction of the relevant section.

1.3 Structure

This thesis is structured as follows: Section 2 introduces the fundamentals of this work which cover many aspects of math and computer science. We start off by giving a brief introduction into the finite element method and its DG variant. We continue with the description of the DUNE framework as the major software project that we base our work on. Afterwards, we summarize and characterize the use of generative programming in the PDE software landscape. A detailed description of the prevalent challenges in HPC on modern hardware architectures will lead us to the introduction of an algorithmic technique that we advocate as a good fit for implementation on such architectures: A matrix-free solution procedure in combination with sum factorization which reduces the algorithmic complexity of finite element problems. Section 3 is concerned with establishing a code generation toolchain for the DUNE framework. For its input DSL and its IR, this toolchain leverages several existing projects which are described in detail. After that, the main code generation algorithms and their embedding into the DUNE user workflow are described. Section 4 will introduce a new class of SIMD vectorization strategies tailored to the finite element assembly problem. These strategies constitute a search space which can be explored from an autotuning process integrated into the code generator. We will back our work with performance measurements on the Intel Haswell and Intel Skylake architectures in section 5. For that purpose we examine the diffusion-reaction equation, the steady state Stokes equations and Maxwell's equations.

CHAPTER 2

Fundamentals

In this chapter, we will study the fundamentals of this work. This will cover the FEM and the DG method, although we restrict ourselves to a very brief description that introduces the necessary notation and highlights relevant aspects of the methods. For a mathematically rigorous description, we refer to dedicated literature e.g. [21] and [34]. After that, we will introduce the DUNE framework as a FOSS environment for the numerical solution of PDEs. Following that we will study how generative programming can be used in PDE software and how other projects already do so successfully. We will then describe prevalent challenges in the current HPC landscape and motivate the introduction of sum factorization and matrix-free methods in subsequent sections.

2.1 The Finite Element Method

We will demonstrate the basics of the finite element method introducing the example of the diffusion-reaction equation, that will be also be the object of investigation in our numerical experiments in chapter 5. Considering a domain $\Omega \subset \mathbb{R}^d$, the diffusion-reaction equation reads

$$\begin{aligned} -\nabla \cdot (k(\boldsymbol{x})\nabla u) + c(\boldsymbol{x})u &= f && \text{in } \Omega \\ u &= g && \text{on } \Gamma_D \subseteq \partial\Omega \\ -k(\boldsymbol{x})\nabla u \cdot \boldsymbol{n} &= j && \text{on } \Gamma_N \subseteq \partial\Omega \end{aligned} \tag{2.1}$$

where $k : \mathbb{R}^d \to \mathbb{R}$ is called *conductivity*, $c : \mathbb{R}^d \to \mathbb{R}$ is the *reaction rate* and f is the *source term*. The function g describes the Dirichlet boundary conditions, whereas j describes the Neumann boundary flux. The above problem formulation requires $u \in C^2(\Omega) \cap C^0(\overline{\Omega})$ which is a prohibitively severe constraint for both analytical

results on existence and uniqueness of solutions, as well as for the development of numerical solution techniques. Therefore, the concept of solution is extended to take into account *weak solutions* that only fulfill the original PDE in a variational sense. Weak solutions are obtained from strong formulations by multiplying with a test function, integrating the result over the domain Ω and applying integration by parts:

$$\begin{aligned} \int_\Omega (-\nabla \cdot (k(\boldsymbol{x})\nabla u) + c(\boldsymbol{x})u)v \; dx &= \int_\Omega fv \; dx \\ \int_\Omega k(\boldsymbol{x})\nabla u \cdot \nabla v + c(\boldsymbol{x})uv \; dx - \int_{\partial\Omega} k(\boldsymbol{x})\nabla u \cdot \boldsymbol{n} \; ds &= \int_\Omega fv \; dx \\ \int_\Omega k(\boldsymbol{x})\nabla u \cdot \nabla v + c(\boldsymbol{x})uv \; dx &= \int_\Omega fv \; dx - \int_{\Gamma_N} jv \; ds \end{aligned} \tag{2.2}$$

In this weak formulation, it is sufficient that u is in the Sobolev space $H^1(\Omega)$. While Neumann boundary conditions are naturally implemented into the weak formulation, Dirichlet boundary conditions are implemented by restricting the solution space of the variational problem.

The finite element method aims at solving the variational problem from equation 2.2 in a discrete subspace of $H^1(\Omega)$. In order to construct this subspace, the domain Ω is subdivided into a triangulation $\mathcal{T}_h$, such that for all $T \in \mathcal{T}_h$, $\overline{T}$ is the image of a polytopal reference element $\hat{T}$ under a differentiable map μ_T. Additionally, $\mathcal{T}_h$ should cover the domain Ω in the sense that $\bigcap_{T\in\mathcal{T}_h} T = \emptyset$ and $\bigcup_{T\in\mathcal{T}_h} \overline{T} = \overline{\Omega}$. We require the triangulation to be *conforming*, meaning that the intersection of the closure of two elements is the image of a boundary facet of the reference polytope. In this work, we restrict ourselves to cuboid reference elements in order to later exploit their tensor product structure. These reference elements do not vary throughout the domain, allowing for a slightly simplified definition of the involved function spaces. We define the local polynomial space $\mathcal{Q}_k^d(\hat{T})$ of polynomials of degree up to k being defined on the d-dimensional reference element. From these, we can define the global space V_h:

$$V_h = \left\{ u \in C^0(\Omega) \,\middle|\, u|_T = p \circ \mu_T^{-1},\ p \in \mathcal{Q}_k^d(\hat{T}) \quad \forall T \in \mathcal{T}_h \right\} \tag{2.3}$$

It is possible to show that $V_h \subset H^1(\Omega)$, allowing V_h as a discrete solution space. We use a subscript h for all elements of V_h, unless the function being in H^1 is sufficient in that context.

Choosing a basis $\{\Phi_i\}_{i=0}^{|V_h|-1}$ for the space V_h allows us to transform the variational problem into an algebraic problem. To this end, we reformulate equation 2.2 in terms of a bilinear form a and a linear form l.

$$a(u,v) := \int_\Omega k(\boldsymbol{x})\nabla u \cdot \nabla v + c(\boldsymbol{x})uv \; dx \tag{2.4}$$

$$l(v) := \int_\Omega fv \; dx - \int_{\Gamma_N} jv \; ds \tag{2.5}$$

Algorithm 2.1: Matrix assembly algorithm for a generic finite element problem. Local contributions are accumulated into the dense matrix data structure $\hat{A}$ and afterwards scattered into global data strutures using the map g, that maps cell-local indices to indices in the global space. The update expression at each quadrature point $q \in QP$ is given by the integrand of the given PDE.

```
1  for T ∈ T_h do
2      N ← |Q_k^d(T) − 1|
3      for i ∈ {0, ..., N − 1} do
4          for j ∈ {0, ..., N − 1} do
5              Â[i, j] ← 0
6      for q ∈ QP do
7          for i ∈ {0, ..., N − 1} do
8              for j ∈ {0, ..., N − 1} do
9                  Â[i, j] ← Â[i, j] + update
10     for i ∈ {0, ..., N − 1} do
11         for j ∈ {0, ..., N − 1} do
12             A[g(T, i), g(T, j)] ← A[g(T, i), g(T, j)] + Â[i, j]
```

The algebraic problem is then obtained in the following way, assuming that $u_h = \sum_i (\boldsymbol{z})_i \Phi_i$:

$$
\begin{aligned}
\text{Find } u_h \in V_h \text{ s.t.:} \qquad a(u_h, v_h) &= l(v_h) && \forall v_h \in V_h \\
a(\sum_i (\boldsymbol{z})_i \Phi_i, \Phi_j) &= l(\Phi_j) && \forall j = 0, \dots \\
\sum_i (\boldsymbol{z})_i a(\Phi_i, \Phi_j) &= l(\Phi_j) && \forall j = 0, \dots \\
\boldsymbol{A}\boldsymbol{z} &= \boldsymbol{b} && (2.6)
\end{aligned}
$$

The fact that the bilinear form a defines the linear system of equations in the algebraic formulation motivates the most common choice of basis functions: Local support of basis functions directly translates into sparsity of the system matrix $\boldsymbol{A}$ which is necessary in order to be able to fit large matrices into computer memory and allows more efficient solution techniques.

Typically, assembly of the matrix $\boldsymbol{A}$ is cheaper if not done matrix entry by matrix entry. Instead, the assembly algorithm 2.1 loops over grid cells and updates the degrees of freedom (DOFs) associated with all basis functions whose support overlaps the current cell. The assembly contributions of the current cell are accumulated into a dense container and afterwards scattered into the global data structures using a local-to-global map $g : \mathcal{T}_h \times \{0, \dots, |\mathcal{Q}_k^d| - 1\} \to \{0, \dots, |V_h| - 1\}$. We will see in section 2.2 that this step can also be avoided in some cases by chosing a favorable memory layout of the global data structures.

The size of the algebraic system in equation 2.6 scales with the dimensionality of the space V_h and therefore with the number of cells in the triangulation $\mathcal{T}_h$. In order to ensure a given bound on the discretization error, the mesh width h needs to be reduced below a certain threshold. This might result in very large problems, as applications with relevance in engineering may involve very large domains, small scale features that need to be resolved or may require very high accuracy. Large PDE problems may easily saturate and exceed the largest supercomputers in the world, making finite element simulations a natural subject for HPC.

2.2 The Discontinuous Galerkin Finite Element Method

We will now move on to extending the introduced notation for DG methods. We restrict ourselves to cuboid meshes and express the tensor product structure of basis functions. We do so in order to have the notation for sum factorization in section 2.6 readily available. The discrete solution space for DG methods allows functions to be discontinuous across interior faces $F \in \mathcal{F}_h$, which results in the functions being two-valued on such interfaces:

$$V_h^{\boldsymbol{k}} = \left\{ v \in L^2(\Omega) \;\middle|\; \forall T \in \mathcal{T}_h, v|_T = p \circ \mu_T^{-1} \text{ with } p \in \bigotimes_{i=0}^{d-1} \mathbb{P}^1_{k_i}([0,1]) \right\} \tag{2.7}$$

Here, the tensor product space is constructed such that the polynomial degree $\boldsymbol{k} = (k_0, \ldots, k_{d-1})$ is allowed to be anisotropic, although $\boldsymbol{k}$ is assumed to be constant across the triangulation.

Given the enlarged space from equation 2.7, the bilinear form from equation 2.2 is lacking analytical properties needed to ensure existence and uniqueness of solutions, such as coercivity. In order to fix this deficit, the bilinear form is modified by adding so called *penalty terms* which enforce continuity across interfaces in an approximation sense. These are integrals over facets $F \in \mathcal{F}_h \cup \mathcal{B}_h$, where $\mathcal{F}_h$ denotes the set of facets inside the domain and $\mathcal{B}_h$ denotes the set of boundary facets. These integrals involve the *average* and *jump* operators across F, which are defined in the following way:

$$\{v\} := \frac{1}{2}(v|_{T^+} + v|_{T^-}) \tag{2.8}$$

$$[\![v]\!] := v|_{T^-} - v|_{T^+} \tag{2.9}$$

Here, $T^+(F) \in \mathcal{T}_h$ and $T^-(F) \in \mathcal{T}_h$ define the outer and inner cell of a facet respectively. Such methods are called *interior penalty* methods and have been introduced and studied in e.g. [11] [119] [20]. For the diffusion-reaction equation

with spatially variable, tensor-valued coefficient $\boldsymbol{k}(\boldsymbol{x})$, the averaging is adapted to be a weighted averaging [38]:

$$\{w\}_\omega := \omega^+ v|_{T^+} + \omega^- v|_{T^-} \tag{2.10}$$

$$\omega^+ := \frac{\delta^-}{\delta^- + \delta^+ + 1e{-}20} \qquad \omega^- := \frac{\delta^+}{\delta^- + \delta^+ + 1e{-}20} \tag{2.11}$$

$$\delta^+ := \boldsymbol{n}^T \boldsymbol{k}^+(\boldsymbol{x})\boldsymbol{n} \qquad \delta^- := \boldsymbol{n}^T \boldsymbol{k}^-(\boldsymbol{x})\boldsymbol{n} \tag{2.12}$$

For the sake of readability, we will restrict ourselves to the simpler case of a scalar permeability function $k(\boldsymbol{x})$.

The resulting method is called Symmetric Weighted Interior Penalty (SWIP) DG method and its formulation for the diffusion-reaction equation reads the following:

$$\begin{aligned} a^{DG}(u,v) = &\sum_{T\in\mathcal{T}_h} \int_T k(\boldsymbol{x})\nabla u \cdot \nabla v + c(\boldsymbol{x})uv \; dx \\ &- \sum_{F\in\mathcal{F}_h} \int_F (\{k(\boldsymbol{x})\nabla u\}_\omega, \boldsymbol{n})[\![v]\!] + [\![u]\!](\{k(\boldsymbol{x})\nabla v\}_\omega, \boldsymbol{n}) - \gamma_F [\![u]\!][\![v]\!] \; ds \\ &- \sum_{F\in\mathcal{B}_h\subseteq\Gamma_D} \int_F (k(\boldsymbol{x})\nabla u, \boldsymbol{n})v + u(k(\boldsymbol{x})\nabla v, \boldsymbol{n}) - \gamma_F uv \; ds \end{aligned} \tag{2.13}$$

$$\begin{aligned} l(v) = &\sum_{T\in\mathcal{T}_h} \int_T fv \; dx - \sum_{F\in\mathcal{B}_h\subseteq\Gamma_N} \int_F jv \; ds \\ &- \sum_{F\in\mathcal{B}_h\subseteq\Gamma_D} \int_F g(k(\boldsymbol{x})\nabla v, \boldsymbol{n}) - \gamma_F gv \; ds \end{aligned} \tag{2.14}$$

where the penalty parameter γ_F is essential for the analytical properties of the bilinear form. It is defined as

$$\gamma_F := \alpha \frac{|F|}{\min\{|T^-(F)|, |T^+(F)|\}} \frac{2\delta^-\delta^+}{\delta^- + \delta^+ + 1e^{-20}} k(k + dim - 1) \tag{2.15}$$

with α being a user-defined constant and k being the polynomial degree.

The given formulation of DG methods is only one variant of defining discontinuous finite elements. Other approaches to DG, such as LDG [25] exist, but are not covered in this work. Hyperbolic conservation laws are another important field of application for DG methods. In this context, DG methods are a natural extension of Finite Volume (FV) methods to higher polynomial degrees [26]. We will cover this (very different) type of DG methods in section 5.4 while studying Maxwell's equations and assume the SWIP DG method everywhere else.

The standard assembly procedure for continuous finite elements from algorithm 2.1 contains a step where the assembly results are scattered back into global data structures using a local-to-global map. This costly step can be avoided in DG methods: As all DOFs are associated with cells (instead of facets, edges or vertices), the global data structure can have a memory layout that contains per-element blocks. Such layout induces a trivial local-to-global map and is vital for an HPC implementation of DG methods.

2.3 The DUNE Framework

DUNE is an open-source software framework for the solution of PDEs with grid-based methods [15, 14]. It supports finite element methods, finite volume methods and finite difference methods. It is written in C++ and tries to provide zero-overhead interfaces to all the components of a simulation program by leveraging C++ templates. Simulation codes programmed to these interfaces allow easy interchange of software components as the scientific requirements on the simulation code evolve. The most notable and unique such interface is the DUNE grid interface. DUNE allows massively parallel, adaptive simulations with a large variety of specialized grid implementations.

We will now shortly describe the structure of DUNE: The source code is provided in a heavily modularized way with each module being a `git` repository and CMake project in itself. Modules can be categorized as follows:

- *Core modules* provide basic infrastructure like build system and data structures (*dune-common*), the grid interface (*dune-grid*), linear algebra including an implementation of Algebraic Multigrid (AMG) (*dune-istl*), geometry implementations (*dune-geometry*) and local basis functions (*dune-localfunctions*).
- *Grid modules* provide additional implementations of the grid interface. Most prominently, this section comprises the current status of the long-standing projects UG [13] and ALUGrid [6], which both provide massively parallel, adaptive, unstructured grid implementations in two and three dimensions.
- *Discretization modules* provide the necessary abstractions to implement the finite element methods by introducing the involved discrete function spaces and their representations in memory. Several such projects exist due to the different focus of their developers. We will describe the discretization framework PDELab in detail later in this section.
- *Extension modules* provide additional functionality extending the DUNE core modules without being specific to a discretization framework. Examples are *dune-python* [32], which provides Python binding to the DUNE core modules or *dune-testtools* [65], which extends the testing infrastructure of *dune-common*.
- *User modules* use the same structure as upstream modules, standardizing the process of building and installing any piece of DUNE software.

We continue with a description of two components that will later have a strong influence on the design decisions for the code generation toolchain: The DUNE grid interface and the discretization framework PDELab.

The DUNE Grid Interface

The DUNE grid interface is the centerpiece of the DUNE framework software design. It is used to access a large variety of grid implementations through an interface. Available grid implementations cover a large variability of grid features such as:

- Dimensionality of the grid: The domain Ω may be a subset of $\mathbb{R}^d$ with $d = 1, 2, 3, \ldots$. Also, the topological dimension of the grid may differ from its geometric one.
- Structure of the grid: Structured grids allow more efficient implementation of algorithms compared to fully unstructured ones. The latter ones are needed to mesh complex geometries.
- Local refinement can be implemented using a variety of refinement rules. This includes building of a conforming closure versus using hanging nodes.
- Geometries in the grid: Grids may use different basic geometries, such as cubes or simplices. Also, they might allow mixing of different geometries within one grid.
- Parallelism can be implemented through overlapping or nonoverlapping subdomains. Not all grids provide communication methods for all these methods.
- The hierarchical structure of a refined grid can be used to implement multigrid schemes.

The grid interface allows iteration over cells using C++ iterators and iterator ranges. Furthermore, given a cell, it provides iterators over a cells intersection with its neighboring cells. For cells and intersections, the grid interface provides the geometry mappings μ_T and μ_F. The interface of these geometry objects gives access to commonly used geometric quantities, such as the inverse of the jacobian $\nabla\mu_T$.

We will now describe the basic abstractions of the discretization framework PDELab [19]. Special emphasis is put on those parts that will affect the code generation approach later on. The description given here is inspired by the tutorial texts in the *dune-pdelab-tutorials* [97] module.

Function Spaces

PDELab provides implementations of discrete finite element spaces in the form of `GridFunctionSpace`s. These combine the following pieces of information:

- a `GridView` object from *dune-grid* that describes the triangulation $\mathcal{T}_h$.
- a `FiniteElementMap` object that maps grid cells to finite element which then provide the mapping of DOFs to subentities of the given cell.

- a `ConstraintsAssembler` that describes how constraints for the space can be assembled. These may arise from essential boundary conditions, hanging nodes or overlapping computations.
- a `VectorBackend` that is used for implementation of the resulting linear algebra containers.

A key feature of PDELab is composition of finite element spaces into product spaces by building arbitrarily nested trees of finite element spaces. PDELab uses template metaprogramming to statically reason about such trees to implement a large variety of blocking schemes for the underlying linear algebra containers [90]. Such blocking scheme can be leveraged for DG methods to block all DOFs from one grid cell into contiguous blocks of memory. This block structure can be exploited from linear algebra algorithms.

In order to enable the user to write local integration kernels, PDELab provides an abstraction for the restriction of function spaces to a single cell. This `LocalFunctionSpace` object provides the local finite element of that cell and the mapping of its DOFs into the local assembly container.

Residual Formulation

In contrast to the introduction from section 2.1, PDELab expects the variational problem to be in *residual formulation*, as it is beneficial for the unified treatment of linear and nonlinear problems. With a residual formultion, we seek a solution $u_h \in U_h$, such that:

$$r_h(u_h, v_h) = 0 \qquad \forall v_h \in V_h \tag{2.16}$$

with U_h and V_h being suitably constrained, discrete finite element function spaces. The formulation from section 2.1 can be translated into residual formulation by defining $r_h(u_h, v_h) = a(u_h, v_h) - l(v_h)$. The solution u_h is a linear combination of basis functions $\{\Phi_j\}_{j=0}^{|U_h|-1}$ of the discrete space U_h. Let $\boldsymbol{z}$ denote the vector of coefficients of this linear combination: $u_h = \sum_j (\boldsymbol{z})_j \Phi_j$. This gives rise to an algebraic formulation of equation 2.16:

$$R_i(\boldsymbol{z}) = r_h(\sum_j (\boldsymbol{z})_j \Phi_j, \Psi_i) = 0 \qquad \forall i \in \{0, \dots, |V_h| - 1\} \tag{2.17}$$

Here, we also used the fact that it is sufficient to test against the basis $\{\Psi_i\}_{i=0}^{|U_h|-1}$ of the discrete space V_h. The non-linear, vector-valued mapping $\boldsymbol{R} : \mathbb{R}^{|U_h|} \mapsto \mathbb{R}^{|V_h|}$ fully describes the given discretization. Equation 2.17 is typically solved using a fixed-point-type iteration method. We assume a Newton iteration taking the following form (with k being the iteration index):

$$\boldsymbol{z}^{k+1} = \boldsymbol{z}^k - \boldsymbol{J}^{-1}(\boldsymbol{z}^k)\boldsymbol{R}(\boldsymbol{z}^k) \tag{2.18}$$

where $\boldsymbol{J}(\boldsymbol{z}^k)$ is the jacobian matrix of $\boldsymbol{R}$ defined as follows:

$$(\boldsymbol{J}(\boldsymbol{z}))_{i,j} = \frac{\partial R_i}{\partial z_j}(\boldsymbol{z}). \tag{2.19}$$

Introducing a correction variable $\boldsymbol{d}^{k+1} = \boldsymbol{z}^k - \boldsymbol{z}^{k+1}$, equation 2.18 boils down to solving the following linear system of equations:

$$\boldsymbol{J}(\boldsymbol{z}^k)\boldsymbol{d}^{k+1} = \boldsymbol{R}(\boldsymbol{z}^k) \tag{2.20}$$

Equation 2.20 draws the line between the abstractions of the discretization framework PDELab and linear algebra libraries. PDELab provides all building blocks necessary to assemble the following building blocks:

- Evaluation of the algebraic residual $\boldsymbol{R}(\boldsymbol{z})$ as needed on the right hand side of equation 2.20. This has to be implemented by users for every new problem.
- Evaluation of the jacobian $\boldsymbol{J}(\boldsymbol{z})$ as needed in equation 2.20 if the linear solver or a preconditioner works on matrices. Numerical differentiation can be used to automatically derive this from the above residual.
- Evaluation of the action $\boldsymbol{J}(\boldsymbol{z})\boldsymbol{w}$ of the jacobian on a vector $\boldsymbol{w}$ for matrix-free linear solvers. Again, numerical differentiation can be used to provide this implementation.

Although the described abstractions are motivated from the non-linear case, they do hold for the linear case as well. In that case, the jacobian becomes independent of $\boldsymbol{z}$ and the Newton scheme from equation 2.18 simplifies to one iteration that solves the linear system.

Grid Operators

As described in section 2.1, finite element assembly is typically implemented in terms of local integration kernels. PDELab follows the same idea with its abstractions. A global assembly class called `GridOperator` performs the work needed to set up local integration kernels: Iterating over the grid's cells and facets, gathering matrix and vector entries needed for the local computation into a contiguous data structure, calling the local integration kernel, applying constraints and scattering the data back into global data structures. The local integration kernels are collected in a `LocalOperator` class, whose interface is of particular interest for this work, as we will generate code against it. We split the residual $r_h(u_h, v_h)$ into three contributions for integrals over cells, boundary facets and interior facets:

$$\begin{aligned} r_h(u_h, v_h) = &\sum_{T \in \mathcal{T}_h} \alpha^{vol}(T, u_h, v_h) \\ &+ \sum_{F \in \mathcal{B}_h} \alpha^{bnd}(F, u_h, v_h) \\ &+ \sum_{F \in \mathcal{F}_h} \alpha^{sk}(F, u_h^+, u_h^-, v_h^+, v_h^-) \end{aligned} \tag{2.21}$$

The functions α^{vol}, α^{bnd} and α^{sk} need to be implemented in the interface methods `alpha_volume`, `alpha_boundary` and `alpha_skeleton`, which take the integration entity, local functions spaces for test and ansatz functions as well as local containers for the finite element functions and residuals:

```
class LocalOperator {
  public:
  template<typename CELL,
           typename LFSU, typename LFSV,
           typename X, typename R>
  void alpha_volume(
    const CELL& cell,
    const LFSU& lfsu, const X& x,
    const LFSV& lfsv, R& r
  ) const;

  template<typename INTERSECTION,
           typename LFSU, typename LFSV,
           typename X, typename R>
  void alpha_boundary(
    const INTERSECTION& intersection,
    const LFSU& lfsu, const X& x,
    const LFSV& lfsv, R& r
  ) const;

  template<typename INTERSECTION,
           typename LFSU_S, typename LFSV_S, typename X_S,
           typename LFSU_N, typename LFSV_N, typename X_N,
           typename R_S, typename R_N>
  void alpha_skeleton(
    const INTERSECTION& intersection,
    const LFSU_S& lfsu_s, const X& x_s, const LFSV_S& lfsv_s,
    const LFSU_N& lfsu_n, const X& x_n, const LFSV_N& lfsv_n,
    R_S& r_s, R_N& r_n
  ) const;
};
```

Algorithm 2.2 shows the algorithmic work typically being done in these assembly methods using the example of `alpha_volume`. The fact that all parameters are accepted as template parameters in a duck-typing fashion enables more complex applications like cut-cell geometries through the same interface [17]. Additional interface methods are available for assembling jacobians and their actions. These rely on a similar splitting as equation 2.21 and are named `jacobian_*` and `jacobian_apply_*` instead of `alpha_*`. The methods for the jacobian action have two different signatures for linear and nonlinear problems, as they do not

Algorithm 2.2: Assembly algorithm in PDELab studied on the example of the cell integral of a residual evaluation for the diffusion-reaction problem from equation 2.2. The local data structures z and r have been set up as dense vectors by the grid operator similar to how it is outlined in algorithm 2.1 for matrices.

```
1  for q ∈ QP do
2      u ← 0
3      gradu ← 0
4      for i ∈ {0, ..., |Q_k^d| − 1} do
5          phi[i] ← basis evaluation at q
6          gradphi[i] ← basis gradient evaluation at q
7          u ← u + z[i] ∗ phi[i]
8          gradu ← gradu + z[i] ∗ gradphi[i]
9      fac ← |detJ^T J|^(1/2) ∗ ω_q
10     gq ← μ_T(q)
11     c ← c(gq)
12     k ← k(gq)
13     f ← f(gq)
14     for i ∈ {0, ..., |Q_k^d| − 1} do
15         acc ← (c ∗ u − f) ∗ phi[i] ∗ fac
16         for j ∈ {0, ..., d − 1} do
17             acc ← acc + k · gradu[j] ∗ gradphi[i][j] ∗ fac
18         r[i] ← r[i] + acc
```

only depend on u_h, but also on a linearization point. The fact that PDELab restricts itself to providing interfaces for integration over cells and intersections with data access being limited to the cell or the neighboring cells of an intersection is a deliberate decision. It is sufficient to implement most finite element schemes. Notable exceptions are higher-order Finite Differences (FD) methods and methods that require basis functions with wider support e.g. spline functions.

2.4 Generative Programming for PDEs

Generative programming is defined by the use of source code generators in the software development process. Code generation is used in many disciplines of computer science with varying motivation and terminology:

- In *compiler design*, code generation is the necessary last step that translates an IR into the compiled output. Research in this field is e.g. focussed on how to achieve this task in a portable fashion [44].
- The *programming languages* community uses the term multi stage programming to refer to the use of code generators as a language design tool [110].

- In *embedded systems* programming, customization of a software product to a physical product is often realized through generative programming [28]. It allows the selection of optimized variants for the specific given product.
- In *performance engineering*, source to source program transformation is employed to transition from one program to another program that performs better in a target metric [82].

Use of code generation in scientific software may fall into both the language design and the performance engineering category, depending on the intended goals. We provided a summary of important design decisions for a code generation approach in scientific software in [61]:

- *Choice of DSL.* DSLs provide the user interface of a code generation toolchain, which is tailored to the application domain. Defining a DSL for a numerical task is the key point in designing a usable code generation toolchain. Only an intuitive, simple design can balance the need for additional training with the new language.
- *To embed or not to embed.* Embedding DSLs into general purpose programming languages is advantageous in many ways: Developer knowledge about language and toolchains can be leveraged and the syntax of the DSL is somewhat standardized. However, you are tying your DSL's popularity and lifetime to those of the general purpose language, which might turn out disadvantageous if the trends within the scientific community change.
- *Scope of code generation.* A simulation workflow consists of many individual tasks from mesh generation to visualization. Generative programming may of course be used for the entire workflow, but it only excels at some specific tasks, mostly where performance-critical innermost loops are involved. A related question is the choice of programming language which is used for the simulation control flow. Fully embedded solutions use the programming language that the DSL is embedded into, other solutions might choose the target language of the code generator.
- *Choice of IR.* Leveraging generative programming for high performance computing requires an IR within the code generator that is capable of hardware-driven transformation. The design of such an IR is a very challenging task as one has to find a trade-off about how many assumptions are built into the IR. Having too many assumptions built in will limit the scope of the toolchain, while a too broad scope can make it hard to actually define performance-enabling transformations.
- *Performance optimization decision making*: Once an IR capable of transformation based performance optimization is found, the question of how decision making is driven in the transformation process arises. Having this transformation process under user control might be intriguing, but further widens the gap between performance and accessibility. A fully automatic

transformation procedure on the other hand is often out of scope or might require very detailed hardware models. Autotuning is another good possibility to do performance optimization, which requires the systematic definition of an optimization search space.

With these general criteria in mind, we study the design decisions of popular projects employing generative programming techniques for the solution of PDEs with the finite element method.

The FEniCS project [80] has established code generation within the PDE community by providing a very popular framework. FEniCS user code is written entirely in Python. To achieve this, FEniCS provides UFL [7], an embedded DSL for the description of discretized weak formulations of finite element problems, and Python bindings to the C++ problem solving environment dolfin [81]. Internally, the UFL input is Just-in-time (JIT)-compiled in a two-step procedure: A *form compiler* translates the DSL into C++ code and a general purpose C++ compiler compiles it into an object file, which is then linked to. The FEniCS form compiler is called FFC [70] and translates directly from UFL to C++ code without using another IR. It draws additional information about the implementation of finite elements from FIAT [68], which provides tabulations of finite element basis functions at quadrature points. The interface between the form compiler and the problem solving environment is of similar granularity as the `LocalOperator` interface in section 2.3, although it is designed with code generation in mind [8]. As a result, the interface e.g. uses plain C arrays for data transfers in order to simplify the code generation process. With no additional IR in place, FFC has limited possibilities to reason about performance of the generated code.

Another project that fully embeds the user workflow into Python is the Firedrake project [101]. It tries to be compatible with the FEniCS input language as much as possible. This includes both reuse of the input language UFL and interface compatibility with the Python bindings of the dolfin framework. However, the applied technology in the code generation toolchain differs vastly, as even the underlying simulation framework is completely Python-driven: The firedrake form compiler TSFC [55] generates a C-like IR, which COFFEE [83] optimizes further by applying SIMD vectorization. Execution of JIT-compiled local integration kernels is then controlled by the parallel grid iteration tool PyOP2 [102]. Lately, this part of the toolchain has been adapted to also apply SIMD vectorization through a transformation which is based on the programming model `loopy`, that we will discuss in detail in this thesis [109]. Firedrake draws much functionality that other frameworks implement in their underlying C++ framework (like grid data structures or linear algebra containers and algorithms) from the PetSc framework [12]. The declared goal of the Firedrake project is to enable separation of concern between domain scientists, numerical mathematicians and performance engineers in finite element software.

The FEniCS and Firedrake projects have established generative programming as a respected tool in the PDE community. They are not the only available projects though and some differ vastly from their Python-based approach: FreeFEM++ [52] provides a DSL that covers the entire simulation workflow, which is not embedded into a general purpose language. There is no publicly available information on the used IR or its capabilities to enable performance optimization. Feel++ [99] embeds a DSL for finite element problems into C++ using expression templates. This provides the usability aspects of DSLs at the cost of unfavorable compile-time complexity, but does not help with any hardware-specific performance optimization. Liszt [33] provides a non-embedded Scala-based DSL, which expresses the entire simulation workflow. It focusses on stencil-like operations and requires users to implement their problem in terms of global operations. Liszt applications are mapped to MPI, `pthreads` or CUDA programs by a code generator. Again, there is no detailed information on the IR available. The ExaStencils project [78] also targets stencil code and aims for separation of concerns through stacked DSLs.

2.5 HPC Challenges on Modern Architectures

The DUNE project started in 2002 and its core software architecture dates back to this time. A fundamental idea of achieving performance in numerical software was to use generic programming and metaprogramming to remove the runtime overhead of introducing interfaces. In the meantime, requirements for HPC have changed drastically. We have entered the so called *Post-Moore-Era*, where the increase in transistor density has drastically slowed down [107]. Clock frequency stays roughly unchanged and additional performance is achieved by introducing and extending parallelism within the architecture. We will now describe the current trends and their impact on how numerical software needs to be written. We have previously published this summary in [61].

Cluster Computing

The number of compute nodes in the latest supercomputers is still increasing, making the challenge of writing code that scales to the limits of the machine an increasingly difficult task involving both algorithmic and technical challenges. The algorithmic challenges are part of an ongoing endeavour to develop communication-minimal, scalable and robust numerical algorithms. The main technical tools and abstractions (like MPI support) are built into the DUNE framework from the very beginning.

Multicore Processing

Modern processors consist of many, often less powerful, cores. Memory band width does not increase at the same speed though, resulting in an effective decrease of memory band width per core. Making effective use of multicore processors with

shared memory domains requires the use of multi threading. Support for multi threading can either be integrated into the software framework or be delegated to the framework user. However, there are many obstacles in doing so:

- There is a serious lack of *standardization* w.r.t. threads. Task-based programming models seem a natural way of expressing numerical tasks, but C++ does not provide a built-in such model. Introducing external libraries such as TBB [103] as hard external dependencies is however a very drastic step for a numerical framework. The DUNE project has decided to stick to task-based programming and a TBB-based implementation has been provided by the EXADUNE project [18].
- *Programmability* is one of the major issues in making use of multi threaded systems. Existing programming model solutions use either programming language extensions (e.g. OpenCL [108]), libraries (e.g. Kokkos [37]) or source code annotation (e.g. OpenMP [29]) to employ parallelism. All of these approaches bear their advantages and disadvantages. Integrating them into a numerical software requires commitment to the choice and prohibits other concurrent solutions in the same codebase. For the DUNE project, no consensus on an invasive solution was to be found, and TBB tasks were found to be least invasive.
- *Debugability* of multi threaded applications, in particular finding nondeterministic bugs is a big usability issue. This can be solved by improving developer toolchains. However, this again introduces constraints on the developer hampering accessibility of numerical software.
- Numerical software will often make use of external libraries for well-studied tasks, such as solving linear systems or eigenvalue problems. *Composability* of multi threaded applications, if those external libraries are themselves multi threaded is a largely unsolved issue.

SIMD Vectorization

The width of SIMD units in modern CPUs has increased over the last years reaching its current peak with 512 bits in the Intel Skylake architecture. Making effective use of these vector units is essential to leveraging the machine's floating point capabilities. However, using these in PDE applications is not straight-forward and there is no unique way of doing so. In our experience C++ compilers are not capable of successfully autovectorizing PDE applications, due to the complexity of the control flow in PDE programs. Explicitly vectorizing instead requires both a concept of what to vectorize and a technical realization. Looking into a typical structure of a PDE program, there are many nested loops that can potentially be vectorized:

- Solution of multiple PDEs (Uncertainty Quantification (UQ), optimization)

- Timesteps of an instationary problem
- Stages of a time stepping scheme
- Iterations of a Newton solver for nonlinear problems
- Iterations of an iterative linear solver
- Iteration over grid cells
- Quadrature points for cell-local integration
- Components of a coupled system of PDEs
- Local degrees of freedom that are updated

Classical choices for SIMD vectorization are the outermost level (if e.g. doing UQ), the grid iteration level or the local degrees of freedom. Given such a choice, a technical realization is necessary: C++ only provides intrinsic functions (or inline assembler) for direct control of executed instructions. However, these pose a severe threat to portability, accessibility and sustainability of a numerical code. Therefore, many SIMD abstraction layer projects have come into existence (we will study these in detail in section 3.2.3). The DUNE framework has started incorporating these and adapting its codebase to be compatible with them. This framework-level SIMD abstraction is best suited for high level parallelism, such as UQ. If SIMD parallelism should be used on the innermost loops of the above loop structure, it needs to be incorporated into user code instead of framework code. Even when using SIMD abstraction libraries, this is a hard task that requires very specific user skills. We will therefore advocate generative programming for this case in this work.

Instruction Level Parallelism

Modern CPUs offer a wide range of features that allow parallel code execution on the lowest level, like instruction pipelining, superscalar execution on multiple floating point units, Fused Multiplication and Addition (FMA), speculative execution (e.g. branch prediction) or out of order execution. In practice, few of these are directly addressed in numerical software frameworks and optimally using these features is left to the C++ compiler and the hardware itself. This is partly because programmers have limited influence on details of code execution at the Instruction Level Parallelism (ILP) level. However, developers with technical knowledge of these features are able to write better and more performant code by avoiding known antipatterns. A good example of this would be to identify and avoid a data access pattern that provokes a pipeline stall.

Heterogenous Computing

The use of GPUs and accelerator chips in extreme scale high performance computing facilities has seen a drastic increase. In fact, eight of the ten largest supercomputers on the last edition of the TOP500 list are of heterogeneous nature [112]. Similar to multi threading, programmability is a large issue for these systems and usage of a

programming model beyond general purpose programming languages is mandatory. Beyond these technical obstacles, GPUs are hard to use for real world applications with PDEs. Reasons are the amount of control flow involved and the amount of data needed for computations and their transfer to the GPU. DUNE has only seen exploratory research in the direction of GPUs [18] and there are currently no plans to integrate GPU support into the framework.

2.6 Sum Factorization

Sum factorization is an algorithmic technique to exploit tensor product structure. In a PDE context, it was first described by Orszag [94]. It has since been adopted into a large variety of methods and code bases [22] [87] [116] [76] [54]. We will explain it using an example from DG methods after introducing the necessary notation. In the literature, tensor product structure is often expressed using Kronecker products. For two matrices $\boldsymbol{A} \in \mathbb{R}^{n_0 \times m_0}, \boldsymbol{B} \in \mathbb{R}^{n_1 \times m_1}$, the Kronecker product $\boldsymbol{A} \otimes \boldsymbol{B}$ yields a 4-way tensor or interpreted differently, a block matrix:

$$(\boldsymbol{A} \otimes \boldsymbol{B})_{i_0,j_0,i_1,j_1} = a_{i_0 j_0} b_{i_1 j_1} \tag{2.22}$$

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{0,0}\boldsymbol{B} & \cdots & a_{1,m_0-1}\boldsymbol{B} \\ \cdots & & \cdots \\ a_{n_0-1,1}\boldsymbol{B} & \cdots & a_{n_0-1,m_0-1}\boldsymbol{B} \end{bmatrix} \tag{2.23}$$

For a review of the Kronecker product and its properties see [79].

In order to present the sum factorized evaluation of a function $u_h \in V_h^{\boldsymbol{k}}$ from the tensor product DG space in equation 2.7, we introduce the matrices $A^{(i)} \in \mathbb{R}^{m_i \times n_i}$, whose entries are the evaluations of the chosen local basis functions of $\mathbb{P}^1_{k_i}([0,1])$ at the given 1D quadrature points $\xi_0, \ldots, \xi_{m_i-1}$. This assumes the quadrature rule to also exhibit tensor product structure, which is true for the reference cube. Given the coefficient vector $\boldsymbol{x}$ as a d-way tensor X, the evaluation of $\hat{u}_h$ at all quadrature points reads the following:

$$\hat{U} = \left(A^{(0)} \otimes A^{(1)} \otimes \cdots \otimes A^{(d-1)}\right) X \tag{2.24}$$

The tensor $\hat{U}$ from equation 2.24 is evaluated in the following way, which is the fundamental idea of sum factorization:

$$\hat{U}_{i_0 \ldots i_{d-1}} = \hat{u}_h(\boldsymbol{\xi}_{i_0 \ldots i_{d-1}}) \tag{2.25}$$

$$= \sum_{j_{d-1}=0}^{n_{d-1}-1} \cdots \sum_{j_1=0}^{n_1-1} \sum_{j_0=0}^{n_0-1} \prod_{k=0}^{d-1} A^{(k)}_{i_k,j_k} X_{j_0 \ldots j_{d-1}} \tag{2.26}$$

$$= \sum_{j_{d-1}=0}^{n_{d-1}-1} A^{(d-1)}_{i_{d-1},j_{d-1}} \cdots \sum_{j_1=0}^{n_1-1} A^{(1)}_{i_1,j_1} \sum_{j_0=0}^{n_0-1} A^{(0)}_{i_0,j_0} X_{j_0 \ldots j_{d-1}} \tag{2.27}$$

Note how the complexity of the calculation reduces in equation 2.27 in comparison to equation 2.26. Assuming $n_i = m_i = p$, the complexity of evaluating all elements of $\hat{U}$ decreases from $\mathcal{O}(p^{2d})$ to $\mathcal{O}(p^{d+1})$, illustrating well the desirability of the sum factorization approach.

Evaluation of u_h is not the only context, where the sum factorization technique can be applied in finite element assembly. For the evaluation of the partial derivative $\partial_j \hat{u}_h$, the basis evaluation matrix $A^{(j)}$ needs to be replaced with the matrix $D^{(j)} \in \mathbb{R}^{m_j \times n_j}$ containing the derivatives of the 1D basis functions at the 1D quadrature points. We use the notation $\partial_j \hat{U}$ for the d-way tensor containing the j-th component of the gradient of $\hat{u}$ at all quadrature points. Also, the tensor product structure of the test functions can be exploited by assembling a tensor that contains the value of the integrand without the test function at each quadrature point and using that tensor as input for sum factorized multiplication with the test function. In this case, the basis evaluation matrices need to be transposed. This requires restructuring of the assembly algorithm, as it is not possible to multiply with the test function within the quadrature loop. Algorithm 2.3 shows how algorithm 2.2 changes in the sum factorization setting.

Of course, the sum factorization technique is not limited to cell integrals, although we restrict ourselves to it most of the time when explaining algorithms. On a facet, the reduced dimension is handled by using a dummy quadrature rule with exactly one quadrature point. On the linear algebra level, a sum factorization kernel boils down to a series of tensor contractions and tensor rotations, which makes it desirable for HPC beyond it being an algorithm of reduced complexity, as all floating point operations in tensor contractions are FMAs. These operations are necessary in order to be able to aim for peak performance, because the theoretical peak performance assumes an all FMA implementation.

The exploratory research within the DUNE framework that this thesis is based on has been published in [91], [89] and [63]. Use of sum factorization on simplicial elements using Bernstein polynomials has been studied in [69] and [4]. Applications in isogeometric analysis are shown in [10].

2.7 Matrix-free Solvers

HPC applications that aim to fully exploit the capabilities of modern manycore CPUs need to have a sufficiently high *arithmetic intensity*. For a critical code section, this measure quantifies the number of floating point operations that are executed for each byte loaded from main memory (*flop-per-byte ratio*). For a given machine, a threshold for the arithmetic intensity can be calculated from the theoretical floating point peak performance and the maximum memory bandwidth: Performance of codes with an arithmetic intensity above this threshold are limited by the machine's floating point capabilities, those below are limited by memory bandwidth.

Algorithm 2.3: Sum factorized algorithm for calculating the cell integral for the residual for the diffusion-reaction from equation 2.2. This illustrates how the sum factorization approach restructures the local integration kernel compared to algorithm 2.2. The input coefficient is given as a tensor X, the output is accumulated into a tensor R. In the DG setting, these tensors are accessed direclty in global data structures without the need of gathering them into a local data structure first.

$\hat{U} \leftarrow \left(A^{(0)} \otimes A^{(1)} \otimes A^{(2)}\right) X$ ▷ Part 1: Evaluation of solution
$\partial_0 \hat{U} \leftarrow \left(D^{(0)} \otimes A^{(1)} \otimes A^{(2)}\right) X$
$\partial_1 \hat{U} \leftarrow \left(A^{(0)} \otimes D^{(1)} \otimes A^{(2)}\right) X$
$\partial_2 \hat{U} \leftarrow \left(A^{(0)} \otimes A^{(1)} \otimes D^{(2)}\right) X$
for $\hat{\xi}_{i_0 i_1 i_2} \in$ quadrature points **do** ▷ Part 2: Quadrature loop
  $fac \leftarrow |detJ^T J|^{\frac{1}{2}} * \prod_{i=0}^{2} \omega_i$
  $gq \leftarrow \mu_T(\hat{\xi}_{i_0 i_1 i_2})$
  $c \leftarrow c(gq)$
  $k \leftarrow k(gq)$
  $f \leftarrow f(gq)$
  $R^v_{i_0 i_1 i_2} \leftarrow (c * \hat{U}_{i_0 i_1 i_2} - f) * fac$
  $(R^{\partial_0 v})_{i_0 i_1 i_2} \leftarrow k * \partial_0 \hat{U}_{i_0 i_1 i_2} * fac$
  $(R^{\partial_1 v})_{i_0 i_1 i_2} \leftarrow k * \partial_1 \hat{U}_{i_0 i_1 i_2} * fac$
  $(R^{\partial_2 v})_{i_0 i_1 i_2} \leftarrow k * \partial_2 \hat{U}_{i_0 i_1 i_2} * fac$
$R \leftarrow R + \left(A^{(0),T} \otimes A^{(1),T} \otimes A^{(2),T}\right) R^v$ ▷ Part 3: Mult. with test fct.
$R \leftarrow R + \left(D^{(0),T} \otimes A^{(1),T} \otimes A^{(2),T}\right) R^{\partial_0 v}$
$R \leftarrow R + \left(A^{(0),T} \otimes D^{(1),T} \otimes A^{(2),T}\right) R^{\partial_1 v}$
$R \leftarrow R + \left(A^{(0),T} \otimes A^{(1),T} \otimes D^{(2),T}\right) R^{\partial_2 v}$

In traditional FEM implementations, the system matrix is assembled in memory and the sparse linear system (e.g. from equation 2.6) is solved with an efficient solver technique. Optimal complexity solvers e.g. multigrid schemes [113] scale linearly in the number of unknowns. Despite their optimal complexity, these schemes cannot leverage the machines capabilities very well as they rely on sparse matrix vector products of the assembled system matrix. The arithmetic intensity of this operations is as low as one FMA operation per matrix entry and therefore inherently memory-bound.

Matrix-free solvers are based on the rather simple idea of never assembling the system matrix in the first place, but recalculate its entries within each matrix-vector product. This has the obvious advantages of removing the costly matrix assembly procedure and drastically reducing the memory requirements of the overall program. The arithmetic intensity of such approach is also much higher, potentially enabling

a compute-bound implementation. However, one needs to make sure to use an optimal implementation of the matrix-free operator evaluation, because it might even take longer than performing the memory bound matrix-vector product. Such optimal implementations use sum factorization as described in section 2.6. The $\mathcal{O}(p^{2d})$ to $\mathcal{O}(p^{d+1})$ complexity reduction of the sum factorization approach allows matrix-free solvers to perform faster than matrix-based ones. This of course assumes that the implementation of the operator evaluation is capable of exploiting the machine, e.g. leverage its SIMD units. Achieving this task in a performance-portable fashion is one of the core contributions of this thesis.

The sum factorization complexity reduction depends on the polynomial degree p, showing more drastic gains for higher polynomial degrees. We therefore advocate the use of higher order schemes, which are an increasingly popular tool in many application areas e.g. Computational Fluid Dynamics (CFD) [118]. However, substantial gains can be observed even for rather low polynomial degrees, such as $p = 2$ [91].

It is worth noting, that iterative solvers using this kind of matrix-free operator evaluation suffer from the additional challenge to implement preconditioners that do not hamper the algorithmic complexity of the overall algorithm. Not applying suitable preconditioning would result in a suboptimal number of iterations of the linear solver that could easily outweigh the gains of being in the compute-bound regime. Matrix-free preconditioning techiques have been studied by various authors: In [89], Block-Jacobi and Block-Gauss-Seidel preconditioners in a fully matrix-free and in a partially matrix-free setting are investigated. [95] uses a Kronecker product singular value decomposition approach to approximate the jacobian such that it can be evaluated matrix-free. In [35], the use of alternate-direction-implicit and fast diagonalization methods is advocated.

In this work we try to provide the building blocks necessary for a solution procedure outlined in [89]. A Krylov subspace method is used as the outer iterative solver in the solution procedure. Any operator applications within the Krylov subspace method are implemented in a matrix-free fashion instead of doing matrix-vector products of a preassembled matrix. The method is preconditioned with a DG-enabled AMG preconditioner from [16], where block smoothers for the DG space are combined with AMG corrections in a low order subspace. In contrast to [16], [89] implements the block smoothing step in a matrix-free way as well using either Block-Jacobi, Block-Gauss-Seidel or Block-SOR smoothers. Within these smoothers the diagonal blocks of the (never assembled) DG matrix need to be inverted. This is done approximatively using an iterative matrix-free solver with a relaxed convergence criterion.

CHAPTER 3

A Code Generation Toolchain for the DUNE Framework

Section 2.4 provided an introduction to the field of generative programming in the application area of PDEs. In this chapter, we will go into detail of how generative programming can be exploited in the DUNE framework in an HPC-enabled fashion. We will start off by discussing important design decisions in section 3.1. After describing existing software projects and how they are leveraged in our software design in section 3.2, we outline the complete code generation process in section 3.3. We finish the chapter with a description of how code generation can be embedded into the DUNE user workflow.

3.1 Design Decisions

In section 2.4 we listed criteria that are important in the software design of a code generation toolchain: DSLs, embeddedness, scope of code generation, choice of IR and performance optimization decision making. We will now describe our decisions w.r.t. these criteria and the rationale behind them.

Given the popularity of the FEniCS project among researchers in scientific computing, knowledge about the UFL DSL is quite widespread in the community. Reusing UFL as the input language to a code generation toolchain is significantly lowering the bar for users to adopt their codes. We will therefore do this in our approach and describe UFL in detail in section 3.2.1. For similar reasons, other projects are using UFL in a non-FEniCS context as well, e.g. Firedrake. Also, UFL succeeds quite well at providing a DSL that resembles the canonical mathematical form of weak formulations. Writing a new DSL with the same rationale would necessarily

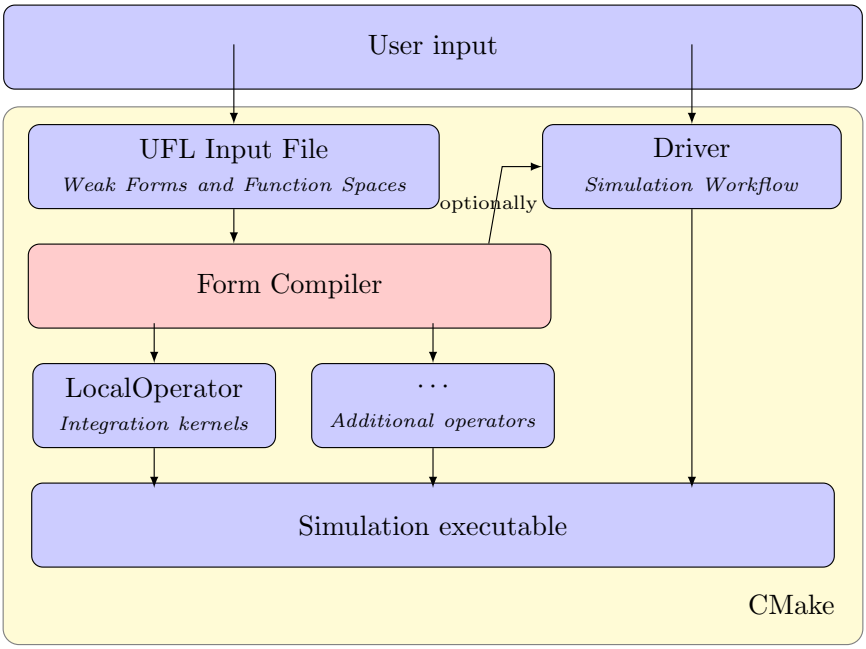


Figure 3.1: Embedding of the code generator into the user workflow: Only innermost loops of finite element assembly are generated from UFL input by the form compiler. It outputs one or more header files each containing a class that fulfills the `LocalOperator` interface. The simulation driver is still provided by the user, although it can optionally be generated as well. The compilation process is governed by DUNE's CMake build system.

lead to a similar language, not making it worth the effort. In fact, the current interest in UFL opens up opportunities for standardization of approaches in FOSS for PDEs.

With UFL being embedded into Python, our code generation toolchain needs to be as well. Given the popularity of Python in scientific computing right now, this should be regarded an advantage of the overall approach. However, we do not intend to fully embed the user workflow into Python, but instead limit the scope of code generation to a collection of innermost loops. We do so in order to preserve the DUNE framework's native strength to provide extensible building blocks for simulation components: Providing Python interfaces to these components makes it harder for non-expert users to extend the framework's capabilities. Nevertheless, such interfaces are currently introduced in a project unrelated to this work [32]. The PDELab interface of the `LocalOperator` class described in section 2.3 is a very good fit to restrict code generation to, as its interface and the abstractions of UFL match quite well. E.g. the assembly methods for facet integrals within the `LocalOperator` class are only given access to data from the two neighboring cells and UFL is by design limited to express exactly this kind of data dependency. Figure 3.1 summarizes how code generation integrates into the user workflow.

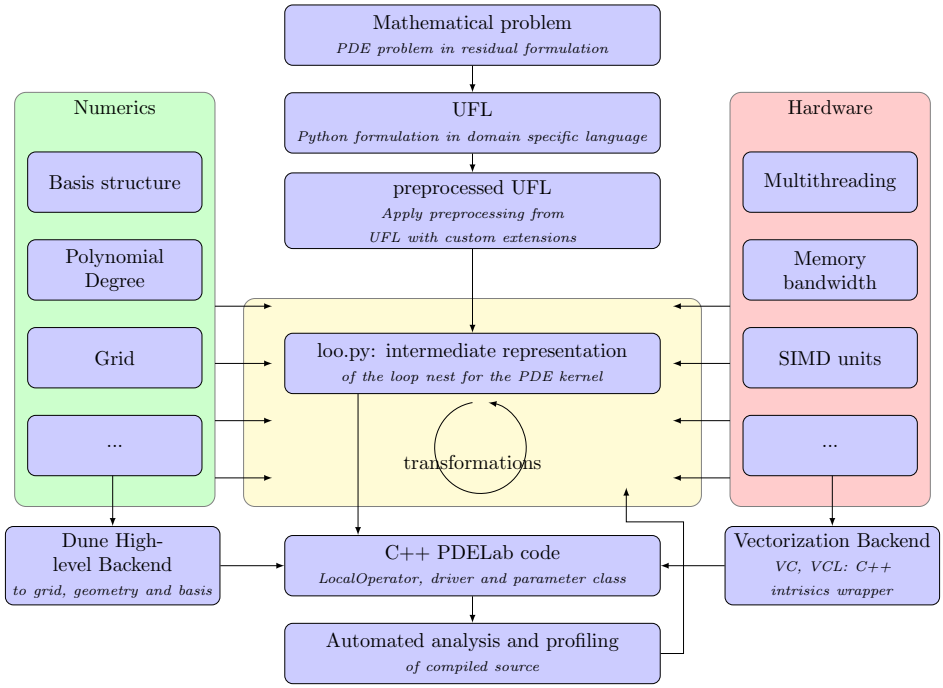


Figure 3.2: The form compiler toolchain at a glance: Input given in the UFL DSL is preprocessed and translated into the `loopy` IR. This IR allows transformations which are motivated from both mathematical concepts and hardware aspects. Autotuning may be used to guide the transformation process in the absence of detailed performance models.

The question of what kind of IR to use within the code generator is strongly tied to the motivation to use code generation in the first place. Our work is driven by the need to exploit knowledge about both the mathematical structure of the given PDE and about the target hardware at code generation time. In this regard, our approach differs from the FEniCS approach, because their main focus is robust code generation and no additional IR that allows reasoning about hardware is employed. Not trying to write the IR from scratch, we chose to use `loopy` which allows symbolic representation of computational kernels and transformation of such kernels with performance optimization in mind. We will study `loopy` in detail in section 3.2.2. The choice of what transformations are applied to the `loopy` IR in order to achieve maximum performance is guided by an autotuning process in our case. This way we do not require users to interact with performance engineering and achieve separation of concern between computational scientists and performance engineers. The absence of this separation of concern is what motivated the introduction of generative programming into DUNE in the first place. Chapter 4 will be concerned with defining a good autotuning search space for the domain of sum factorized finite element assembly. The employed toolchain within the form compiler is summarized in figure 3.2.

Another area where our prerequisites differ from those of the FEniCS and the

Firedrake project that partly rules out reusing their form compilers is the question of target language. In their approaches, the generated C code adheres to an interface which is designed for code generation. In our case, the interface is given by a longstanding project and the code generation toolchain needs to adapt to it. This is especially important when it comes to memory layouts of global data structures: While a purely code generation based approach would have the liberty to choose such layout, it needs to stick to the layout prescribed by the simulation framework in our case. As a consequence, the chosen IR needs to provide some flexibility in expressing memory layouts.

3.2 Involved Tools

The presented code generation toolchain for DUNE leverages many existing software projects. The intention of this section is to present them in a fashion that both provides the reader with the technical knowledge to understand the software design of the code generation process, as well as the hands-on knowledge to use it to implement their own PDE model. Section 3.2.1 covers UFL, a popular DSL for finite element problems, which serves as user interface for code generation. Section 3.2.2 studies `loopy`, a Python package providing a powerful IR for array programming. Finally, section 3.2.3 presents abstraction layer projects for SIMD programming.

3.2.1 UFL - a DSL for Finite Element Problems

UFL has been developed by the FEniCS project as a DSL for finite elements. In section 3.1 we have already argued why we want to reuse UFL as the input language of our code generator. For the rest of this section, we describe the most important UFL components in detail to both enable the reader to implement his own models and to understand the technical aspects of our code generation approach. We highlight similarities, restrictions or differences to PDELab abstractions wherever they occur. All code examples in this section assume that all symbols from UFL have been imported into the global scope with `from ufl import *`. UFL has previously been described extensively in [7].

Multilinear Forms

UFL is about defining multilinear forms of *arity* n, which take the form

$$a : V_1 \times \cdots \times V_n \to \mathbb{R}.$$

The form a is linear in each of its arguments $v_i \in V_i$. However, in finite element practice we usually deal with $n = 1$ (linear forms) and $n = 2$ (bilinear forms). The considered multilinear forms can additionally be parametrized with a number of finite element functions. The form is not necessarily linear in these additional

arguments, which are called *coefficient functions* from now on. In accordance with the UFL nomenclature from [7] we write this down as:

$$\begin{aligned} a : W_1 \times \cdots \times W_m \times V_1 \times \cdots \times V_n &\to \mathbb{R} \\ (w_1, \ldots, w_m, v_1, \ldots, v_n) &\mapsto a(w_1, \ldots, w_m; v_1, \ldots, v_n) \end{aligned}$$

In our intended toolchain, we are focussing on expressing the residual formulation described in section 2.3, which is a linear form. It takes the general form $r(u; v)$, where v is the test function and u is the finite element solution. The residual form is always linear in v and depending on the PDE being linear, either affine linear or nonlinear in u . Of course, r might depend on arbitrarily many additional coefficient functions.

This is a full implementation of a $\mathcal{P}_1$ discretization of the Poisson equation in UFL:

```
1 FE = FiniteElement('CG', triangle, 1)
2 u = TrialFunction(FE)
3 v = TestFunction(FE)
4 f = Coefficient(FE)
5 r = (inner(grad(u), grad(v)) - f*v)*dx
```

In this very condensed minimal example, all the basic building blocks of multilinear UFL forms are present. We will now briefly describe these blocks and then proceed to give detailed information about each of these and their inner workings for the rest of this section:

- Line one defines a finite element e.g. here the continuous $\mathcal{P}_1$ finite element on triangles. The finite element object does only store the given information and does have no knowledge of the implementation of that finite element, e.g. mapping of DOFs to subentities of the reference element or local interpolation.
- Lines two and three define trial and test functions using that finite element.
- Line four defines a coefficient function that is used for the source term f. The function f is expected to be interpolated onto the finite element space defined by `FE`.
- Line five defines a form object. It does so by building an arithmetic expression (potentially using compound tensor algebra operations like an inner product) from the given finite element functions and multiplying it with an integration measure object. The object `dx` indicates that we are expressing a volume integral.

The scope of UFL stops at providing an Abstract Syntax Tree (AST) data structure for the form object `r`. Getting from this AST to generated code is the task of the form compiler developed in this work.

Short	Name	Reference Element	Degree	in FIAT
`'CG'`	`'Lagrange'`	all	$k = 0$	yes
		1D, 2D, 3D simplex	all	yes
		2D, 3D cube	$k = 1, 2$	yes
`'DG'`	`'Discontinuous Lagrange'`	1D, 2D, 3D cube	all	yes
`'GL'`	`'Gauss-Legendre'`	1D, 2D, 3D cube	all	yes
`'DGLL'`	`'Disc. Gauss-Lobatto-Legendre'`	1D, 2D, 3D cube	all	no
`'Monom'`	`'Monomials'`	1D, 2D, 3D cube	all	no
`'OPB'`	`'L2-Orthonormal Polynomials'`	1D, 2D, 3D cube	all	no
`'RaTu'`	`'Rannacher-Turek'`	2D, 3D cube	$k = 1$	no
`'RT'`	`'Raviart-Thomas'`	2D cube	$k = 0, 1, 2$	yes
		2D simplex, 3D cube	$k = 0, 1$	yes
`'BDM'`	`'Brezzi-Douglas-Marini'`	2D simplex/cube	$k = 1$	yes

Figure 3.3: A list of finite elements that are currently available from PDELab and how they can be accessed from UFL.

Finite element spaces

UFL contains a description of finite elements which only serves the purpose of annotating the symbolic representation of a finite element function with all the information needed within UFL. This does not incorporate the actual implementation of basis function evaluation, mapping of degrees of freedom to subentities of the reference element and local interpolation. Within the FEniCS/Firedrake toolchain these tasks are handled by FIAT [68], which has it's own, extended description of the finite element. Finite elements in UFL are grouped into element families, such that elements within a family only vary in polynomial degree and reference element. The following properties are stored for each such family:

- *Family Name* A descriptive name of the finite element family.
- *Short Name* An abbreviation to identify the family.
- *Sobolev Space* The Sobolev space the space defined by the element is contained in. For elements in UFL, the space is typically one of L^2, H^1, H^2, $H(div)$ or $H(curl)$.
- *Rank* An integer value specifying the tensor rank of the finite element's range, typically 0 (for scalars), 1 (for vectors) or 2 (for matrices).
- *Mapping* A string identifying the pullback transformation for this element. This is typically `'identity'` or e.g. in the case of a Raviart-Thomas element, a more involved mapping like `'contravariant Piola'`.

The finite elements available from PDELab and how they can be accessed from UFL are summarized in figure 3.3.

Space Dimension	Simplex Cell	Cube Cell
0	`vertex`	`vertex`
1	`interval`	`interval`
2	`triangle`	`quadrilateral`
3	`tetrahedron`	`hexahedron`

Figure 3.4: A list of reference elements available in UFL as named objects. Additional reference elements beyond these need to be constructed by building tensor products of these building blocks.

In order to define a finite element, specification of the reference element - *cell* in UFL terminology - is needed. UFL uses simple named objects to identify basic reference elements, with the added convenience of accepting strings containing that same name. The available cells are restricted to simplices and cubes in space dimensions 0 to 3, as shown in figure 3.4. Given the desired element family, the reference element and the polynomial degree, a finite element can be constructed:

```
FE = FiniteElement('CG', triangle, 1)
```

When dealing with systems of PDEs, finite elements for each system component need to be combined into a larger data structure. As highlighted in section 2.3, this data structure should exhibit a tree structure in order to allow for composability and optimally blocked linear algebra data structures. UFL supports the construction of such trees based on `FiniteElement` leaf nodes through the following three classes:

- `VectorElement`: Given a leaf element and a dimension (or all the parameters needed for a finite element and a dimension), constructs an element for a vector field. The resulting element builds a tree structure from several components of the given leaf elements. This mirrors the tree structures for finite element spaces in PDELab exactly. Note, that using `VectorElement` differs from using vector-valued finite elements in the first place.
- `TensorElement` extends the concept of `VectorElement`: Instead of a dimension, a shape tuple is given e.g. to discretize a stress tensor in solid mechanics. Tensor elements allow specification of symmetries as a Python dictionary which contains pairs of indices of identified tensor entries. E.g. `symmetry={(0,1): (1,0)}` describes a symmetric 2×2 matrix. The resulting element is flattened out and components redundant due to symmetries are omitted.
- `MixedElement`: Given an arbitrary list of finite elements, a mixed element combines these into a larger finite element. This operation is also available through the overloaded multiplication operator of finite elements, although this does not allow for an element to have more than two child elements.

The following example shows the construction of a continuous $\mathcal{P}_2/\mathcal{P}_1$-Taylor-Hood-Element on quadrilaterals:

```
FE_V = VectorElement('CG', quadrilateral, 2, dim=2)
FE_P = FiniteElement('CG', quadrilateral, 1)
TH = MixedElement(FE_V, FE_P)
```

The number of available reference elements and finite elements in UFL increased beyond the already listed through the introduction of tensor product cells and elements [86]. Arbitrary tensor product reference elements can be constructed through the `TensorProductCell` class using the basic building blocks from figure 3.4. Additionally, tensor product elements can be constructed from arbitrary finite elements using the `TensorProductElement` class. In fact, the elements `TE1` and `TE2` in the following snippet are equivalent:

```
cell = interval
product_cell = TensorProductCell(cell, cell)
FE = FiniteElement("DG", cell, 1)

TE1 = FiniteElement("DG", product_cell, 1)
TE2 = TensorProductElement(FE, FE)
```

Tensor product elements are useful for the definition of elements over non-simplex non-cube reference elements, such as prisms. They also allow elements with anisotropic polynomial degree, where the degree property becomes a tuple. Using a tensor product element also allows to indicate tensor product structure to a form compiler, which can then algorithmically exploit it.

UFL supports more manipulation of existing finite elements which may be interesting to leverage in the future: `EnrichedElement` takes a tuple of elements and combines their degrees of freedom. This is used in practice by methods like Extended Finite Element Method (XFEM) [24]. Enriched elements are also available through the addition operator of finite elements. `RestrictedElement` allows to remove degrees of freedom by specifying a restriction domain, that degrees of freedom need to be associated with to be kept. This restriction domain can be one of `'interior'`, `'facet'`, `'edge'` or `'vertex'`. Such serendipity elements are commonly used in solid mechanics. Of course, UFL only describes these finite elements, the actual implementation is up to the assembly framework.

The finite elements described in this section map quite cleanly onto the PDELab concept of a `FiniteElementMap` from section 2.3. However, these elements do not fully describe the discrete finite element function space, as there is no notion of a mesh. This enhanced concept exists in UFL under the name of `FunctionSpace`, which combines a *domain* object and a finite element. Domain objects are implemented through a `Mesh` object, whose scope is however not comparable to the scope of a DUNE grid view object, as it only stores a finite element that describes the nature of the coordinate transformation. For the sake of simplicity, we therefore omit these function spaces and domains from our examples. Technically this means,

that we are automatically constructing domains from cells and function spaces from finite elements respectively.

Building UFL expressions

We will now look into how arithmetic expressions are constructed within UFL. Right now, we will only investigate this from an end-user perspective and leave the question of how the resulting ASTs look like for later in this section.

The most important basic building block of arithmetic expressions for weak formulations of PDEs are functions from finite element spaces. These are used for test and ansatz functions, but may also appear in more scenarios, such as solutions from another PDE in a weakly coupled system of PDEs or a physical parameter interpolated into a finite element space. UFL provides three user-facing classes to create such finite element functions: one each for test, ansatz and coefficient functions:

```
FE = FiniteElement('CG', triangle, 1)
u = TrialFunction(FE)
v = TestFunction(FE)
c = Coefficient(FE)
```

If the given finite element is of mixed nature, finite element functions can be split using the UFL function `split`. It returns a tuple of UFL expressions that reflects the structure of the `MixedElement`:

```
FE_V = VectorElement('CG', triangle, 2)
FE_P = FiniteElement('CG', triangle, 1)
TH = FE_V * FE_P
u, p = split(TrialFunction(TH))
v, q = split(TestFunction(TH))
c, d = split(Coefficient(TH))
```

UFL provides an additional shortcut for this splitting in the form of `TestFunctions(TH)` (mind the plural!) etc. With UFL being embedded into Python, float and integer literals can directly be plugged into UFL. These are however automatically wrapped into UFL types representing these literals. This can be explicitly controlled by calling the idempotent function `as_ufl`. Using Python literals for constants results in the constant being directly written into generated code. There is however also a concept to define runtime constants through the `Constant`, `VectorConstant` and `TensorConstant` classes. How exactly these are assigned their proper runtime value is not in the scope of UFL and might differ between problem solving environments.

Using these basic building blocks, we can compose mathematical expressions using the most common operators and functions. With UFL being embedded into Python,

arithmetic operators (`+`, `-`, `*`, `/`[1]) are implemented through operator overloads. Mathematical functions are provided using their canonical name from mathematics, e.g. `exp`, `ln`, `sqrt`, all trigonometric functions and more complex functions like Bessel functions. The power function is available through both the `**` operator and the built-in `pow` function. The functions `max` and `min` cannot be intuitively defined, as they would override the corresponding Python built-in functions. Therefore minima and maxima must either be expressed using the `max_value` and `min_value` functions or their `Max` and `Min` aliases. A three-argument `conditional` function allows to implement branching in the form of

$$\texttt{conditional(cond, A, B)} = \begin{cases} \texttt{A} & \texttt{cond is True} \\ \texttt{B} & \texttt{cond is False} \end{cases}$$

In order to express the conditions for the first argument to `conditional`, the DSL comprises a subset for logical expressions. Comparison operators can be accessed either through Python operator overloads or by using their named form as `eq`, `ne`, `le`, `ge`, `lt` and `gt`. Defining the logical operators for conjunction, disjunction and negation again suffers from a name clash with Python language keywords. The DSL therefore uses the uppercase function names `And`, `Or` and `Not` instead.

UFL has symbolic differentiation capabilities built in. It can both be explicitly controlled by the user when specifying a form or used by a form compiler during form processing. We will now focus on the user perspective and describe the form compiler use cases later on. Common differential operators from tensor analysis are available under their canonoical names, like `grad`, `div` and `curl` (or `rot`). These use the convention that the newly created tensor axis is appended to the existing tensor. There is a prepending version available under the names `nabla_grad` and `nabla_div`. If only a specific partial derivative is needed, it can be accessed through `expr.dx(d)` or `Dx(expr, d)`. UFL expressions can also be differentiated w.r.t. user-defined expressions, e.g. to implement sensitivity analysis. To do so, that expression needs to be wrapped within the `variable` function. One can then use `diff(expr, var)` to differentiate any expression `expr` that depends on the wrapped `var` object.

So far, several ways to construct tensors directly from mathematical concepts have been described. However, tensors can also be constructed manually, by using the helper functions `as_vector`, `as_matrix` and `as_tensor` which take a (nested) iterable of entries and construct a tensor from it. This example constructs an identity matrix from Python generators, though the same result can be achieved by using the more convenient `Identity(n)` from UFL:

```
I = as_matrix([[int(i == j) for i in range(n)] for j in range(n)])
```

[1] Even when used with Python 2, UFL never implements the division operator as integer division. One should be careful to not accidentally do it in user code though e.g. by writing `1/2*expr`. Instead write `0.5*expr`.

Math	UFL	Sum notation
$A + B$	`A + B`	$C_{i_0 \cdots i_{n-1}} = A_{i_0 \cdots i_{n-1}} + B_{i_0 \cdots i_{n-1}}$
$A \cdot B$	`dot(A, B)`	$C_{i_0 \cdots i_{n-2} j_1 \cdots j_{m-1}} = \sum_{i_{n-1}} A_{i_0 \cdots i_{n-1}} B_{i_{n-1} j_1 \cdots j_{m-1}}$
$A : B$	`inner(A, B)`	$C = \sum_{i_0} \cdots \sum_{i_{n-1}} A_{i_0 \cdots i_{n-1}} B_{i_0 \cdots i_{n-1}}$
$A \otimes B$	`outer(A, B)`	$C_{i_0 \cdots i_{n-1} j_0 \cdots j_{m-1}} = A_{i_0 \cdots i_{n-1}} B_{j_0 \cdots j_{m-1}}$
$A \times B$	`cross(A, B)`	$C_k = \sum_i \sum_j \epsilon_{ijk} A_i B_j$
$A \circ B$	`elem_mult(A, B)`	$C_{i_0 \cdots i_{n-1}} = A_{i_0 \cdots i_{n-1}} B_{i_0 \cdots i_{n-1}}$
$A \oslash B$	`elem_div(A, B)`	$C_{i_0 \cdots i_{n-1}} = \frac{A_{i_0 \cdots i_{n-1}}}{B_{i_0 \cdots i_{n-1}}}$
$A^{\circ B}$	`elem_pow(A, B)`	$C_{i_0 \cdots i_{n-1}} = A_{i_0 \cdots i_{n-1}}^{B_{i_0 \cdots i_{n-1}}}$

Figure 3.5: A list of tensor algebra operations in UFL. Whenever indices appear on both input tensors A and B the shape of these tensor axes is assumed to match. Indices on the output tensor C describe the output shape depending on the shape of A and B.

Mathematical name	UFL	Remark
determinant	`det(A)`	
transposition	`transpose(A)` or `A.T`	
diagonal part	`diag(A)`	returning a matrix
diagonal part	`diag_vector(A)`	returning a vector
trace	`tr(A)`	
inverse	`inv(A)`	
cofactor matrix	`cofac(A)`	
symmetric part	`sym(A)`	$sym(A) = 0.5 \cdot (A + A^T)$
skew-symmetric part	`skew(A)`	$skew(A) = 0.5 \cdot (A - A^T)$
deviatoric part	`dev(A)`	$dev(A) = A - \frac{tr(A)}{tr(\mathbb{I})}\mathbb{I}$

Figure 3.6: A list of matrix operations in UFL

Tensor entries can be accessed using the square bracket operator of the tensor object with an index tuple. The resulting expression has all the necessary knowledge to be treated as a UFL expression in its own right, making the tensor language in UFL fully composable. Explicit creation of indices and reductions over indices is - while of course possible - not advisable for end users and therefore not decribed here.

We will now continue with a description of the tensor algebra operations available in UFL. Binary operators known from tensor algebra, such as inner products, dot products etc. and how they are available from UFL are summarized in figure 3.5. On top of these, several linear algebra operations available only on matrices are summarized in table 3.6.

A lot of numerical techniques explicitly depend on evaluation of geometric quantities (as opposed to implicitly depending on it through transforming integrals to the

reference element). Examples are DG methods, where penalty terms may depend explicitly on the mesh width. For this purpose, UFL provides the user a variety of geometry objects, which are class objects constructed given the domain (or in our case, the cell):

- A `SpatialCoordinate` object describes a global coordinate. In the context of quadrature-based integration, this is the coordinate of the current quadrature point. We will later see how quadrature is realized in the code generation toolchain.
- A `FacetNormal` object describes a unit normal vector of a facet in the mesh. This is only well-defined when combined with a restriction to either the inside or outside cell, which we will cover later in this section. An analogon for cells is only useful for manifold meshes, which we currently do not support.
- Basic geometric properties of the cell can be accessed through the self-explanatory names `CellVolume`, `CellDiameter`, `Circumradius`, `MinCellEdgeLength` and `MaxCellEdgeLength`.
- Similarly, basic properties of a facet are available as `FacetArea`, `MinFacetEdgeLength` and `MaxFacetEdgeLength`.
- The jacobian of the geometry mapping from the reference element and some derived quantities can be explicitly accessed through the `Jacobian`, `JacobianInverse` and `JacobianDeterminant` nodes. However, this is usually not necessary, as the transformation of the integral to the reference element is done automatically by the code generator.

Note, that UFL only describes cell-local operations. Consequently, all these geometric properties are evaluated by the form compiler or the problem solving environment for the current cell in the grid loop. Accessing such information from neighboring or arbitrary cells is not possible, with the notable exception of cell information from adjacent cells being accessible on facet integrals by applying suitable restrictions.

UFL has support for implementing DG methods built in [93]. In DG, integrals over interior facets in the mesh appear. Finite element functions and cell quantities are discontinuous and therefore two-valued on such a facet. UFL disambiguates these by adding *restrictions* to those expressions. The inside and outside cells are identified by the strings `'-'` and `'+'`. A restriction is applied to an arbitary expression through the round bracket operator: `expr('+')`. As described in section 2.2, DG methods are typically formulated using jump and averaging operators. These are implemented in UFL as well and allow a very brief syntax without explicitly restricting all quantities:

```
r = avg(u)*jump(v)*dS
```

Integration Measures

The multilinear forms arising from the discretization of PDEs are given by integrals over the domain, the domain boundary and over mesh facets in the interior of the domain. These three types of integrals are represented in UFL by multiplying an UFL expression that represents the integrand with a measure object from the right. Measure objects do not need to be instantiated by the user, but are provided by the language: Cell integrals are realized by `dx`, boundary facet integrals by `ds` and integrals over interior facets by `dS`. Multiplying with such a measure from the right constructs an integral object, which is wrapped in a form object. Form objects can be constructed by adding other form objects using the addition operator. Sums of integrals over matching measures are joined by adding the integrands during UFL's preprocessing step described in section 3.2.1. Note that the limitation to cell, boundary and interior facet integrals is a limitation of the language. Fortunately, it matches the limitations of our problem solving environment PDELab and of many other finite element discretization packages..

The measure object does not only describe the geometric integration domain, but it also allows to express restrictions of an integral to a subdomain. The most common use case for this is to define heterogeneous boundary conditions, composable multi-domain simulation might be another one. In order to restrict a measure to a subdomain, a pair of parameters called *subdomain ID* and *subdomain data* needs to provided. To this end, each subdomain is assigned an integer ID. A measure can be customized by providing a single such ID or a tuple thereof, if the integration domain is the union of multiple subdomains. The subdomain ID of a measure defaults to the magic string `'everywhere'`, which represents the union of all subdomains. The subdomain data used in measure customization is an object that UFL does not further interpret which tells a form compiler how the subdomain can be characterized. In our approach, this is a UFL expression, which evaluates to the subdomain ID if and only if the current quadrature point belongs to the subdomain. The easiest way to construct a restricted measure is to call an existing measure's round bracket operator with all the named keywords to override:

```
dx_sub = dx(subdomain_id=subdomain_id,
            subdomain_data=subdomain_data)
```

Note that using this direct tweaking of measure objects is not how boundary conditions are handled in other UFL-based problem solving environments, such as dolfin. Instead, these use higher level concepts that wrap around this measure modification.

We will now give examples of how to customize boundary integral measures. In the above example with the Poisson equation, we have so far assumed essential Dirichlet boundary conditions, which are realized at the function space level by constraining the boundary degrees of freedom. Instead applying mixed Dirichlet/Neumann boundary conditions requires us to define a boundary measure for $\Gamma_N \subseteq \partial\Omega$ and restrict integration of the Neumann boundary flux j to it. For this example we

assume the domain $\Omega \subset \mathbb{R}^2$ to be a rectangular domain and impose the Neumann boundary condition on the left and right boundaries:

```
class BCType:
    Dirichlet = 0
    Neumann = 1

n = FacetNormal(triangle)('+')
bc_select = conditional(abs(n[1]) < 1e-8,
                        BCType.Neumann,
                        BCType.Dirichlet)
ds_neumann = ds(subdomain_id=BCType.Neumann,
                subdomain_data=bc_select)
r = (inner(grad(u), grad(v)) - f*v) * dx
  + j*v*ds_neumann
```

Introducing the enum-like structure `BCType` is not strictly necessary, but should be preferred over implicitly defining such a mapping for the sake of maintainability and readability. In this example, we used the normal vectors y-component to identify the left and right face of the domain. Alternatively, one might use global coordinates - hardcoding the domain extents (here to $\Omega = (0, 1)^2$):

```
x = SpatialCoordinate(triangle)
bc_select = conditional(Or(x[0] < 1e-8, x[0] > 1 - 1e-8),
                        BCType.Neumann,
                        BCType.Dirichlet)
```

For more complicated domains, conditionals like this might be impossible or infeasible to write down. We therefore present an additional solution which relies on storing the boundary condition type in a vector. We (ab)use the $\mathcal{P}_0$ finite element to define element-wise constant boundary condition types.

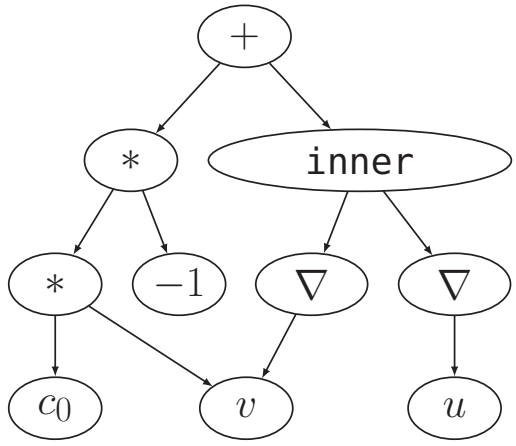
```
bc_select = Coefficient(FiniteElement('DG', triangle, 0))
```

This assumes that the boundary condition data has been correctly interpolated into this coefficient function. Typically, this will happen in C++ under user control, where the user can implement methods out of the scope of UFL, such as a lookup of data parsed from a Gmsh file [46].

Attaching subdomain information is not the only way to modify existing measures. They also allow to attach a dictionary of arbitrary meta data to a measure like this:

```
dx = dx(metadata={'quadrature_order': 42})
```

UFL does not process this data beyond distinguishing measures that only differ in meta data. Meta data can be used to customize form compiler behaviour directly and is highly form compiler-specific. We are not currently supporting measure meta data in our form compiler, but we will add this functionality in the near future.




```
Sum(
  Product(
    IntValue(-1),
    Product(
      Argument( ... ),
      Coefficient( ... ))),
  Inner(
    Grad(
      Argument( ... ))
    Grad(
      Coefficient( ... ))))
```

Figure 3.7: Tree structure of an UFL expression by example: To the left, the tree structure of an expression for the integrand of the Poisson equation is shown, to the right a simplified version of the Python type realizing this tree.

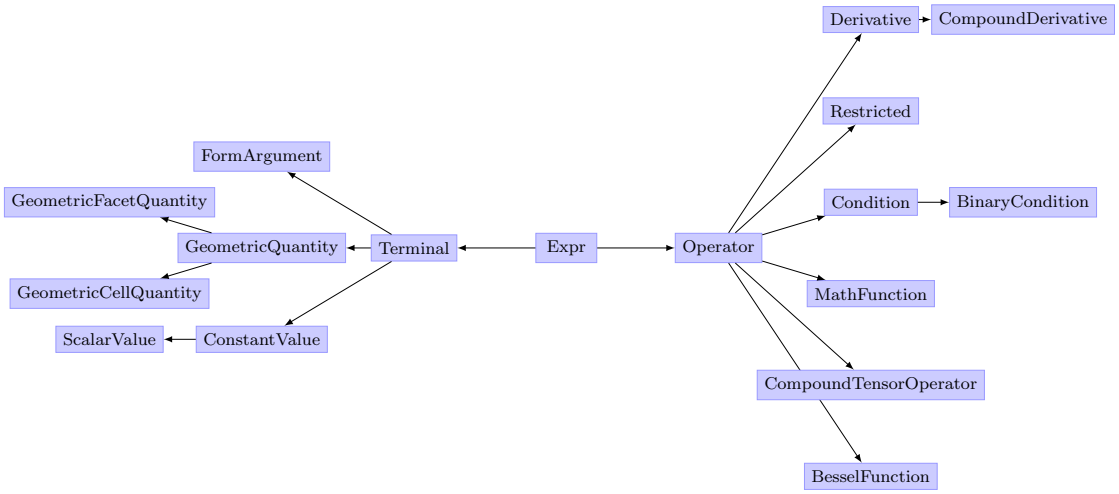


Figure 3.8: An overview of the abstract class hierarchy of UFL expression nodes. The `Expr` node is the root node. Using these base classes, UFL defines a total of 115 non-abstract nodes.

UFL implements additional measures beyond the mentioned volume, boundary and skeleton integrals. As these are tuned specifically to applications in cut-cell methods, we are not covering them here. For future applications that require additional annotation of the integration measure, the possibility of creating custom measures exists.

Expression Trees

So far, we described the user interface of UFL. We will now turn to details about the UFL IR, which are necessary to understand the code generation process.

UFL expressions are internally represented as an AST. ASTs are typically used in programming language and compiler design to represent program semantics. A simple example of a UFL AST is shown in figure 3.7. In the following, we will study some relevant classification of the node types used in the UFL AST:

- *Abstract vs. non-abstract nodes*: With UFL being embedded in Python, class polymorphism is used for node types. A hierarchy of base classes, as shown in figure 3.8 is used to classify nodes. The root of the inheritance hierarchy is the `Expr` node, which defines the interface for all nodes and provides the operator overloads that implement the seamless embedding into the Python language. Inner nodes of the inheritance tree are called abstract node types. UFL currently comprises a total of 115 non-abstract node types based on this hierarchy.
- *Leaf nodes* (or *terminal nodes*) are where the finite element application domain of UFL is obvious: These are either finite element functions, geometric quantities or literals. The inner nodes in contrast implement a fairly generic symbolic math language. In contrast to other symbolic languages like e.g. `sympy` [88], UFL tries to preserve the mathematical structure as much as possible - meaning that (if the form compiler follows a similar policy) arithmetic expressions in generated code will resemble the input.
- Some inner nodes are further classified as so called *terminal modifiers*, although this classification for historical reasons does not appear in the inheritance tree from figure 3.8. These terminal modifiers ideally only occur as direct parents of terminal nodes or other terminal modifiers. Examples are the differential operators `Grad` and `ReferenceGrad`, the facet integral restriction operators `PositiveRestricted` and `NegativeRestricted` and the indicator node for evaluations in local coordinates `ReferenceValue`. In user code, these nodes can appear everywhere, so one challenge in expression preprocessing will be to propagate these towards leaf nodes.
- *User-facing node types vs. code-generator-facing node types*: The user input previously described is translated into nodes almost one to one. However, some of these nodes express a high-level mathematical concept, that is not useful in the code generation process. This applies especially to the tensor algebra nodes listed in figure 3.5 and 3.6, which we will seek to rewrite into their low-level forms introducing reductions later on.

Visitor Patterns for UFL Expressions

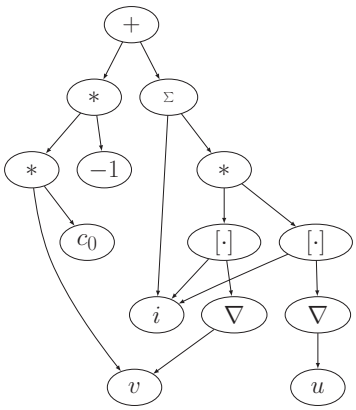
We mentioned the necessity of expression modification and rewriting in several places already. Such transformation algorithms are implement using a tree visitor design pattern [43] ensuring a clean separation of the AST data structure and the algorithms working on it. An expression tree is traversed depth-first, calling a

function on each node, which is called a *handler*. The handler is selected using type-based function dispatch. All the handlers that constitute a visitor algorithm are gathered in a class which inherits from the design pattern base class `MultiFunction`. During its construction, a handler cache is built using a naming convention to automate the handler dispatch process: For a given node class name its "snake case" (lower case with underscores between words) version is used as the handler name. In type-based function dispatch, not only the tree node is inspected, but also its method resolution order available from Python internals. This allows to define handlers on abstract node types as well and ensures that the most specialized handler for given node type is selected. When traversing an AST, there is several possibilities regarding the traversal order: Pre-order traversal visits parent before children, post-order traversal children before parent and in-order traversal visits the parent after visiting the first of two children. In UFL, all of these traversal methods can be implemented from within a handler, as the recursive call for the child nodes is implemented by the programmer. Such handlers need to have exactly one argument: the AST node to process. When a `MultiFunction` is called with such recursive handlers, the Python call stack depth will be of the order of the depth of the AST. However, there is also an entirely different way of writing handlers in `MultiFunction`s to achieve a superior version of post-order traversal: If a handler takes more arguments than just the AST node, the traversal algorithm passes the return values of the visit of the children to the handler. The visiting process is then controlled by a Directed Acyclic Graph (DAG) traversal algorithm, which makes sure that identical subtrees are only visited once. Finally, mixing of these two traversal algorithms is possible, by defining single handlers (so called *cutoff types*) that need pre-order visiting, which will cause the DAG traversal to treat the entire AST node as inseparable. The `MultiFunction` distinguishes cutoff types by inspecting its argument list. Figure 3.9 shows an example of the available tree traversal paradigms.

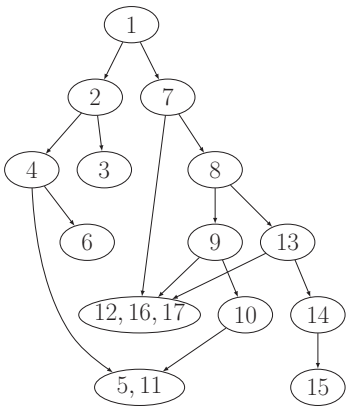
Algorithms on UFL Forms

The raw user input is processed by several algorithms provided by UFL before it is fed into the form compiler toolchain. We will mention these algorithms in this section, shortly outlining the transformation behaviour and the intent of the algorithm. Application of these algorithms (and especially the application order) is controlled by the entry point `compute_form_data` from the `ufl.algorithms` package.

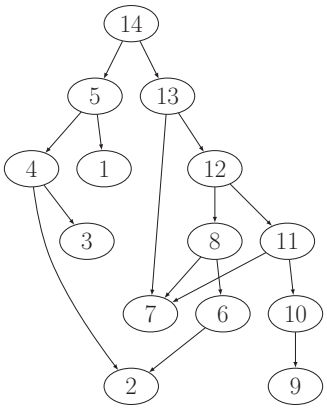
User input may apply restriction operators to arbitrary expressions as long as these are not already restricted. In preprocessing the resulting `Restricted` nodes are propagated towards terminal nodes by recursively applying them to all child nodes. This process is necessary, because the code generation handler for a terminal node is influenced by the presence of a restriction terminal modifier.



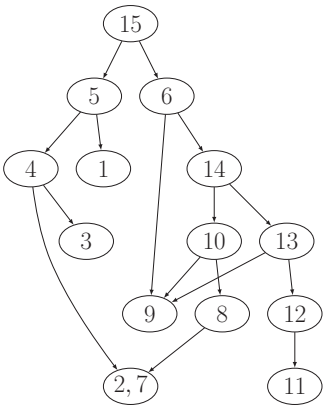
(a) Preprocessed UFL AST describing the integrand of the volume integral of the Poisson equation: $(\nabla u, \nabla v) - c_0 v$.



(b) Iteration order for pre-order/top-down tree traversal. Multiple visits of multi-parented nodes are possible.



(c) Iteration order for post-order/bottom-up DAG traversal. Multiple visits are prevented.



(d) Iteration order for post-order tree traversal with the indexed sum $\sum$ being a cutoff type.

Figure 3.9: Summary of iteration methods for UFL trees: Figure 3.9a shows a preprocessed UFL AST for the Poisson equation used to study iteration order in figures 3.9b to 3.9d. With $[\cdot]$ we denote an `Indexed` note, which implements indexing of a tensor.

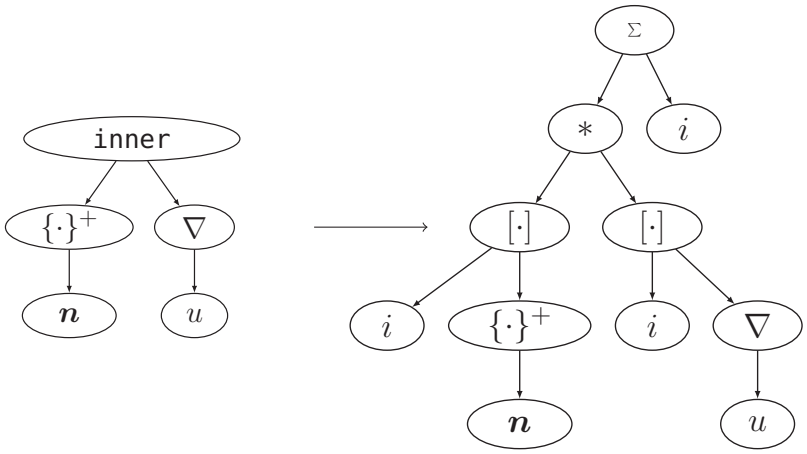


Figure 3.10: Example of how the inner product $(\nabla u, \boldsymbol{n})$ is simplified to a tensor expression in UFL. Inner products of higher-dimensional tensors will produce a chain of reduction nodes.

The tensor algebra operations described in figures 3.5 and 3.6 describe mathematical operations very accurately. However, when it comes to code generation, these abstractions are not the optimal choice of IR. Even if the underlying problem solving environments provided interfaces for these high-level tensor operations, generating code against such an interface would introduce code optimization barriers by hiding the innermost loops in library code. We therefore seek to rewrite these tensor operations into simple arithmetic operations. In these rewrites two UFL node types not explicitly highlighted so far play a key role: The `IndexSum` node realizes a reduction over a single index, which is introduced by the node. The `ComponentTensor` node introduces a new tensor whose internal indexing is completely separated from the outer indexing. Such a node is a key ingredient for composability of tensor expressions. In the figures, we represent the component tensor node by $(\cdot)[\cdot]$. The algebraic simplification algorithms are exemplified in figure 3.10 and 3.11. While the algebraic simplifications might actually look more complicated to the human eye, they are much better suited for handling in a form compiler toolchain due to their much smaller set of nodes and the closeness to multi-dimensional array programming.

Another important algorithmic task in the scope of UFL is application of the integral transform from the world space geometry cell to the reference cell. The functionality of this pullback is split out over two separate passes. A *function pullback* transforms all the coefficient functions and form arguments into the reference frame, which might involve complex mappings in the case of vector-valued finite elements. In most cases, it only wraps a `ReferenceValue` node around the terminal node though. After that, an *integral scaling* transformation multiplies the integrand with the jacobian determinant and a quadrature weight.

The above two transformations may introduce a large variety of geometric quantities.

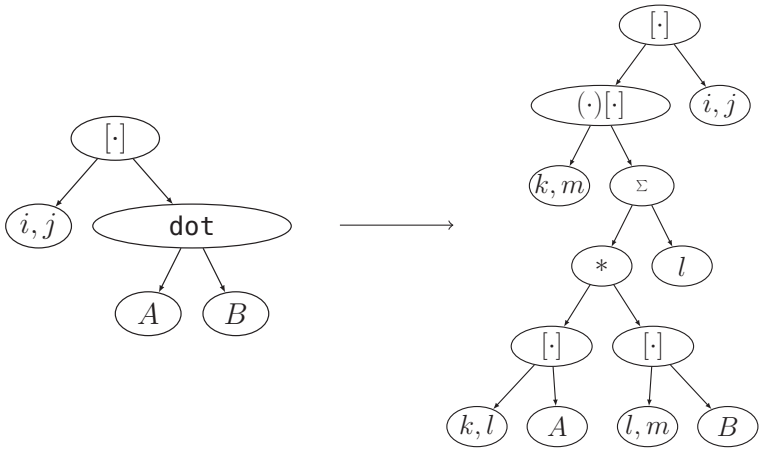


Figure 3.11: Example of how a dot product is simplified to a tensor expression in UFL. A and B are arbitrary input tensors that will most likely be realized by component tensors themselves.

However, different problem solving environments might have different policies on the granularity level of geometric quantities they provide. For example, dune-grid provides the transpose of the inverse of the jacobian as a first class citizen of a generic geometry, where other packages might want to invert the jacobian within the generated code. For this reason, a *geometry lowering* transformation is applied, which can be passed a list of *cutoff types*, which are not simplified further. We use this to specify the inverse of the jacobian inverse as a cutoff type in order to prevent the introduction of a symbolic matrix inversion being added to our UFL expressions.

UFL comes with differentiation capabilities built in. There are three main ways of doing differentiation in computer programs: Automatic Differentiation (AD), symbolic differentiation and numerical differentiation. Numerical differentiation is supported by our problem solving environment PDELab at the cost of a substantial runtime increase due to the increased number of necessary residual evaluations. The classification of UFL's differentiation algorithms is a bit confusing because of it's dependence on the frame of reference. Looking from the Python perspective, the algorithms implement forward AD as described in the literature [51]. However, looking from the perspective of generated C++ code, the differentiation would be classified as symbolic. Being aware of this ambiguity, we use the term AD from now on.

There is a total of four distinct types of differentiation operations in UFL: the gradient, the gradient on the reference element, differentiation w.r.t. a user-defined expression and Gateaux differentiation w.r.t. a coefficient function. Note that some additional mathematical differential operators are expressed in terms of the above though during algebraic simplification e.g. the divergence is rewritten in terms of the gradient. We have not covered Gateaux derivatives so far and will

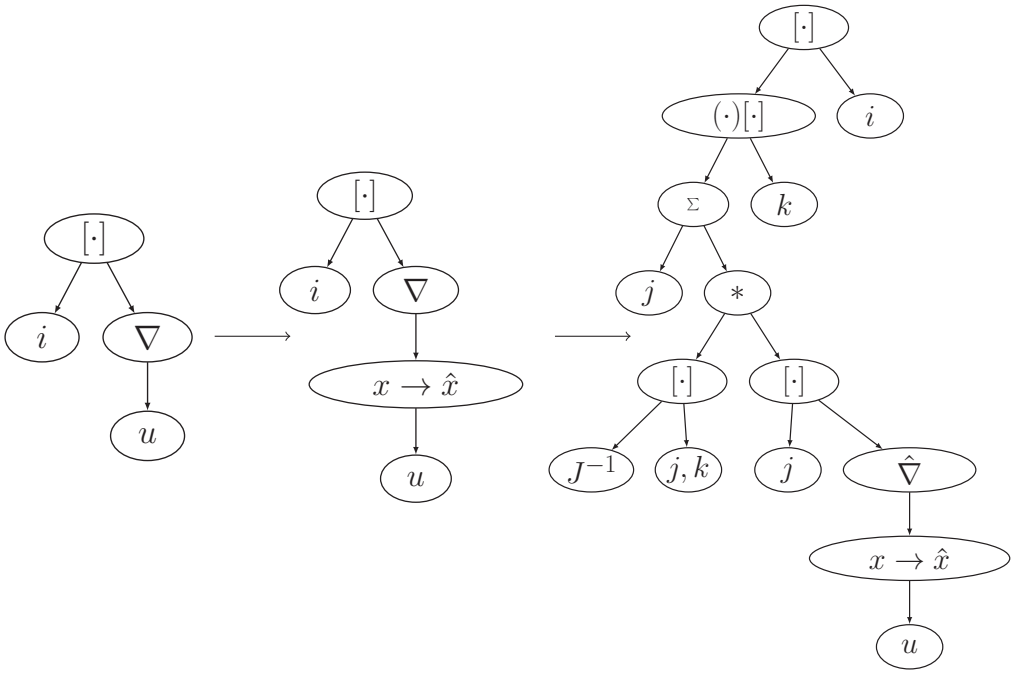


Figure 3.12: Pullback transformation of the gradient of a finite element function: The pullback transformation marks the finite element function for treatment in the reference frame, automatic differentiation actually transforms the gradient operator to the gradient operator in the reference frame. This tree transformation is equivalent to the mathematical transformation $(\nabla u)_i = \sum_j J_{ij}^{-T} (\hat{\nabla} \hat{u})_j$.

postpone this to section 3.3.1, where we will study them in the light of our typical use case. The tree visitor that removes differential operators from the input applies a two-step procedure: An outer visitor looks for differential operators and then dispatches to an inner visitor that handles this type of differential operator. These additional visitors all inherit from a base class that implements basic differentiation rules like the chain and product rule. Figure 3.12 shows an example of how the gradient is applied to a finite element function after the pullback is applied. This also illustrates that the order of preprocessing transformations can be a delicate issue, as the pullback transformation requires an additional application of the AD transformation.

There are two more algorithmic operations on UFL forms that are relevant for our applications: The `action` transformation creates a form that describes the action of a n-form on a vector as an $(n-1)$-form. We will discuss this transformation later on in our application context: Deriving symbolic descriptions of the matrix-free application of the jacobian. The `adjoint` operator allows to symbolically derive the adjoint of a PDE, which plays a key role in PDE-constrained optimization. Although we are not covering this in this work, such transformation adds substantial value to the overall approach as it might be quite tedious to manually derive and implement

these forms for nonlinear problems.

3.2.2 Loopy

We will now describe the `loopy` project, a programming model for array computations embedded into Python. `loopy` will serve as the IR for our code generation toolchain and provides the key ingredient for the generation of C++ code in our application. It has been proven to be capable of handling the complexity of PDE applications [73]. Our description of `loopy` is loosely based on the author's description in [71] though we put emphasis on topics relevant for our intended toolchain. For the remainder of this chapter, we will use a different notion of the term *user*: When talking about users of `loopy`, we mean developers of code generators such as ourselves, whereas in other chapters we were referring to users of our code generator. This also means that whenever something is *user defined*, we will later seek to define it automatically in our code generator.

Design Decisions

`loopy`'s main goal is to solve the performance portability problem of modern hardware, where many mathematically equivalent representations of a computation result in code variants that vary drastically w.r.t. to their performance. Furthermore, the optimal code variant highly depends on the given hardware. `loopy` asks the user to specify the computation by providing *one* such representation, which is usually the one that follows mathematical notation the closest. Given this representation, users may provide a sequence of transformations that transforms it into the representation which is best suited for the given hardware. These transformations may either come from `loopy`'s builtin transformation library or be user-provided. `loopy`'s data model is based on three core components: A tree of polyhedra describing the loop domain(s), a collection of statements describing the computations which are tied to one node of the loop domain and a collection of arrays. We will look at the data model and transformations in more detail later on.

It is a deliberate design decision of `loopy` to not provide any automatic code transformation. Instead, generated code solely depends on user input in the form of statements, loop bounds and a transformation sequence. `loopy` merely implements the explicitly requested behaviour. This design principle rules out internal use of many Python packages for symbolic expressions, such as `sympy` [88], which unconditionally rewrites arithmetic expressions into a canonical form. This canonical form will not necessarily translate into the best possible code. With transformations being completely user-driven, final responsibility for the applicability of a transformation and correctness of the resulting code also remains with the user. This allows the `loopy` transformation space to be a superset of the optimization search space of a conventional compiler, because `loopy` transformations do not necessarily need to preserve the semantics of the generated C++ code. In other words, there are `loopy` transformations that have no valid equivalent in

the optimization step of a C++ compiler. As an example, we mention changing the memory layout of data structures that are exposed to the user. We consider this a defining feature of `loopy`, which is necessary for our memory layout based transformations in chapter 4.2.

`loopy` is embedded into the Python language. Due to the dynamic nature of Python, the `loopy` IR is open for inspection and modification by the user. `loopy` embraces this fact by considering the IR a user interface. This allows users to extend it to their own needs by writing their own transformations, adding new code generation backends or even manipulating internal data structures. `loopy` further utilizes the embedding into the Python language by providing deep integration with popular Python projects such as `numpy` [114] and `pyopencl` [72]. Using this integration allows users to drive their computations entirely from Python.

`loopy` does not only provide an IR and a transformation library, but also code generators for a variety of target languages. These map the IR to actual code variants. Encapsulation of the code generators in backend objects (called *targets*) provides a clean separation between transformations and code generation. Currently loopy supports plain C, OpenCL, CUDA, ISPC and numba as targets. We will also study the translation to C code in more detail later on.

Loopy Data Model

`loopy` gathers an array computation kernel into a `LoopKernel` data structure. Such kernel typically only represents the inner, compute-intensive part of a larger program. This is reflected in the moderate amount of control flow that is expressable in the language. `loopy`'s notion of kernel granularity matches that of frameworks like OpenCL. A kernel object can be created with the `make_kernel` function from `loopy`. The following example implements multiplication of two vectors $\boldsymbol{b}, \boldsymbol{c} \in \mathbb{C}^n$, where multiplication of two values $x, y \in \mathbb{C}$ is defined as

$$\Re z = \Re x \Re y - \Im x \Im y \tag{3.1}$$
$$\Im z = \Re x \Im y + \Im x \Re y. \tag{3.2}$$

The complex vectors are represented as two-dimensional arrays. This example was advocated in [5] as a test case for memory layout changes in generative programming for HPC. We will use this kernel throughout this chapter to demonstrate `loopy`[2]:

```
knl = make_kernel("{ [i]: 0<=i<n }",
                  ["a[i, 0] = b[i, 0]*c[i, 0] - b[i, 1]*c[i, 1]",
                   "a[i, 1] = b[i, 0]*c[i, 1] + b[i, 1]*c[i, 0]"],
                  [GlobalArg("b", dtype=np.float64, shape=auto),
                   GlobalArg("c", dtype=np.float64, shape=auto),
                   GlobalArg("a", dtype=np.float64, shape=auto),
```

[2] All code examples assume that all symbols from `loopy` have been imported with: `from loopy import *`

```
                ValueArg("n", dtype=np.int32)],
            )
```

The three arguments given here do specify the loop domain, the computational statements and the array arguments that define the interface of the kernel with the rest of the code. Using the **print** function on the obtained kernel object gives us a summary of the kernel:

```
---------------------------------------------------------------------------
KERNEL: loopy_kernel
---------------------------------------------------------------------------
ARGUMENTS:
a: GlobalArg, type: np:dtype('float64'), shape: (n, 2), dim_tags: (N1:stride:2, N0:stride:1)
b: GlobalArg, type: np:dtype('float64'), shape: (n, 2), dim_tags: (N1:stride:2, N0:stride:1)
c: GlobalArg, type: np:dtype('float64'), shape: (n, 2), dim_tags: (N1:stride:2, N0:stride:1)
n: ValueArg, type: np:dtype('int32')
---------------------------------------------------------------------------
DOMAINS:
[n] -> { [i] : 0 <= i < n }
---------------------------------------------------------------------------
INAME IMPLEMENTATION TAGS:
i: None
---------------------------------------------------------------------------
STATEMENTS:
for i
  a[i, 0] = b[i, 0]*c[i, 0] + (-1)*b[i, 1]*c[i, 1]  {id=insn}
  a[i, 1] = b[i, 0]*c[i, 1] + b[i, 1]*c[i, 0]  {id=insn_0}
end i
---------------------------------------------------------------------------
```

We will now look carefully at the provided arguments. The domain was specified as a string that follows a syntax used in the integer set library ISL [115], which is also commonly used in compiler development. `loopy` uses Python bindings to that library in order to represent polyhedra and apply algorithms to these. The example domain here is the most simple one: A one-dimensional interval. Higher-dimensional cuboid domains can easily be expressed, e.g. as `"{ [i,j,k]: 0<=i,j<n and 0<=k<m }"`. For the sake of simplicity we restrict ourselves to cuboid domains here, as non-cuboid index domains do not appear in our work. In our example, the upper bound `n` is provided as an argument to the kernel. The name `i` is called an *iname* in `loopy` terminology, which is an axis of a polyhedral domain. In the summary, we see that inames may have tags attached that annotate how this axis should be implemented. Not having specified any such so far, it defaults to `None`, which results in a sequential **for** loop being generated. We will see more implementation tags as soon as we discuss kernel transformations.

The next argument is the list of statements that comprises the computation. Here, these are given as strings, but these strings are immediately parsed into instances of `Assignment` by `loopy`. It is possible to directly pass these `Assignment` instances which is advantageous when programmatically creating `loopy` kernels. The same holds for above domain objects. The `Assignment` object has several important data fields: For the above example the `assignee` and `expression` represent the left and right hand side of the assignment. These should be ASTs that represent the given arithmetic expression. Such a representation needs to have a full

symbolic understanding of the expression in order to allow manipulation by kernel transformations later on. The AST library used in `loopy` is called `pymbolic`. While it is specifically designed for the needs of `loopy`, it is also a standalone Python project in its own right. It provides an AST for basic arithmetic expressions and an implementation of the visitor pattern. This implementation is quite similar to the one in UFL and uses type-based function dispatch leveraging the dynamic nature of Python. It differs from UFL AST in two important ways though: `pymbolic` does not provide any terminal nodes specific to finite elements or to tensor algebra. Instead, `pymbolic` allows subscripted array accesses with arbitrary indices as leaf nodes, which does not exist in UFL as it does not have a notion of an array. Array information such as shapes and data types is not part of `pymbolic`. Unlike e.g. `sympy` [88], `pymbolic` preserves the input exactly and does not apply any automatic symbolic modification. Beyond the left and right hand side expressions, assignments have some more data fields:

- The `id` field uniquely identifies the assignment within the kernel (and defaults to `'insn_*'` above).
- The `within_inames` field describes where in the polyhedral domain the statement needs to be executed. Its default is derived from the inames that appear in subscripted expression of the left and right hand side expressions.
- For explicit dependencies between statements, the `depends_on` field can be given a set of ids that the statement does depend on.

Although the `Assignment` class is the most important realization of a computation statement, the possibility to add a C snippet as a statement exists. This should be seen as a sort of an escape hatch though, because those statements are opaque to many other parts of `loopy`, preventing many transformations.

The last argument of kernel creation describes the arguments of the kernel. Here we use the `GlobalArg` class to pass arrays from the calling scope into the kernel. `loopy` automatically determines whether these are read or written and will apply the `const` qualifier suitably. The array base type is specified using the `numpy` type system. For this work, only `np.float32` (single precision) and `np.float64` (double precision) are relevant. The `shape` field is used to define the size of the array as a tuple of logical axes - or with the marker `auto` that enables autodeduction from the array accesses in the kernel. Index tuples in the symbolic representation of the computation refer to these logical axes. Mapping these arbitrary many logical axes to a linear representation in memory can be defined in many ways. The `dim_tags` field, which we did not specify explicitly, defines this mapping allowing the following values:

- `'c'` and `'f'` indicate row major ordering (C-style) and column major ordering (Fortran-style) respectively.

- `'N<N>` defines an explicit nesting level for the axis with `'N0'` being the innermost, stride 1 axis.
- `'stride:<N>'` is using a fixed stride of N for a given axis. Internally, all tags are translated into this one at some point. Using this directly allows for the definition of arrays which are non-contiguous in memory by specifiying a stride which exceeds the size of the associated domain axis.
- `'vec'` indicates an axis to be SIMD vectorized. This forces the axis to be of stride 1 and have a length that matches the SIMD width. Larger array axes need to be split into SIMD chunks before this tag can be applied. Using this tag is of course only meaningful if the code generation target supports SIMD vectorization.
- `'sep'` splits the implementation of the axis into separate C-arrays. This allows `loopy` to deal with a multidimensional array, but have several lower dimensional arrays in the implementation.

An array memory layout is defined by a comma-separated list of per-axis tags. Combining the above tags in arbitrary ways allows us to define many, possibly very complex layouts. Defining computations indepedently of the memory layout and later changing it by transformation is one of the key features of `loopy`.

Transforming into Source Code

`loopy` encapsulates the code generation step into a backend object, which is called a *target*. The target object is a data field of the kernel object, which can be passed on construction or added later on:

```
knl = knl.copy(target=CTarget())
```

Here, we present code generation results using the target that generates C code. We will customize the target mechanism to our needs in section 3.3.3. `loopy` also supports CUDA, OpenCL, ISPC and numba as target languages. Some of these provide additional support for kernel execution from Python.

Code generation is triggered through the `generate_code` and `generate_body` functions, where the latter omits the function signature. For the above example, this yields:

```
for (int i = 0; i <= -1 + n; ++i)
{
  a[2 * i + 1] = b[2 * i] * c[2 * i + 1] + b[2 * i + 1] * c[2 * i];
  a[2 * i] = b[2 * i] * c[2 * i] + -1.0 * b[2 * i + 1] * c[2 * i + 1];
}
```

In order to achieve this result, `loopy` needs to provide three core facilities:

- A *scheduling* algorithm needs to provide a nesting of inames and an explicit order of statements. As statements within a `loopy` kernel are not strictly

ordered, but instead have dependency relations between each others, there might be some freedom for the scheduler in doing so. The scheduler implemented in `loopy` performs a rather simple backtracking algorithm in order to find a valid schedule.

- Generation of additional code snippets that are not part of the given statements. This includes a function signature, declaration of all temporary variables, a set of custom - maybe target-specific - preambles and the actual `for` loops for the given iname nesting. Also, `loopy` can express a limited amount of control flow at this level by attaching conditionals to statements, which result in `if` blocks being wrapped around statements.
- A mapping from statements to their C code implementation. This is a two-step procedure: At first the given ASTs need to be transformed to ASTs that structurally resemble C code. Rewriting of indexing expressions as linear subscripts using the array axis implementation tags is e.g. performed in this step. Also, function names are mapped to their target-specific implementation names by querying the target class with the given name and the inferred argument types. As an example, `loopy` kernels are defined using a generic `"min"` function, which is then mapped to `fmin` (C language double precision), `fminf` (C language single precision), `std::min` (C++) etc. The second step transforms the reduced AST into C code. Both of these steps are implemented as tree visitor algorithms. Target languages that are a superset of C extend these mappers by inheriting from them.

Applying Transformations

All `loopy` kernel transformations are implemented as Python functions, which take the kernel to be transformed as their first argument. They return a transformed copy of the kernel. Transformations can be classified into three categories: Those that apply transformations to the domain (e.g. loop tiling), those that annotate inames and arrays for the code generator (e.g. loop unrolling) and those that perform symbolic manipulation of the computation statements (e.g. prefetching).

We will use the above example kernel implementing the multiplication of two $\mathbb{C}^n$ vectors to provide some examples of loopy transformations. These are partly adopted from [71]. First, we will perform a splitting of the domain axis of size n into chunks of 4 and unroll the inner loop. This is accomplished by the following sequence of transformations, where we assume the argument `n` to be divisible by four in order to avoid the generation of a lengthy tail loop including conditionals:

```
knl = lp.split_iname(knl, "i", 4, outer_iname="j", inner_iname="k")
knl = lp.tag_inames(knl, [("j", "unr")])
knl = lp.assume(knl, "n mod 4 = 0")
```

The resulting kernel is changed in the following way:

```
...
---------------------------------------------------------------------------
DOMAINS:
[n] -> { [j, k] : k >= 0 and -4j <= k <= 3 and k < n - 4j }
---------------------------------------------------------------------------
INAME IMPLEMENTATION TAGS:
j: None
k: unr
---------------------------------------------------------------------------
STATEMENTS:
for k, j
    a[k + j*4, 0] = b[k + j*4, 0]*c[k + j*4, 0] + (-1)*b[k + j*4, 1]*c[k + j*4, 1]  {id=insn}
    a[k + j*4, 1] = b[k + j*4, 0]*c[k + j*4, 1] + b[k + j*4, 1]*c[k + j*4, 0]  {id=insn_0}
end k, j
---------------------------------------------------------------------------
```

Note how the tiling transformation has changed the domain axes and the indexing expressions in the computation, whereas the unrolling transformation merely marked an iname for unrolling within the code generation stage.

A commonly applied transformation is a memory layout change that transforms an Array of Structures (AoS) into an Structure of Arrays (SoA) layout. For the specific use case of arrays of complex numbers, AoS stores pairs of double precision values, whereas an SoA layout stores two arrays: One contains all the real parts, one all the imaginary parts. Switching between these is common enough that some C libraries such as UMFPack [30] provide interfaces for both layouts. In `loopy`, the memory layout may be changed by applying a transformation that retags the data axes of the arrays using the aforementioned data axis tags. With the following transformation, the output array uses a SoA layout, where the inputs assume AoS:

```
knl = lp.tag_array_axes(knl, "a", "N0,N1")
```

Again, the effect on the IR is only an altered tag. The generated C code does look completely different though:

```
for (int i = 0; i <= -1 + n; ++i)
{
  a[i + n] = b[2 * i] * c[2 * i + 1] + b[2 * i + 1] * c[2 * i];
  a[i] = b[2 * i] * c[2 * i] + -1.0 * b[2 * i + 1] * c[2 * i + 1];
}
```

Note that this transformation does not actually change the memory layout of the input data by applying a sequence of permuting instructions. The responsibility to pass an array with matching layout is with the calling scope.

Finally, `loopy` transformations may also be used to compose kernels in a procedural manner. The fundamental difference with these transformations is that they alter the mathematical problem that the kernel describes as opposed to just modifying its implementation. An example for such a transformation is `to_batched` which applies a given kernel to an array of inputs. The following code results in the same kernel as in our previous example, only that the original kernel only describes multiplication of complex numbers and the batching is done via a transformation:

```
knl = make_kernel("{ : }",
                  ["a[0] = b[0]*c[0] - b[1]*c[1]",
                   "a[1] = b[0]*c[1] + b[1]*c[0]"],
                  [GlobalArg("b", dtype=np.float64, shape=lp.auto),
                   GlobalArg("c", dtype=np.float64, shape=lp.auto),
                   GlobalArg("a", dtype=np.float64, shape=lp.auto)],
                  target=lp.CTarget())

knl = to_batched(knl, "n", ["a", "b", "c"])
```

The given parameters here denote the batch size `"n"` and the list of arguments which should vary across the batch. Dropping e.g. `"c"` from that list would result in a kernel that multiplies all elements of a vector $\boldsymbol{b} \in \mathbb{C}^n$ with a fixed value $c \in \mathbb{C}$.

`loopy` is by no means restricted to its builtin library of transformations. Custom transformations benefit greatly from the openness of the `loopy` IR: Data fields can be identified and modified by inspecting data structures in an interpreter session.

3.2.3 Vector Class Library

Writing explicitly SIMD-vectorized C++ code is an enormous technical challenge, as the C++ standard does not provide a high level interface to do so. Currently, there are three possible approaches: Write inline assembly, use intrinsic functions or use a library. Inline assembly is the worst choice from a performance portability point of view, as the code is tailored to the specific instruction set. Also, inline assembly requires expert knowledge to write and even if written perfectly, it introduces an optimization barrier for the compiler. Intrinsic functions mitigate this last disadvantage: Although the granularity level of intrinsic functions is close to the one of instructions, the compiler is able to optimize across intrinsic function calls. Intrinsic functions still suffer the problem of being architecture-dependent and compiler-specific, which poses a threat to performance portability. We will therefore advocate the use of a library for this purpose.

Choosing a SIMD Library

As intrinsic SIMD functions are proper, statically typed, C++ functions (as opposed to macros) they come with a full type system describing SIMD registers, e.g. `__m256d` describes an Advanced Vector Extensions 2 (AVX2) register interpreted as a SIMD vector of double precision values. With such a type system already being in place, the idea to extend this into a convenience library for SIMD programming using C++ operator overloading is quite obvious. In the last years, many such approaches have been published. We mention Agner Fog's vector class library [42], Vc [75], Boost.SIMD [39], Generic SIMD library [117], Sierra [77], MIPP [23], xsimd [100], Google's dimsum [50] UME::SIMD [58] and libsimdpp [57]. Additionally, compilers

have lately (e.g. GNU compiler version 6 or later) started adding compiler-specific extensions that allow definition of similarly rich SIMD types using a special attribute:

```
using AVXdoublevec = double __attribute__ ((vector_size (32)));
```

In order to decide which of the above libraries suits our purpose the best, we study classification criteria for these library approaches:

- *Portability.* Libraries should be usable from common compilers (g++, clang, icc) and across the most relevant cross-section of microarchitectures using one unified interface. This is a big issue with compiler vector extensions, as they lack standardization across compilers.
- *Templates.* Many libraries internally use templates which take the vector length as a template parameter. These templates are then specialized for the SIMD width available in the architecture. Other libaries use classes with fixed SIMD width mangled into the name instead. Some libraries take the templatization aspect even further to provide a truely portable approach: Code is written using vector lengths independent of the available SIMD width and the implementation splits these vectors into chunks of SIMD width. These approaches are better described as *programming models* rather than just convenience libraries.
- *Library functions.* All of the above mentioned libraries define basic types and overloaded operators for these types. However, they differ vastly in the amount of additional functionality they provide. This functionality covers vector construction, permutations, blends, reductions, conversions and mathematical functions. The missing availability of a lot of these is another big disadvantage of compiler vector extensions.
- *Stability and support.* For explicitly vectorized numerical codes, one usually wants to stay up-to-date with recent hardware developments. This boils down to a question of how well maintained the library is. In the case of the introduction of AVX-512, the differences in library adaptation ranged from “implemented before hardware was for sale” to “still missing as of this writing”.

For our code generation toolchain, we put emphasis on portability (across instruction sets and compilers), instruction set support and additional functionality in form of mathematical function implementations. Use of C++ templates to define SIMD-width independent vectors is not an important feature to us, as the code generation toolchain allows some flexibility in hardcoding the SIMD width in a performance-portable fashion. Infact, having non-template types might even be beneficial in code generation as it makes the generated code more simple, removing some potential sources of bugs. With these criteria in mind, we chose Agner Fog’s vector class library.

Fortunately, a satisfactory solution to the chaotic situation in SIMD programming is on the horizon, as Matthias Kretz, the developer of the Vc library, makes a move to introduce such functionality into the C++ standard [74]. Once such effort is successful, the advantages of standardization and of a wide user base will outweigh most of above considerations.

SIMD Vector Types

The SIMD vector types available in Agner Fog's vector class library can be subdivided into three categories: Floating point types, integer types and boolean or mask types. Floating point types can be described as the product of the available total vector widths 128, 256 or 512 bits and the bit width of the underlying type: 32 or 64 bits. The naming scheme in use is `Vec<n><ident>`, where `n` is a placeholder for the number of SIMD lanes (quotient of total bits and bits of underlying type) and `ident` is a short string describing the underlying type - here either `f` or `d`. Integer types work the same, except the underlying type is allowed to vary from (identifiers in brackets): 8 (`c`, not for 512 total bits), 16 (`s`, not for 512 total bits), 32 (`i`) and 64 bits (`q`). Additionally, all of these types are available as unsigned types by prepending `u` to the identifier. Mask types are available for all of the above types by appending `b` to the full class name, e.g. `Vec4db`. Data can be stored into these SIMD vectors through a variety of methods:

- *Construction from one scalar* results in a broadcast instruction.
- *Construction from scalars* using compile time constants is possible. Using runtime data effectively results in a gather operation.
- *Lower and upper half construction* uses instructions intended for compatibility with previous instruction sets to initialize e.g. an AVX2 register from two Streaming SIMD Extensions (SSE) registers.
- *Loading from an (un)aligned address* is implemented as a `load` or `load_a` method on the vector. This results in a load instruction.
- *Insertion of a scalar* is available as an `insert` method of the vector. The fact that the more convenient non-`const` overload of `operator[]` is not available is intentional to discourage users from writing such code, as it might often incur a *forward store stall* [41].
- *Assignment and copy construction* work with vectors of the same type. This is especially important when working with C-arrays of these vector types.
- *Gather operations* are possible through a `gather` (compile-time indices) and `lookup` (runtime indices) function using an integer vector with indices describing the offsets w.r.t. a given base pointer. Though available, vector gather instructions are not advisable on latest Intel architectures, as they have the highest latency and lowest throughput of all instructions in the instruction set [40].

Similarly, data can be retrieved from SIMD vectors through a variety of methods:

- *Extraction of a scalar* can either be done using an explicit `extract` method or using the `const` version of `operator[]`.
- *Retrieval of lower and upper half* is available using the methods `get_low` and `get_high` again using compatibility instructions.
- *Store to an (un)aligned address* is implemented as `store` or `store_a` in analogy to loads.
- *Scatter operations* are available in analogy with the above gather operations as a `scatter` function. The same performance warning applies.

SIMD vector types from the vector class library support all common arithmetic operations via C++ operator overloading. The same holds for logical operators, which yield the associated mask type. On top of that, the vector class library provides functions for the common tasks of permutation and blending, which are useful in data memory layout changes. It also gives direct control over FMA via the functions `mul_add`, `mul_sub` and `nmul_add`. As a unique feature compared to other SIMD libraries (to this extent), the vector class library provides its own set of high performance implementations of mathematical functions like `min` and `max`, `sqrt`, `pow`, integer rounding and truncation functions, exponential and logarithmic functions, trigonometric and hyperbolic functions as well as error functions. Having such implementations is necessary, as the C++ standard library does not provide templates, but specialized implementations for single and double precision. Efficient SIMD implementations of such functions are much more intricate than just executing a state-of-the-art algorithm on each lane, as these often contain too much branching to be feasible in SIMD calculations. Dedicated research on this matter exists, e.g. [106] and [84].

3.3 Code Generation Algorithms

With all the software pieces of the code generation puzzle being described, we will now look into the description of the actual code generation process. We will first look at how the input is preprocessed, then cover the main tree visiting algorithms and finally look at the generation of C++ code.

3.3.1 Defining and Preprocessing of UFL Expressions

The UFL input is provided by the user as a single Python object, describing the residual form of the PDE to solve. In this section, we go into detail about how we customize UFL compared to the general description in section 3.2.1 and what algorithms we apply to the input before starting the code generation process.

Residual Formulation

It is our intent to express the residual formulation from equation 2.16 directly in UFL. This approach differs from the FEniCS approach which distinguishes the linear and the nonlinear case. They do define bilinear and linear forms in the linear case, but use a residual formulation in the non-linear case - requiring users to switch to a different class for the ansatz function:

```
u = TrialFunction(FE)  # Linear case
u = Function(FE)       # Nonlinear case
```

The `Function` class used in FEniCS programs is not part of UFL, but imported from the `dolfin` Python package that extends UFL with `dolfin`-specific abstractions. As we want to use the same approach for both linear and nonlinear problems, we do not need this distinction and instead modify the `TrialFunction` node from UFL such that it is a coefficient function instead of a form argument. We distinguish trial functions and other coefficient functions by assigning a special reserved index to the internally stored count variable (which is used for distinguishing coefficient functions from each other). The modification is done in the module `dune.codegen.ufl.execution`, which provides the execution context for our UFL files.

Automatic Differentiation

As mentioned, AD greatly extends the power of our code generation approach. While in handwritten PDELab code maximum performance can only be achieved if the residual, its jacobian and potentially also the action of its jacobian on a vector are manually implemented, we can derive the latter two from the residual form using form manipulation algorithms from section 3.2.1. On a function space level, the jacobian matrix from equation 2.19 is obtained as the Gateaux differential Dr of the residual form. The resulting form is bilinear and still depends on u in the nonlinear case. UFL naturally handles Gateaux differentials through its `derivative` form transformation:

```
u = TrialFunction(FE)
jac = derivative(residual, u)
```

Having derived a jacobian form using AD, we can apply another symbolic transformation to obtain the linear form describing its action on a vector. Given the linearization point $x := \sum_j (\boldsymbol{w})_j \Phi_j$, this reads:

$$
\begin{aligned}
(\boldsymbol{J}(\boldsymbol{z})\boldsymbol{w})_i &= \sum_j (\boldsymbol{J}(\boldsymbol{z}))_{ij}(\boldsymbol{w})_j \\
&= \sum_j J(u; \Phi_j, \Psi_i)(\boldsymbol{w})_j \\
&= J(u; \sum_j (\boldsymbol{w})_j \Phi_j, \Psi_i) \\
&= J(u; x, \Psi_i) \\
&=: ja(u, x; \Psi_i)
\end{aligned}
$$

Assuming `jac` to be the jacobian form, this UFL transformation reads the following:

```
coeff = Coefficient(FE, index)
jacapply = action(jac, coeff)
```

`index` is used as a placeholder here for another reserved index, just like we implemented the trial function in the context of the residual formulation. This procedure is necessary to distinguish coefficient functions with special meaning in the code generation process.

Here, we have shown how the form compiler toolchain uses AD. The possibility to add AD code into the user input is untouched, e.g. for doing sensitivity analysis.

Simplifying Expressions

We have studied UFL's algorithms for preprocessing in section 3.2.1. In our toolchain we do always apply all of these algorithms to the input: Simplification of tensor expressions, pullback to the reference element, rewriting of geometric expressions and application of AD. We do this in order to reduce the number of UFL nodes that need to be handled by the main code generation visitor algorithm to a minimum. In the case of geometric quantities, it is worth having a closer look at cutoff types in use, meaning the UFL nodes that we keep in the input although a simplification might be available. We chose these such that they match the interface of DUNE's generic geometry interface, cutting off at the following types: `CellVolume`, `FacetArea`, `FacetJacobianDeterminant`, `FacetNormal`,`JacobianDeterminant` and `JacobianInverse`. It is worth taking a closer look at the effect of keeping `JacobianInverse`: The resulting expressions are as easy as the example AST from figure 3.12, whereas the simplified version would contain a symbolically inverted matrix expression. These inverted expression grow very fast with the geometric dimension and make the generated code very hard to read. Additionally, as UFL does not allow any procedural statements, the inverted matrix needs to be written out element by element, whereas inversion within C++ allows to define a temporary variable holding the matrix inverse. In the case of the jacobian inverse, this can be avoided by using the appropriate cutoff type. For general matrix inversion nodes, UFL does not currently allow keeping these in the

input, even if it was beneficial. We have targetted this issue in order to make the example of Maxwell's equation from section 5.4 work, where a 6×6 matrix needs to be inverted as part of the discretization (UFL cannot invert matrices larger than 4×4). We *monkey patch* (runtime source code manipulation) the inverse handler of the simplification algorithm to be non-operational. Furthermore the AD code needs to be patched with the following identity obtained from the product rule:

$$\frac{\partial A^{-1}}{\partial u} = -A^{-1}\frac{\partial A}{\partial u}A^{-1} \tag{3.3}$$

This identity is taken from [47], which also provides a good source for handling other matrix quantities in a similar way. The inverse node will instead be handled such that the matrix will be assembled into a dense matrix structure in the generated code and a library function for inversion from *dune-common* is called.

Note that in this section, we have only described simplifications that are applied on the UFL level. Some simplifications, like propagation of zeroes or Common Subexpression Elimination (CSE) will be applicable at a later stage.

3.3.2 Transforming UFL Expressions to Loopy Kernels

The main goal of this section is the transition of the AST of a UFL integral to a `loopy` kernel. That kernel will then provide the basis of any transformation-based performance reasoning. Finally, this kernel object can be translated into a piece of C++ code. The crucial point in transitioning towards a `loopy` kernel is identifying and adding those parts that are not part of the UFL AST.

Splitting Forms into Assembly Terms

UFL provides a symbolic description of the integrals in a multilinear form. The arguments of these forms are represented by a dedicated AST node. In implementation loops over test functions need to be added, as the generic problem formulation from equation 2.17 requires us to test against all basis functions of the test space. Within this loop, an update expression is added to the corresponding entry of the residual vector. When assembling jacobian matrices, additional loops for the trial functions need to be wrapped around the quadrature loop, and the results need to be accumulated into the corresponding matrix entry. This accumulation process in PDELab is controlled by the `accumulate` method of the local container that is passed to the assembly method. It takes the local function space object, a test function index and the update expression (or a pair of spaces and indices in the jacobian case) as arguments. We now seek to transition from a given UFL integral to this update expression, so that we can generate code for the call to `accumulate` directly. In order to do so, we have to programmatically separate those update expressions, that require a separate call to `accumulate`. This is necessary if the method needs to be called on a different container or if a different child of the local function space needs to be passed. This is the case, if (for facet integrals)

the restrictions on the form arguments do not match or the sub element index of a `MixedElement` node differs. We traverse the given integrand expression in a preprocessing step to identify all combinations of such modified terminals. Then, we run the main tree visiting algorithm from the next section once for each such term, looking to isolate the update expression needed for the call to `accumulate`. Due to `loopy`'s way of implicitly defining loop structure through *inames*, having the update expression depend on the assembly index will automatically result in the `accumulate` call to be correctly nested in the assembly loop nest.

Tree Visitor Algorithm

As seen in section 3.2.1, algorithms working on UFL ASTs are implemented using the visitor pattern. The algorithm that translates UFL integrals into `loopy` kernels is no exception. However, providing one such visitor class would lead into a lot of branching based on form compiler options within the handlers. We therefore construct the visitor class as a composition of mixins, which are selected by form compiler options. Currently, four types of mixin configuration options are considered: `geometry_mixins` (providing specialized geometries for subclasses of grids e.g. structured equidistant grids), `quadrature_mixins` (allowing tensor product quadrature formulae in section 4.4), `basis_mixins` (again allowing tensor product structure of basis functions later on) and `accumulation_mixins` (describing the splitting described above, which also varies in sum factorization). All of these mixin parameters default to a generic implementation that works for any DUNE grid and uses generic high-level constructs from PDELab. We will discuss specialized mixins where appropriate.

We will now study how UFL node types are handled by the UFL-to-`loopy` visitor looking at groups of nodes that are treated similarly. The visitor would be better described as a converter from UFL to `pymoblic`, as the return value of the traversal process is a `pymbolic` expression, that we can then plug into the `loopy` statement that realizes the `accumulate` call. The visitor is not only translating expressions to `pymbolic`, but handlers of the visitor may also have side effects, which add more statements to the `loopy` kernel to be created (such as evaluation of basis functions).

In section 3.2.1 we introduced terminal modifiers as a special node type that can only occur as a direct parent node of a terminal or another terminal modifier. Examples are the restriction node types or the `Grad` node. These nodes cannot be directly translated into the target language, as they have no `pymbolic` analogon. Instead, the presence of a terminal modifier alters the way that the child terminal node is handled, e.g. a gradient node attached to a test function should result in evaluation of the gradient of the test function, not differentiation of the evaluation of the test function. Therefore, handlers for terminal modifiers modify the state of the visitor marking their presence and go into recursion.

We will now have a look at how the code for indices and indexed expressions is generated. First, we observe that integrands of UFL integrals are always scalar i.e.

no free indices. The only way for indices to be introduced within a UFL AST is through the `IndexSum` node. When handling such nodes, we do unrolling of the reduction index space at code generation time, meaning we visit the reduction expressions once for each possible index, turning the reduction into a regular sum. TSFC, the form compiler used by the firedrake project, behaves similarly in this regard [55]. This is a drastic decision, so we give a few more reasons about why we consider it a good one: Reductions expressed in UFL typically only appear with index ranges of the order of the geometric dimension or of the number of components in a PDE system. In both cases, these are rather small, such that the compiler would typically unroll the corresponding loops anyway. Furthermore, treatment of the resulting expressions is much easier when performing CSE in order to find a code variant that uses the minimum number of floating point operations. Also, these unrolled expressions allow additional optimizations that propagate zeroes and thereby reduce the complexity of expressions. We will talk about this zero propagation in more detail later this section. When doing the code generation time reduction unrolling, each `IndexSum` node introduces a replacement of its symbolic index with the current unrolled integer index. As UFL does not reuse index names across an expression, we can simply store the index replacement rule in a dictionary on the visitor. In this unrolled setting, the handling of `ComponentTensor` nodes is straight-forward, as it only adds index replacement rules the index dictionary. In the non-unrolled setting however, handling component tensors would require us to introduce a temporary variable array for each such tensor.

The handler for the `Indexed` node looks up the given multi index (its second operand) in the transitive closure of the replacement dictionary and pushes the result onto a stack of indices stored with the visitor. It then continues to visit its first operand recursively. Afterwards, if the top stack element is a multi index, it wraps the returned `pymbolic` expression into a `Subscript` node indexed with that multi index. This procedure is necessary to enable other handlers to treat indices implicitly instead of explicitly adding a subscripted expression. For an example of where this happens, we return to the Taylor-Hood element studied before:

```
TH = FiniteElement("CG", cell, 1) * VectorElement("CG", cell, 2)
test = TestFunction(TH)
p, u = test.split()
```

Here, the children of the finite element tree, such as `p`, are accessed by indexing the test function object `test`. However, when handling the test function object, we need to generate code for leaves of the finite element tree, so we implicity handle the index that selects the component of the PDE system. Another such example is the handling of the `ListTensor` node, as in the unrolled setting, it can just implicitly handle the index by selecting the correct tensor entry at code generation time. Handlers that implicitly treat indices push `None` onto the index stack in order to prevent the `Indexed` handler from erroneously adding a subscript from the stack.

Many non-leaf node types have a `pymbolic` counterpart, that can be translated verbatim. This applies to all arithmetic operations, mathematical functions and logical operators. One important optimization that is applied to these nodes is code generation-time evaluation, which is quite relevant in two cases: Firstly, multiplication with zero should yield zero and not even generate code for any other factors. Secondly, if some operands of mathematical operators and functions are literals, code generation-time evaluation (a more general version of *constant folding*) can be applied. With all operands being literals, it is even possible to completely omit the operator and write the result directly into the generated code. With the above unrolling of reductions, this situation appears quite common, e.g. think of a gravity vector being defined as $(0, 0, -9.81)^T$: It will never be assembled into a vector in generated code as the x and y component will be removed during the code generation process, saving some floating point operations. When we look into code generation for sum factorization on axiparallel grids in section 4.4, this simplification will be the foundation of a very huge gain: As the interior facet integrals are implemented separately for each facet orientation, the normal vector $\boldsymbol{n}$ will be a (negative) unit vector, resulting in elimination of a lot of work, e.g. $(\nabla u, \boldsymbol{n}) = \pm\partial_i u$.

We will now turn to handlers for finite element functions. The handler for the `Argument` node takes into account the splitting into accumulation terms mentioned above. For each accumulation term, the visiting process is started once and if the visited argument node's terminal modifiers do not match the current accumulation term, `0` is returned. With the described propagation of zeroes, this will lead to expressions collapsing in such a way that the visiting process returns a minimal representation of the accumulation update term in `pymbolic`, which we can directly plug into a `loopy` statement that realizes the `accumulate` call. If the given `Argument` node's terminal modifiers do match the current accumulation term, we need to return the properly indexed temporary variable that contains the evaluations of basis functions (or respectively their gradients). Also, we need to make sure to trigger the evaluation of these basis functions, which will be realized in separate loopy statements. `Coefficient` nodes are handled in two ways: If they have a reserved index (finite element solution or linearization point in matrix-free computations), we implement the evaluation of its basis and the linear combination with given coefficients. Other coefficient functions are expected to be grid functions in the PDELab sense, which provide a high-level interface for evaluation. All finite element function related handlers are grouped into a mixin class, as the code generation process for sum factorization requires entirely different handlers.

Another category of handlers that is grouped into a mixin are quadrature-related ones. Again, the intent is to allow the tensor product structure exploiting variant to behave differently. It provides the quadrature weight and position in local coordinates for the geometric quantity handlers to operate on. The returned quantites are subscripted with the quadrature iname(s), again implicitly defining the nesting of statements inside the quadrature loop.

Mixin name	J^{-1}	$\lvert \det J \rvert, \lvert T \rvert, \lvert F \rvert$	$\boldsymbol{n}$
`generic`	position-dependent full pattern	position-dependent	position-dependent
`axiparallel`	constant on cell diagonal pattern	constant on cell	constant on cell
`equidistant`	constant on grid diagonal pattern	constant on grid	constant on grid

Figure 3.13: Geometry mixins available to exploit geometric structure in certain grids. The generic mixin works for all grids conforming with the *dune-grid* interface. The geometry mixin is selected through a form compiler option `geometry_mixins`. Sum factorization-specific mixins are not shown.

The most important group of handlers controlled by a mixin are the ones that calculate properties of the geometry mapping μ_T. Also, it has the richest variety of implementations available. Table 3.13 shows those, omitting any mixins introduces in section 4.4 for the sum factorization case. Choosing the correct mixin for structured grids can greatly reduce the number of required floating point operations for several reasons: In structured grids, many quantities do not need to be evaluated at every quadrature point, as they do not vary across these. In equidistant grids, this precomputation can even be moved into the class constructor. Also, exploiting the diagonal structure of the jacobian of the geometry mapping μ_T and its inverse saves some operations. This optimization can be nicely implemented in the loop unrolling setting, as the handler implicitly handle the indices at code generation time and return `0` if an off-diagonal entry is requested.

Memoization Patterns in Code Generation

Memoization is a computing technique that reuses cached results of function calls upon reevaluation, quite similar to dynamic programming. While memoization is an optimization technique in the case of expensive function calls, it can as well be used to give guarantees about a function being called exactly once with a given set of parameters. The latter is what we are primarily interested in in the field of code generation: Nodes that appear multiple times in ASTs should often map to a single snippet of C++ code in order to ensure the generation of valid C++ and to avoid unnecessary operations. As a large portion of the code generator code base consists of memoized functions, we will describe the software design of the memoization infrastructure in a bit more detail.

The following requirements on the memoization pattern arise in our code generation toolchain:

- *Partial memoization*: The caching decision should not necessarily depend on all function arguments.

- *Fine-grained cache lifetime control*: Some cache results need to be reset for each integration kernel, some for each UFL form, some never. We need the possibility to filter cache entries when retrieving and deleting them.
- *Insertion order preservation for caches*: This is important if the memoization technique is used to ensure validity of C++ code. Otherwise, objects might be instantiated in the wrong order.
- *Simplicity*: The development workflow in Python should be simple.

We meet these requirements by implementing our own memoization infrastructure that provides Python decorators [43]. These decorators are used on most functions in the code generation process. Here is a simplified example that shows how a nested finite element tree is traversed bottom up to with memoization being applied to all subtrees:

```
@cached
def define_gfs(FE):
    if isinstance(FE, MixedElement):
        for e in FE.sub_elements():
            define_gfs(e)
        return "Defining mixed element {}".format(FE)
    else:
        return "Defining finite element {}".format(FE)

TH = FiniteElement("CG", cell, 1) * VectorElement("CG", cell, 2)
define_gfs(TH)
```

The `cached` decorator has insertion order preservation enabled, such that the output upon cache retrieval correctly traverses the finite element tree bottom up:

```
'Defining finite element <CG1 on a triangle>'
'Defining finite element <CG2 on a triangle>'
'Defining mixed element <vector element of <CG2 ...>>'
'Defining mixed element <Mixed element:'
    '(<CG1 ...>, <vector element of <CG2 ...>>)>'
```

For partial memoization, a lambda can be applied to map the function arguments to a memoization cache key. Life time control is implemented by instantiating the memoization decorators with a list of tags. Retrieval and deletion of cached function evaluations are done with `delete_cache_items` and `retrieve_cache_items` functions. These can filter cache items based on the tags that the decorator was built with.

3.3.3 Generating C++ Code

So far, we described the translation of PDE models between the IRs of UFL and `loopy`. We will now turn to describing how the latter IR is translated into C++ code.

A Loopy Target for DUNE

We have studied the code generation abstraction of loopy - a *target* - in section 3.2.2. The built-in `loopy` target which is closest to our needs is the plain C target. We inherit from it to enhance it for our additional needs which stem from the following requirements:

- *C++.* Generating C++ code instead of C code does not require much adaptation, as we are sticking with the C subset of C++ wherever possible and only generate C++ code, when strictly necessary. This includes using plain C arrays for temporary variables of a kernel. An example of what needs to be adopted is registration of math functions from the C++ standard library to have these take precedence over the plain C functions.
- *VCL support.* The C target is extended to support the vector class library from section 3.2.3. The only necessary step to do so is to register the additional vector types into `loopy`'s type system. Having done so, vectorized statements are expressed by tagging domain array axis with the `vec` tag. Also, functions operating on these vector types need to be registered.
- *FMA.* As of this writing, `loopy` does not treat FMA as a separate tree node. We introduce this node and its handlers in order to ensure FMA instructions are issued in code using the vector class library. The newly introduced node is directly used in tensor contraction code covered in section 4.4. In order to also use FMA nodes in expression that were translated from UFL, a `loopy` transformation is called that matches FMA patterns.
- *Compatibility with framework data structures.* Some memory layouts, e.g. the layout of global DOF data structures, are fixed by PDELab and the generated code needs to respect this memory layout. In contrast to C arrays, in C++ this memory layout is defined by using nested containers. `loopy`'s way of calculating indices from multi-indices and the corresponding strides does not work here, as the only correct implementation of data accesses into a nested container is through multiple square bracket operators. We subclass the `loopy` classes for arrays such that we can switch the indexing behaviour between flat indexing and nested `[]` operators. DUNE's data structure for a dense matrix from $\mathbb{R}^{n\times m}$ is an example of such data structure.
- *Operation-counting floating point type.* In the benchmark setup description in section 5.1, we will introduce a floating point type, that counts the conducted floating point operations. Our target can switch between Plain Old Data Structures (PODs) and these.

We implement all of these changes into a new class `DuneTarget`, although it would also make sense to further split this into an inheritance hierarchy or mixins for future customization.

The Local Operator Class

The target mechanism of `loopy` produces the function bodies of the assembly methods `alpha_*`, `jacobian_*`, etc. The rest of the local operator class and the header file it is contained in is generated using a more simple approach using code snippets and the described memoization infrastructure. Snippets gathered in this way include the following: Include directives, class template parameters, base classes, constructor arguments, initializer list entries, class member variables and methods. A notable exception is the constructor body: It is also generated from a `loopy` kernel, allowing precomputation of complex quantities on the operator level. Some template parameters and constructor parameters are generated regardless of the visiting process in order to reduce the variation of how local operator objects are instantiated. This minimal interface looks as follows:

```
template<typename TRIAL_GFS,
         typename TEST_GFS>
class LocalOperator
{

  public:
  LocalOperator(const TRIAL_GFS& trial_gfs,
                const TEST_GFS& test_gfs,
                const Dune::ParameterTree& iniParams);
};
```

The given parameters are the trial and test grid function spaces and the configuration tree, whose keys can be read by the operator class. The visiting process may append additional arguments to the template parameter list and additional constructor arguments. E.g. if the UFL input contains a coefficient function, the user needs to pass a grid function to the constructor. The ordering of arguments is made deterministic by adding tags that serve as sorting criterion.

Generating Simulation Driver Code

Although we deliberately chose to restrict the code generation process to the integration kernels gathered in a `LocalOperator`, generating code for the entire simulation workflow can sometimes be beneficial. This is the case in automated testing, as it drastically reduces the amount of code duplication involved in a test suite. It is also beneficial when it comes to rapid prototyping, as it removes the burden of coding up a simulation driver when quickly trying out a new discretization scheme. We therefore try to provide automatically generated simulation drivers as well. Of course, this comes at a cost: Several choices of building blocks from

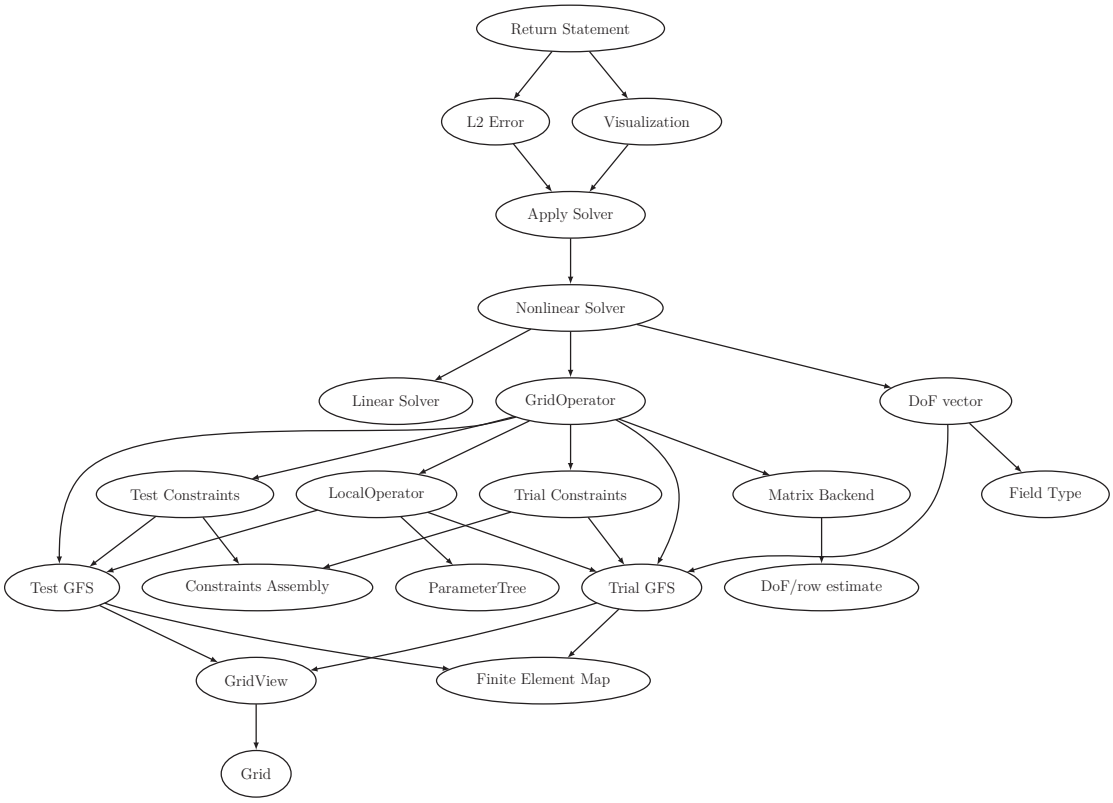


Figure 3.14: Flowchart of a PDELab simulation driver: Arrows define dependencies between simulation components. Traversing this DAG bottom-up and generating C++ snippets on the fly yields the generated code for the driver function.

DUNE/PDELab are fixed at code generation time with no means of changing them. Such blocks are: the grid implementation (always using a structured grid - although implemented through an unstructured grid manager in the simplicial case) and the linear solver, which is always chosen to be UMFPack [30], a direct solver. Many other building blocks can be correctly selected at code generation time without additional knowledge e.g. a Newton solver is selected if and only if the UFL input is a nonlinear form. Generation of such driver code is implemented as a large collection of functions that implement small C++ snippets and make heavy use of the memoization infrastructure from section 3.3.2 to ensure the generation of valid C++ code. The code generation process is triggered by a function call that generates the return statement of the `main` function. With each function calling all other functions which it depends on, a correct topological sorting of code snippets can be achieved if the memoization infrastructure is preserving the order of cache entries. Figure 3.14 shows the dependency graph of the framework components involved.

We do allow a few exceptions from the above rule of not providing customizations to the driver generation process with the intent of lowering the bar for extensive automated system testing (section 3.4.2 will extend on this). These are mainly

concerned with information which is not part of the UFL form, but is expressable using the symbolic language of UFL:

- If an expression `interpolate_expression` exists in the UFL file, it is interpolated into the DOF vector before solvers are applied. This way, Dirichlet boundary conditions and initial conditions of instationary problems can be implemented.
- An expression `is_dirichlet` from the UFL file is interpreted as the condition when to apply Dirichlet constraints (using `1` for constrained and `0` for unconstrained).
- If an expression `exact_solution` exists, it is interpolated into a different vector, which is used to calculate the error norm $\|u_h - \mathcal{I}_h u\|_{L_2(\Omega)}$, with $\mathcal{I}_h$ being the operator interpolating the solution onto the finite element space. This is important to rule out purely numeric errors in automated testing.

When solving a PDE system, all of the above accept a tuple of UFL expressions, a vector-valued UFL expressions or a mixture thereof. In all cases, the provided data is flattened into a tuple who entries correspond to the leaf nodes of the finite element tree.

3.4 Integration into the DUNE Framework

We will now turn to describing how the Python-based toolchain can be embedded into the DUNE user workflow. Alternatively, one could seek to provide Python bindings for DUNE, as pioneered in other work [32]. As explained in section 3.1, we will instead trigger the code generation process from the DUNE build system.

3.4.1 The Module dune-codegen

The modular structure of the DUNE framework makes it a natural choice to provide the code generation toolchain in the form a new DUNE module. The module dune-codegen is available under a BSD license from the DUNE Gitlab server [62], installation instructions can be found in appendix B.1. The dune-codegen module provides the following components:

- CMake code needed for integration of the code generation process into the DUNE user workflow. This will be covered in the rest of this section.
- C++ code used from code generation: Sometimes it is easier to generate a call to a C++ function instead of actually generating the function body itself. We provide such code in C++ header files, which are included by generated header files.
- The Python package `dune.codegen` that implements the code generation process, as described in section 3.3.

- Upstream Python package dependencies bundled as `git` submodules. Having the source code of Python packages available is important for a seamless development workflow and to be able to apply patches to upstream dependencies.

3.4.2 CMake Integration

DUNE uses a CMake build system [85] to control the important tasks of configuring, building, testing and installing software. CMake provides functionality well-suited to integrate code generators into the build process. In this section, we will describe the user interface of the integration into DUNE, omitting any CMake-internal technical details.

Integrating Python Support into the DUNE CMake Build System

In order to automatically trigger the code generation process as part of the build process, CMake needs to be able to run Python code in a well-defined environment. Additionally, the Python sources provided by dune-codegen need to be installed into this environment. We have implemented a build system extension in dune-common which allows CMake to set up its own virtual environment[3] and install Python software into it at CMake runtime. This virtual environment is shared across all DUNE modules in the same build allowing the interplay of Python packages provided by multiple Dune modules. The advantage of such an automated approach is that the requirements on the user side are kept to a minimum: A Python interpreter needs to be available and the automatic environment setup needs to be enabled according to the instructions in appendix B.1.

Triggering Code Generation from CMake

A simulation executable whose local assembly kernels should be generated is added using the following CMake function:

```
dune_add_generated_executable(
    UFLFILE uflfile
    INIFILE inifile
    TARGET target
   [SOURCE source]
)
```

The target parameter is given the name of the CMake target to be created, the source parameter is the C++ source, which defaults to a simulation driver auto-generated as described in section 3.3.3. The UFL file is where the actual code in the UFL DSL is supplied. These files are also used in the FEniCS context and use

[3] This uses one of the Python packages `virtualenv` and `venv`.

the extension `ufl`. The content is pure Python, where you can assume that all symbols from `ufl` have been imported.

The file given to the ini file parameter is a standard DUNE configuration file. These files consist of key/value pairs separated by an equal sign and arbitrarily nested sections specified through square bracket section headers or dots in keys. The above CMake macro only reads the `[formcompiler]` section of the inifile, allowing the use of the same configuration file for the simulation run. A comprehensive list of valid keys in the formcompiler section can be obtained using instructions from appendix B.1. Right now, we will only mention those necessary to drive the code generation process. The `operators` key expects a comma-separated list of identifiers, where each identifier refers to one header file with one `LocalOperator` to be generated. A subsection of the same name as this identifier can be used to provide form-specific options. The following example generates the two operators necessary for a simple instationary problem:

```
[formcompiler]
operators = mass, residual
[formcompiler.mass]
form = mass
filename = mass_operator.hh
classname = MassOperator
[formcompiler.residual]
form = residual
filename = residual_operator.hh
classname = ResidualOperator
```

The `form` parameter specifies the name of the Python object that describes the UFL form to generate code for. It defaults to the operator identifier. The example also shows how names of generated classes and headers can be controlled. Including the generated headers in a C++ source code and building the executable using `make` will first invoke the source code generator and then the C++ compiler. Furthermore, changes in Python code will retrigger the code generation process, whereas changes in C++ code will not.

Automated Testing of Code Generation

Software testing is one of the central challenges in the development of research software. With general purpose PDE software, system testing that covers the variability of the framework is essential. We have described the issue before in [65] and provided a solution that allows description of a variability model through a DSL that is embedded into DUNE configuration files. This DSL is available in the module dune-testtools [64]. In the context of code generation, system testing becomes even more important, as bugs introduced in this additional layer are hard to spot if they do not break compilation, but just produce wrong simulation results. We therefore extend the approach of dune-testtools by not only allowing static (compile-time)

and dynamic (run-time) variation points, but also generation-time variation, which is expressed by applying the dune-testtools DSL to the **[formcompiler]** section in the configuration file.

A CMake function to add such system tests is also provided:

```
dune_add_formcompiler_system_test(
    UFLFILE uflfile
    INIFILE inifile
    BASENAME basename
   [SCRIPT script]
   [SOURCE source]
)
```

Compared to above, the base name replaces the target name parameter, which static variants use to append suffixes defined in the dune-testtools DSL. The script parameter can be given any script that is wrapped around the simulation run during test execution and that may determine test success or failure. It defaults to a no-op wrapper, but dune-testtools also provides more involved wrappers that do comparison of output files or calculation of convergence rates. In order to be able to introduce variation points that require variation of the UFL input e.g. polynomial degrees, a special section **[formcompiler.ufl_variants]** is implemented, that injects its key value pairs into the execution context of the UFL file.

CHAPTER 4

SIMD vectorization of DG methods

Making effective use of the SIMD capabilities of a modern CPU is a necessary criterion for an HPC implementation of finite element assembly to be competitive. In this chapter, we will study the vectorization problem for the DG assembly problem with sum factorization from section 2.6. We will start off with a discussion of the challenges unique to the finite element problem in section 4.1. In section 4.2 we will present a new class of vectorization strategies based on identifying multiple, structurally similar workloads within the assembly problem on one cell or facet. Implementation of these contains some performance-critical SIMD operations, which we will study at the assembly level in section 4.3. Having defined these new strategies, we will discuss in section 4.4 how they are integrated into the code generation procedure from chapter 3. Section 4.4.3 will put special focus on the use of autotuning for identifying optimal vectorization strategies in the code generation process.

4.1 Challenges in Vectorizing PDE Codes

To understand the challenges in SIMD vectorization of PDE code, we reiterate on the nested loops typically present in a general potentially nonlinear and time-dependent PDE problem already mentioned in section 2.5:

- Solution of multiple PDEs (UQ, optimization)
- Timesteps of an instationary problem
- Stages of a time stepping scheme
- Iterations of a Newton solver for nonlinear problems
- Iterations of an iterative linear solver

- Iteration over grid cells
- Quadrature points for cell-local integration
- Components of a coupled system of PDEs
- Local degrees of freedom that are updated

Despite many of these loops being perfectly nested, compiler-based autovectorization is typically not capable of fully saturating the SIMD capabilities of modern CPUs. Two prominent reasons for this are unfavorable loop bounds and the amount of complex control flow involved. The loop bounds of the innermost loops typically depend on the number of local basis functions, the number of quadrature points, the geometric dimension of the domain or the number of components in a system of PDEs. These quantities are typically not divisible by the SIMD width and the loops are too short to amortize the cost of a tail loop. Typically, the granularity of a cell integral evaluation (and the amount of control flow involved) is too big for a compiler-based technique, such as whole function vectorization [59].

The effect of unfavorable loop bounds is even more pronounced when applying sum factorization, as the loop bounds stem from the 1D basis functions and quadrature rules. Although the sum factorization algorithm consists only of tensor contractions, which can be recast into the well-studied form of a General Matrix Multiply (GEMM), it is hard to leverage existing implementations due to the very severe loop bound constraints. Level 3 Basic Linear Algebra Subprograms (BLAS) do not only suffer from the same divisibility problem, but also from the fact that the workload of a single GEMM is too small to amortize the function call overhead. To target such problems a *batched* BLAS Application Programming Interface (API) has been introduced, that allows amortizing the function call overhead more easily by executing a large batch of GEMMs at once. Unfortunately, it is geared towards very large batch sizes of GEMMs, which do not fit the requirements of the sum factorization algorithm. Quite recently, Intel has introduced a *compact* BLAS API [67] into the Intel Math Kernel Library (MKL) that defines GEMM operations in batches of SIMD width. Such an API will be very useful for the explicit vectorization strategies described in this work once it is sufficiently mature. As of this writing, the support for memory layouts and matrix transpositions is not sufficient for our algorithms.

Another approach developed at Intel is the use of JIT-compiled GEMM kernels for small matrices through a library called *libxsmm* [53]. However, the same technical obstacles as in the compact BLAS case apply: The library currently does not implement the necessary memory layout and transposition variants of GEMM. Additionally, it does not batch GEMM operations and therefore - although geared towards small matrices - suffers from divisibility constraints for very small matrices such as 3×3.

Several approaches towards explicit SIMD vectorization for PDE codes have been developed in the literature. We review these in order of appearance of the targeted

loops in the above nesting order. Vectorizing multiple realizations of a simulation in the UQ setting has been studied e.g. in [98] under the name *embedded ensemble propagation*. The goal of this strategy is to manipulate global data structures such that the PDE is solved with different data on each SIMD lane. For the easier case of restricting the variation in data to the right hand side of the finite element problem, a similar procedure has also been studied for the DUNE framework [18]. In such approaches, *ensemble divergence* is a fundamental, largely unsolved, problem: Whenever the control flow of the computations on SIMD lanes diverges, expensive restarting or rearranging of ensembles needs to be performed. Such scenarios arise in many simulation scenarios like: Data-dependent iteration numbers in iterative solution techniques, adaptively refined grids or discrete events like contact in Fluid-Structure Interaction (FSI) problems.

A popular choice for SIMD vectorization of PDE code is the grid iteration loop. Here, several cell integrals are calculated in parallel on the SIMD lanes. The approach has the big advantage of being always applicable, as there are always enough cells in a grid to amortize a potential tail loop. Also, it is applicable for any SIMD width. It bears however some disadvantages as well, as the memory footprint of the integration kernel is increased by a factor of the SIMD width. This becomes even more of a problem when batching facet integrals due to their reduced dimensionality. Additionally, data structures for an integration kernel need to be setup through a (partial) SoA to AoS transformation, which is similar to the one we will show in section 4.3, even when an individual kernel would be able to operate directly on global data structures. Such cross-element vectorization is the most common choice in numerical software packages. deal.ii tries to mitigate the memory footprint effects by using a special Hermite basis with reduced support [76], firedrake employs cross-element vectorization on structured grids fully automated from a code generation toolchain [109]. A very broad overview of applicable vectorization methods and how autotuning is used to select an optimal code variant for the finite element code BLAST is provided in [1].

4.2 SIMD Vectorization Strategies

In the following we will develop vectorization strategies for the sum factorized assembly algorithm 2.3. These strategies are based on an original idea from [91] and were further extended in [63]. Our description here is in part taken from the latter. The novel idea of these vectorization strategies is to perform explicit vectorization within one cell integral regardless of the polynomial degree by performing parallel evaluation of several sum factorized quantities. We will classify our approaches into two categories: Loop-fusion based approaches and loop-splitting based ones. Loop fusion based approaches studied in section 4.2.1 typically require a drastical change in memory layout to work, where loop splitting based ones from section 4.2.2 only work optimally if the mathematical problem leads to loop bounds satisfying

suitable divisibility constraints. Section 4.2.3 will then aim at extending the scope of applicability of these strategies by introducing hybrid strategies as well.

4.2.1 Loop Fusion Based Strategies

An integration kernel on a cell or facet typically computes several quantities that need to be evaluated by a sum factorized algorithm. Many of these sum factorization algorithms exhibit great structural similarities. We explain the idea using the example of the volume integral of the residual evaluation algorithm 2.3 for a second order elliptic PDE in 3D with a SIMD width of 256 bits. We do this restriction to illustrate our core ideas and later discuss generalizations to other models, architectures, space dimensions and jacobian assembly. The core idea is to use the necessary four tensor quantities of step one in algorithm 2.3 for vectorization: The finite element function $\hat{U}$, evaluated at all quadrature points $\xi_{i_0 i_1 i_2}$, as well as the three components of its gradient $\partial_k \hat{U}$. We recall the main tensor product formulae from section 2.6 for these quantities (in 3D):

$$\partial_0 \hat{U} = \left(D^{(0)} \otimes A^{(1)} \otimes A^{(2)}\right) X \tag{4.1}$$

$$\partial_1 \hat{U} = \left(A^{(0)} \otimes D^{(1)} \otimes A^{(2)}\right) X \tag{4.2}$$

$$\partial_2 \hat{U} = \left(A^{(0)} \otimes A^{(1)} \otimes D^{(2)}\right) X \tag{4.3}$$

$$\hat{U} = \left(A^{(0)} \otimes A^{(1)} \otimes A^{(2)}\right) X \tag{4.4}$$

As the tensor bounds m_j and n_j match in all of these computation, so do the loop bounds in the resulting sum factorization kernel implementation. Therefore, the loops in the implementation can be fused to achieve an implementation suitable for SIMD vectorization. In such an implementation, each of the equations 4.1 - 4.4 would be carried out in one SIMD lane. We will now introduce mathematical notation to reason about such a fused kernel as a tensor calculation. To this end, we define two operations commonly used in tensor algebra [79]:

- The *vec* operator maps a d-way tensor to a vector by flattening. This operation imposes an order on the tensor axes. This is completely analoguous to selecting strides in multi-dimensional array computations. We will use this operator to refer to the representation of a tensor in memory.
- Given d-way tensors $A_0, \ldots, A_n$ with identical bounds, we define $A_0 | \ldots | A_n$ as the $(d+1)$-way tensor constructed by stacking the tensors A_i on top of each other. We assume that the order of axes of the input tensors is preserved in the stacked tensor and that the new axis generated by stacking is the fastest varying (or has stride 1).

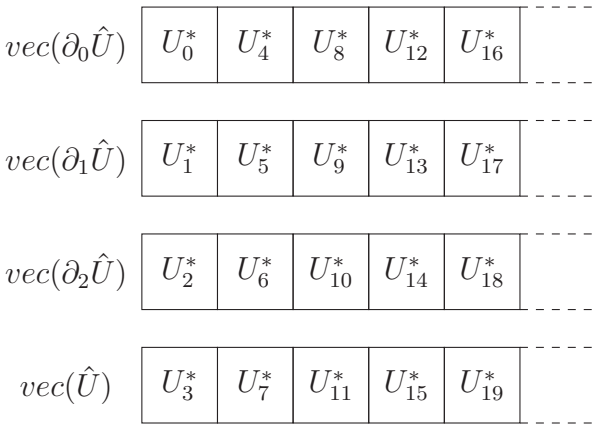


Figure 4.1: The memory layout of the tensor $U^* := vec(\partial_0\hat{U}|\partial_1\hat{U}|\partial_2\hat{U}|\hat{U})$ is an AoS obtained from the four original arrays $vec(\partial_0\hat{U})$, $vec(\partial_1\hat{U})$, $vec(\partial_2\hat{U})$ and $vec(\hat{U})$ by interleaving.

Using this notation, equations 4.1 to 4.4 can be combined into one equation:

$$\begin{aligned}\partial_0\hat{U}|\partial_1\hat{U}|\partial_2\hat{U}|\hat{U} = \big(&D^{(0)}|A^{(0)}|A^{(0)}|A^{(0)}\\ &\otimes A^{(1)}|D^{(1)}|A^{(1)}|A^{(1)}\\ &\otimes A^{(2)}|A^{(2)}|D^{(2)}|A^{(2)}\big)\, X|X|X|X\end{aligned} \tag{4.5}$$

We will now discuss the memory layout implications of implementing equation 4.5. The memory layout of the input tensor X is prescribed by the underlying discretization framework. Accesses to the stacked tensor $X|X|X|X$ can be implemented easily by accessing an element of X and broadcasting it into a SIMD register. Our code generator will do this, whenever it finds a stacking of identical matrices. The layout of the stacked basis evaluation matrices is given by interleaving the individual basis evaluation matrices, such that the stacked axis has stride 1. We assemble these stacked basis evaluation matrices in memory. This is a trade off decision between the increased memory traffic of loading redundant data and the necessity of instructions that manipulate single SIMD lanes. These underlying matrices may be stored in column major or row major fashion, the code generation approach allows for flexibility in this regard. Carrying out the sum factorization algorithm with these stacked tensors, the resulting stacked tensor $\partial_0\hat{U}|\partial_1\hat{U}|\partial_2\hat{U}|\hat{U}$ will be given as an interleaved tensor as well. This can be seen as an AoS layout, where the inner structure is of fixed size 4. The layout is illustrated in figure 4.1. All data structures are aligned to the vector size to allow aligned loads into SIMD registers. We will now discuss how the AoS nature of the data structure affects step 2 of algorithm 2.3, the quadrature loop.

Our idea of vectorizing the quadrature loop is based on the idea to treat four quadrature points at a time. We have found it beneficial to neglect the tensor product structure of the quadrature loop here and instead use flat indexing. In order

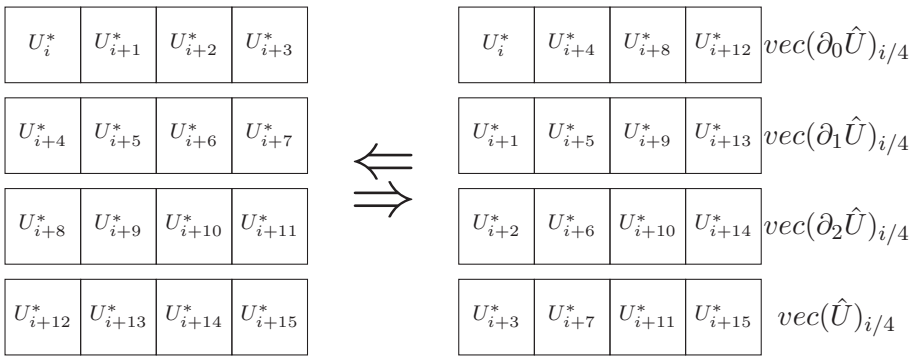


Figure 4.2: Register transposition needed in the quadrature loop: Four SIMD vectors of $U^* := vec(\partial_0\hat{U}|\partial_1\hat{U}|\partial_2\hat{U}|\hat{U})$ are loaded and transposed in-place. The resulting four SIMD vectors have the layout needed for an efficient, straightforward vectorized implementation of the quadrature loop. The inverse operation is needed to get the correct layout for the input tensor for step 3 of algorithm 2.3.

to assemble the tensors $R^v_{i_0i_1i_2}$ and $R^{\partial_k v}_{i_0i_1i_2}$ from algorithm 2.3 with vector arguments, we need to undo the AoS layout. We do so in the quadrature loop by applying a transposition of four consecutive SIMD vectors of $U^* := vec(\partial_0\hat{U}|\partial_1\hat{U}|\partial_2\hat{U}|\hat{U})$. When before the transposition a SIMD register would hold function evaluation and evaluation of the gradient at one quadrature point, it will hold one of these quantities at four consecutive quadrature points afterwards. The procedure is illustrated in figure 4.2. Implementation of this transposition code is crucial for the overall performance of the algorithm and will be studied in detail in section 4.3, where the core idea is to hide the cost of the transposition algorithm behind the floating point workload of the quadrature loop.

Step three of algorithm 2.3 again expects an AoS type layout, that is not naturally imposed on the tensors $R^v_{i_0i_1i_2}$ and $R^{\partial_k v}_{i_0i_1i_2}$. We achieve this by applying the transposition algorithm from figure 4.2 again. The overall quadrature loop algorithm is summarized in algorithm 4.1. It is worth noting that the i-loop does not need a tail loop although the total number of quadrature points is not necessarily divisible by four: It is sufficient to overallocate the storage of R to assure that the last loop iteration cannot write out of bounds. Step three of algorithm 2.3 is treated in the exact same way step one is, with the notable exception of the necessity to accumulate the results of four sum factorization kernel into the residual tensor. With the chosen memory layout, this requires an intra-register reduction. The implementation of this operation is subject to special care in section 4.3, as it benefits greatly from microarchitecture-dependent optimization.

So far, we have studied the explicitly vectorized assembly algorithm under quite a number of simplifications and assumptions, such as only studying volume integrals, restricting to residual evalutions, the problem being defined in 3D space, the SIMD width being 256 bits, the use of double precision arithmetic and the PDE residual $r_h(u, v)$ depending on u, ∇u, v and ∇v. We will now discuss which of these

Algorithm 4.1: Explicitly vectorized quadrature loop for a SIMD width of 4 and a total of M quadrature points. We use U^* and R^* as shortcuts for $vec(\partial_0\hat{U}|\partial_1\hat{U}|\partial_2\hat{U}|\hat{U})$ and $vec(R^{\partial_0 v}|R^{\partial_1 v}|R^{\partial_2 v}|R^v)$. The j loop is implemented through SIMD vectorization. This is formulated in the same frame as algorithm 2.3: Assembly of a 3D volume integral for a second order elliptic problem.

1 $i \leftarrow 0$
2 **while** $i < M$ **do**
3 TransposeRegisters($U^*_{4i}, \ldots, U^*_{4i+15}$)
4 **for** $j \in \{0, 1, 2, 3\}$ **do**
5 $R^*_{4i+j} \leftarrow \tilde{r}^{volume}_{\partial_0 v_h}(U^*_{4i+j}, U^*_{4(i+1)+j}, U^*_{4(i+2)+j}, U^*_{4(i+3)+j}, \mu_T, \hat{\xi})$
6 $R^*_{4(i+1)+j} \leftarrow \tilde{r}^{volume}_{\partial_1 v_h}(U^*_{4i+j}, U^*_{4(i+1)+j}, U^*_{4(i+2)+j}, U^*_{4(i+3)+j}, \mu_T, \hat{\xi})$
7 $R^*_{4(i+2)+j} \leftarrow \tilde{r}^{volume}_{\partial_2 v_h}(U^*_{4i+j}, U^*_{4(i+1)+j}, U^*_{4(i+2)+j}, U^*_{4(i+3)+j}, \mu_T, \hat{\xi})$
8 $R^*_{4(i+3)+j} \leftarrow \tilde{r}^{volume}_{v_h}(U^*_{4i+j}, U^*_{4(i+1)+j}, U^*_{4(i+2)+j}, U^*_{4(i+3)+j}, \mu_T, \hat{\xi})$
9 TransposeRegisters($R^*_{4i}, \ldots, R^*_{4i+15}$)
10 $i \leftarrow i + 4$

assumptions were made for the sake of presentability in this thesis and which are actual limitations of the approach.

Extension to boundary and interior facet integrals can be implemented by replacing the basis evaluation matrix for the normal direction of the facet with a special matrix consisisting of only one quadrature point. Note, that this quadrature point is either located at 0.0 or 1.0 depending on the facet being on the upper or lower boundary of the reference cube. Consequently, facet kernel implementations depend on the embedding of the facet into the reference cube. Treating this dependency in the code generation workflow has the additional advantage of being able to exploit additional generation-time knowledge about geometric quantities. Facet kernel implementations generally have a lower arithmetic intensity, as the number of floating point operations both in sum factorization kernels and in the quadrature loop is of reduced dimensionality, while the number of DOFs that the kernel depends on is not reduced.

While we are typically denoting the Kronecker product of basis evaluation matrices for a sum factorization kernel in x-y-z order, it is important to understand that we might implement the kernel in any order, as long as we correctly access the entries in the output tensor. In fact, some orders are strictly better in terms of the total floating point operations carried out. This can be seen best in the case of a facet integral: Performing the tensor contraction associated with the reduced size matrix first will result in a temporary output of reduced dimensionality. For anisotropic polynomial degree, the optimal order depends on the degree tuple even for volume integrals. We always aim for the implementation with the minimum number of floating point operations available. Not doing so would taint any SIMD throughput measurements provided in section 5.

Although algorithm 2.3 is formulated for a residual evaluation, the same techniques can be applied when assembling jacobians or their action on a vector. In the latter case, an additional finite element function may need to be evaluated for nonlinear problems, as described in section 3.3. This additional evaluation is also done in a sum factorized fashion and gives rise to additional vectorization opportunities that we will describe below. Although not the primary target of this work, sum factorization can also be used when assembling jacobian matrices: In that case, step two and three of algorithm 2.3 are executed once per basis function in the ansatz space. This introduces additional d loops wrapped around steps two and three. Evaluations of the ansatz function are implemented as the product of the 1D basis functions directly in the quadrature loop.

The scenario studied so far in this section was a best-case scenario in the sense that the number of sum factorization kernels within one step of algorithm 2.3 matches the number of SIMD lanes available. Although this described special case is an absolutely valid use case with many physically relevant applications, this ratio will often be below or above 1. The ratio decreases for 2D simulations, where the gradient has one component less and for PDE models that do not depend on all given quantities, such as e.g. equation 2.1 in the absence of a reaction term. The ratio can also decrease if the number of available SIMD lanes is increased by either doing single precision calculations or moving to an instruction set with wider SIMD units, such as the AVX-512 instruction set. The ratio increases in the above cases of calculating the action of the jacobian in the nonlinear case and on facet integrals due to the necessity to evaluate finite element functions w.r.t. the inside and the outside cell and in the case of systems of PDEs. In the case of systems of PDEs, finite element functions defined over different spaces cannot be fused together into one vectorization strategy, as the loop bounds of the sum factorization kernel do not match.

In those cases where there is less sum factorization kernels than there are SIMD lanes, we have the possibility to fuse this smaller amount of kernels and ignore any values in the additional SIMD lanes. Of course, such an approach wastes some of the floating point capabilities of the processor and artificially increases the measured floating point throughput of the computation. We will consider such strategies in the code generation driven autotuning process from section 4.4 whenever there is a mismatch of one SIMD lane, but not beyond that.

If there are more sum factorized quantities than SIMD lanes, sum factorization kernels can be grouped into batches of SIMD width given that their tensor bounds agree. If the number of quantities is not divisible by the number of SIMD lanes, the above padding strategy might again become necessary. Special care needs to be taken concerning the input tensor X from equation 4.5: So far we only looked at fusing kernels that operate on the same input tensor X, where the stacked input tensor $X|X|\dots|X$ could be realized as a broadcast instruction into a SIMD register. However, sum factorization kernels with differing input tensors can be fused together as well, sacrificing the very efficient broadcast instruction. In the

AVX2 and AVX-512 instruction sets, dedicated instructions to load a value into the lower and upper half of a register exist. These stem from backwards compatibility with older instruction sets during the SIMD transition from SSE to AVX-512. We leverage these instructions to fuse kernels with two different input tensors. This scenario often arises on interior facets, where quantities need to be evaluated on the inside and outside cell:

$$U^-|\partial_0 U^-|U^+|\partial_0 U^+ = \left(A_F^{\uparrow}|D_F^{\uparrow}|A_F^{\downarrow}|D_F^{\downarrow} \otimes A|A|A|A \otimes A|A|A|A\right) X^-|X^-|X^+|X^+ \tag{4.6}$$

Here, $A_F^{\uparrow}$ and $A_F^{\downarrow}$ denote the basis evaluation matrices for the direction of reduced dimension with the arrow indicating whether the facet is embedded into the lower or upper face of the reference cube. We will restrict ourselves to treating two different input tensors at a time in section 4.4, although arbitrary input tensors could be combined by means of vector gather instructions. However, these instructions are not amortized in our workload setting and should be avoided if alternatives exist. In step three of algorithm 2.3, the analogon of the SIMD broadcast is horizontal addition. We implement accumulation into multiple output tensors by providing reduction functions over the lower and upper half of a SIMD register.

While the above strategies manage to mitigate some of the mentioned disadvantages of our loop-fusion based SIMD vectorization, we will now turn to extend the possibilities of the approach by introducing a new concept. This is necessary in order to be able to target the AVX-512 instruction set, where the number of SIMD lanes commonly exceeds the number of sum factorized quantities to be evaluated.

4.2.2 Loop Splitting Based Strategies

While the loop fusion vectorization from section 4.2.1 tries to fuse multiple sum factorization kernels, the idea of this section is to split the workload of one sum factorization kernel such that execution can make use of SIMD parallelism. This does not suffer from the disadvantages seen above: Increased memory footprint of the kernel and the necessity of memory layout adjustments. On the other hand, these splitting based vectorization techniques come with the disadvantage that maximum efficiency can only be reached if the kernel structure exhibits loop bounds with suitable divisibility constraints. We will now explore these strategies. For the sake of readability, we again limit ourselves to a SIMD width of four lanes and to the treatment of the evaluation of $\hat{U}$ via

$$\hat{U} = \left(A^{(0)} \otimes A^{(1)} \otimes A^{(2)}\right) X \tag{4.7}$$

and discuss possible generalizations later.

We base our strategy on the idea to split the set of quadrature points into a number of subsets equal to the SIMD width. We do so by choosing one direction i and splitting the basis evaluation matrix $A^{(i)}$ into w matrices, where w is the SIMD width. For now, we assume the number of 1D quadrature points to be divisible by

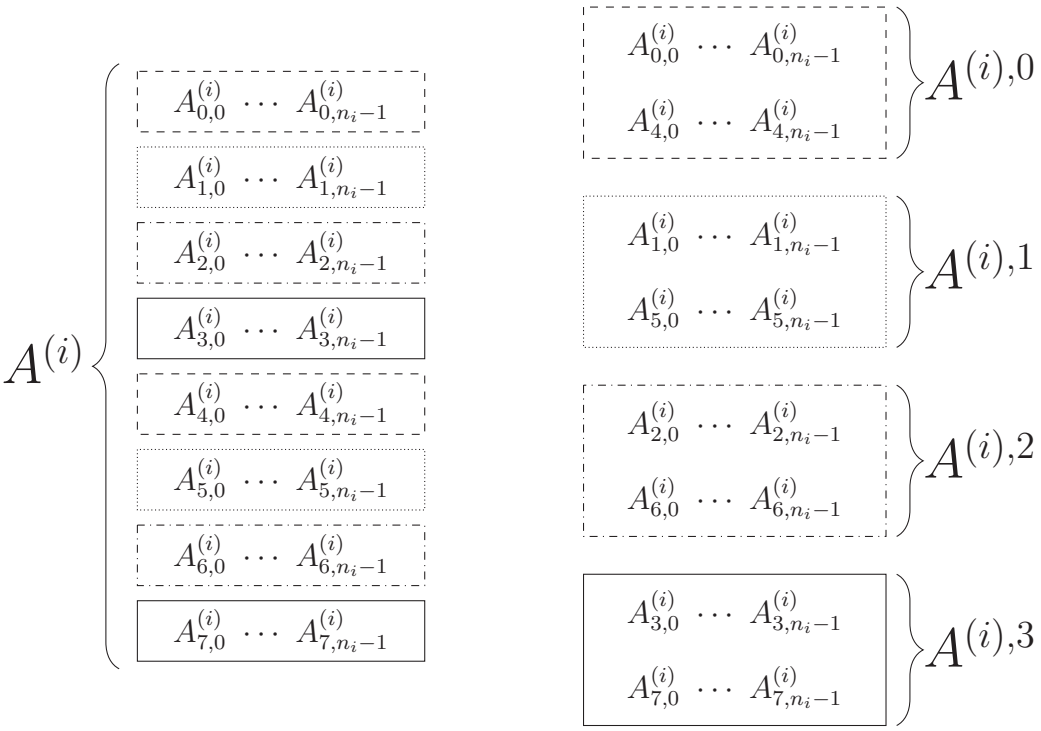


Figure 4.3: Slicing of the basis evaluation matrix: $A^{(i)}$ is split into four matrices $A^{(i),s}$ with $s \in \{0, \ldots, w-1\}$ in a circular fashion. The sliced matrices are used in sum factorization kernels that compute $\frac{1}{w}$ of the entries of the output tensor $\hat{U}$.

w and discuss other cases later on. The index $s \in \{0, \ldots, w-1\}$ denotes the index of the slice $A^{(i),s}$ of the basis evaluation matrix. Note that we do not split $A^{(i)}$ in a blocked fashion, but in a circular one, as illustrated in figure 4.3. Carrying out the sum factorized computation from equation 4.7 with a slice $A^{(i),s}$ instead of $A^{(i)}$ leads to an output tensor $\hat{U}^s$ only containing evaluations at $1/w$ of the quadrature points. Equation 4.7 can then be recast into the following equivalent formulation using the notation from section 4.2.1:

$$\begin{aligned}\hat{U}^0|\hat{U}^1|\hat{U}^2|\hat{U}^3 = \big(&A^{(0),0}|A^{(0),1}|A^{(0),2}|A^{(0),3} \\ &\otimes A^{(1)}|A^{(1)}|A^{(1)}|A^{(1)} \\ &\otimes A^{(2)}|A^{(2)}|A^{(2)}|A^{(2)}\big)\, X|X|X|X\end{aligned} \tag{4.8}$$

The fact that we have sliced $A^{(0)}$ in a circular fashion leads to the following, desirable property that allows us to load data of the resulting tensor $\hat{U}^0|\hat{U}^1|\hat{U}^2|\hat{U}^3$ without further manipulation of memory layout:

$$vec(\hat{U}^0|\hat{U}^1|\hat{U}^2|\hat{U}^3) = vec(\hat{U}) \tag{4.9}$$

We observe that in contrast to section 4.2.1, the combined basis evaluation matrices do not have to be explicitly set up beforehand, as $vec(A^{(0),0}|A^{(0),1}|A^{(0),2}|A^{(0),3}) =$

$vec(A^{(0)})$ and $A^{(i)}|A^{(i)}|A^{(i)}|A^{(i)}$ can again be implemented as a broadcast of elements of $A^{(i)}$. The loop splitting based approach described in this section does not depend on the problem structure in the same way as the loop fusion based one from section 4.2.1. As there is no need to group multiple sum factorization kernels, the approach vectorizes equally well in two and three dimensional space, as well as with arbitrary combination of terms present in the problem formulation. However, applicability of the approach depends on the divisibility of the number of 1D quadrature points. Having this constraint on the number of quadrature points is not as bad as having it on the number of basis functions: Artificially increasing the number of quadrature points is equivalent to overintegration, which even yields additional accuracy for problems that are not exactly integrated. However, one has to be cautious as the increase in floating point operations affects the whole algorithm and not only the sum factorization kernel to be vectorized. We now study the additional cost of such a procedure and refer to section 4.4 for discussion of the necessary trade off decisions.

Recall that the number of quadrature points per direction is given as a tuple $\mathbf{m} = (m_0, \dots, m_{d-1})$ and the number of basis functions per direction as a tuple $\mathbf{n} = (n_0, \dots, n_{d-1})$. The floating point cost $\mathcal{C}^{SF}(\mathbf{m}, \mathbf{n})$ of a single sum factorization kernel reads

$$\mathcal{C}^{SF}(\mathbf{m}, \mathbf{n}) = 2 \sum_{k=0}^{d-1} \prod_{i=0}^{k} m_i \prod_{j=k}^{d-1} n_j. \tag{4.10}$$

We observe that $\mathcal{C}^{SF}(\mathbf{m}, \mathbf{n})$ is linear in m_0. This also holds for other relevant parts of the algorithm such as the quadrature loop. Consequently, the total cost of the algorithm will be increased by a factor of $\frac{m^*}{m_0}$, if the number of quadrature points in the first direction is increased to m^*. Setting m^* to the next multiple of the SIMD width w, we get a cost increase of $\lceil \frac{m_0}{w} \rceil \cdot \frac{w}{m_0}$. For sufficiently high numbers of quadrature points, this increase becomes reasonably small. In the worst case scenario of a $\mathbb{Q}_1$ discretization with minimal quadrature order however, it can be as high as $\frac{w}{2}$.

In section 4.2.1, we briefly touched on the necessity to reorder the tensor contractions in a sum factorization kernel in order to get to an implementation with a minimal number of floating point operations. This makes the loop splitting based strategy described in this section only partly applicable to facet integrals, as in order to get the desired result, the basis evaluation matrix of the first tensor contractions needs to be split. On facet integrals, this matrix has only one quadrature point and is not splittable. Splitting a different basis evaluation matrix is of course possible, but prevents vectorization of the first tensor contraction. See this example, where we assume only two SIMD lanes for the sake of simplicity:

$$\hat{U}^0|\hat{U}^1 = \left(A_F^{(0)}|A_F^{(0)} \otimes A^{(1),0}|A^{(1),1} \otimes A^{(2)}|A^{(2)}\right) X|X \tag{4.11}$$

Here, the contraction with $A_F^{(0)}$ needs to be done first in order to be FLOP-minimal, but the same calculation would be carried out on each SIMD lane - effectively turning the code into an unvectorized one.

In this section, we formulated the implementation idea in terms of the fusion based vectorization described in section 4.2.1. The same ideas could have been developed from a different perspective, but having it formulated using the same notation will be of great benefit for developing hybrid strategies in section 4.2.3 and also for the vectorization heuristics in the code generator described in section 4.4.

4.2.3 Hybrid Strategies

Neither the strategy described in section 4.2.1 nor the strategy from section 4.2.2 are in general a perfect fit for vectorization with wider SIMD widths. For the loop fusion strategy, the problem description will usually not exhibit enough quantities that can be computed in parallel. On the other hand, the loop splitting strategy leads to prohibitively severe constraints on the number of quadrature points with increasing SIMD width. We now seek to combine these two strategies into a hybrid vectorization strategy mitigating their individual disadvantages.

We have formulated the loop fusion approach from section 4.2.1 and the loop splitting one from section 4.2.2 using a common framework: A set of sum factorization kernels is collected into a larger kernel, which is suitable for vectorization, where these kernels are potentially obtained by first splitting the given sum factorization kernels. We will now generalize this to arbitrary SIMD widths and combinations of these techniques. We define f and s, such that f denotes the number of sum factorization kernels to be combined through loop fusion and s denotes the number of slices these are split into. We only treat those cases where $f \cdot s = w$ with w the number of SIMD lanes. For $f = 4$ and $s = 2$, this allows a natural extension of section 4.2.1 for a SIMD width $w = 8$ (AVX-512), which calculates $\hat{u}_h$ and $\nabla\hat{u}_h$ in parallel and introduces a divisibility constraint of 2 on m_0:

$$\begin{aligned}
&\partial_0\hat{U}^0|\partial_0\hat{U}^1|\partial_1\hat{U}^0|\partial_1\hat{U}^1|\partial_2\hat{U}^0|\partial_2\hat{U}^1|\hat{U}^0|\hat{U}^1 \\
&\qquad = (D^{(0),0}|D^{(0),1}|A^{(0),0}|A^{(0),1}|A^{(0),0}|A^{(0),1}|A^{(0),0}|A^{(0),1} \\
&\qquad\quad \otimes A^{(1)}|A^{(1)}|D^{(1)}|D^{(1)}|A^{(1)}|A^{(1)}|A^{(1)}|A^{(1)} \\
&\qquad\quad \otimes A^{(2)}|A^{(2)}|A^{(2)}|A^{(2)}|D^{(2)}|D^{(2)}|A^{(2)}|A^{(2)}) \\
&\qquad\quad X|X|X|X|X|X|X|X
\end{aligned} \tag{4.12}$$

Again, we will study the memory layout implications of implementing equation 4.12. The stacked basis evaluation matrices are preevaluated and loaded from memory, just like in section 4.2.1. The input tensor $X|X|X|X|X|X|X|X$ is implemented

Algorithm 4.2: A generic shuffling algorithm for hybrid strategies that generalizes the transposition from figure 4.2. Here, f and s denote the number of quantities fused together and s the splitting factor as described in section 4.2.2. We only treat the case of $f \cdot s = w$, where w is the SIMD width.

```
1 function GenericTranspose(data[f][w])
2     for i = 0, ..., f − 1 do
3         for j = 0, ..., f − 1 do
4             for k = 0, ..., s − 1 do
5                 swap(data[i][js + k], data[j][is + k])
```

Algorithm 4.3: General quadrature loop with a hybrid vectorization strategy for SIMD width w: The input data is given as a set of flat tensors $\{U_j^*\}$. Similarly, the output will be written as a set of flat tensors $\{R_k^*\}$. Each of of these flat tensors results from a fusion of a set of w sum factorization kernels. The function f maps those tensors to the number of fused quantities as described in section 4.2.3.

```
1 i ← 0
2 while i < M do
3     for j ∈ {0, ...} do
4         TransposeRegisters((U_j^*)_{f(U_j^*)i}, ..., (U_j^*)_{f(U_j^*)i+f(U_j^*)w−1})
5     for k ∈ {0, ...} do
6         R_k^* ← Quadrature computation
7         TransposeRegisters((R_k^*)_{f(R_k^*)i}, ..., (R_k^*)_{f(R_k^*)i+f(R_k^*)w−1})
8     i ← i + w
```

by broadcasting the values of X. The only remaining question is how the memory layout of the output tensor $\partial_0\hat{U}^0|\partial_0\hat{U}^1|\partial_1\hat{U}^0|\partial_1\hat{U}^1|\partial_2\hat{U}^0|\partial_2\hat{U}^1|\hat{U}^0|\hat{U}^1$ affects the quadrature loop implementation. Independently of the choice of f and s, the quadrature loop treats w quadrature points at a time. However, to get w values of the f quantities present in the data, we need to shuffle f consecutive vectors. This results in the need for non-square matrix shuffles which generalize the transposition algorithm from figure 4.2. We fix the intra-register layout to be such that kernels resulting from splitting need to be on adjacent SIMD lanes. In other words, we disallow tensors like $\partial_0\hat{U}^0|\partial_1\hat{U}^0|\partial_2\hat{U}^0|\hat{U}^0|\partial_0\hat{U}^1|\partial_1\hat{U}^1|\partial_2\hat{U}^1|\hat{U}^1$. These generic shuffle operations are described in algorithm 4.2. Note how the generic formulation from algorithm 4.2 generalizes both the case of purely fusion based vectorization from figure 4.2 (where $f = w$) and the case of purely splitting based vectorization from section 4.2.1 (where $s = w$ and the transposition is the identity). The quadrature loop algorithm 4.3 is further complicated by the fact that an integration kernel might consist of more than one vectorized sum factorization kernel and that the choice of f and s can differ for each of these.

We have seen, that the techniques of sections 4.2.1 and 4.2.2 can be combined to

mitigate their disadvantages and target wider SIMD widths. However, for a given problem, the number of possible vectorization strategies grows exponentially in the number of input kernels and it is not a priori known, which one performs best. This issue is targeted in section 4.4 by introducing a cost model approach.

4.3 Performance Critical SIMD Operations

SIMD abstraction libraries as described in section 3.2.3 do a great job at keeping many very technical low-level considerations away from the developer. However, it is always worth taking a closer look at the assembly code that is generated from using these libraries. This is especially true if several sequences of instructions exist that accomplish a given task. For the vectorization algorithms presented in section 4.2, two such operations stood out in our experiments: Intra-register reduction (or *horizontal addition*) and the register transposition algorithm 4.2. Therefore, we will study these in a bit more detail in this section. The assembly code provided here is obtained using the API of the Godbolt Compiler Explorer [48]. An interactive version of the examples is available by clicking the given Godbolt short code[1].

4.3.1 Intra-register Reduction

We only consider intra-register addition, although similar considerations may apply to other reductions on SIMD registers. Focussing on the more relevant double precision case, we will study implementations for the AVX2 and AVX-512 instruction sets.

AVX2 Double Precision

We will assume the Intel Haswell architecture for the remainder of this paragraph. Horizontal addition of four double values in an AVX2 register using the implementation from Agner Fog's vector class library [42] and `g++` is shown in figure 4.4. The same implementation is obtained from other libraries from section 3.2.3 that provide horizontal addition functionality, such as xsimd [100]. With the horizontal addition being a bottleneck in our experiments (we will see this in detail in section 5), it is worth studying the generated assembly code in more detail using Agner Fog's instruction tables [40], a vendor-independent source of information about instruction μ-ops, latency and throughput. As the horizontal addition function will always be inlined in our use case, we assume the input being in a register (ignoring the `vmovapd` instruction) and assume the costly `vzeroupper` instruction to be moved towards the bottom of the coarse-grained enclosing function. Following the analysis of horizontal addition in [27], we count the executed (unfused) μ-ops and on which port they execute. The vector class library implementation from figure 4.4 requires

[1] Godbolt short codes can also be entered manually: `https://www.godbolt.org/z/<shortcode>`

```
#include<vectorclass.h>
double hadd(const Vec4d& a) {
    __m256d t1 = _mm256_hadd_pd(a,a);
    __m128d t2 = _mm256_extractf128_pd(t1,1);
    __m128d t3 = _mm_add_sd(_mm256_castpd256_pd128(t1),t2);
    return _mm_cvtsd_f64(t3);
}
```

Godbolt code: 40jd15

Compiler: g++ 8.3

```
_Z4haddRK5Vec4d:
        vmovapd ymm0, YMMWORD PTR [rdi]
        vhaddpd ymm0, ymm0, ymm0
        vextractf128    xmm1, ymm0, 0x1
        vaddsd  xmm0, xmm0, xmm1
        vzeroupper
        ret
```

Figure 4.4: Godbolt disassembly of VCL's implementation of horizontal addition for SIMD vectors of four double precision values.

a total of five μ-ops, three of which execute on the processor's shuffle port. Most notably, the `vhaddpd` instruction takes four μ-ops and has the worst throughput. In contrast, the alternative implementation from figure 4.5 avoids `vhaddpd`, requires only four μ-ops and removes some of the pressure on the shuffle port (only two μ-ops on it). We have made good experiences with such alternative implementation, although rigorously benchmarking such implementations is impossible in isolation, as the execution context is crucial for the performance. We therefore postpone benchmarks for this problem to section 5.2, where we have the benchmark setup for the full finite element assembly problem available.

AVX-512 Double Precision

Horizontal addition does naturally not scale perfectly with the size of the SIMD register. Being already a bottleneck in AVX2 applications, it is worth taking a very close look at available implementations in the transition to the AVX-512 instruction set. We limit ourselves to the Intel Skylake architecture with the AVX-512 instruction set e.g. the system described in appendix A.2[2]. With the introduction of the AVX-512 instruction set, Intel added an intrinsic function `_mm512_reduce_add_pd` for horizontal addition. Unfortunately, the common compilers differ vastly in what code they generate from that intrinsic, as can be seen in figure 4.6. The implementation within the vector class library resembles the GCC implementation from figure 4.6, as it reuses its 256 bit implementation. In order to achieve best

[2] On a side note, many of the problems regarding scalability with SIMD width are even more pronounced on the Intel Xeon Phi (Knights Landing) architecture.

```
#include<vectorclass.h>
double hadd(const Vec4d& a) {
    __m128d t1 = _mm_add_pd(a.get_low(), a.get_high());
    __m128d t2 = _mm_unpackhi_pd(t1, t1);
    __m128d t3 = _mm_add_sd(t2, t1);
    return _mm_cvtsd_f64(t3);
}
```

Godbolt code: qlY759

Compiler: g++ 8.3

```
_Z4haddRK5Vec4d:
        vmovapd ymm0, YMMWORD PTR [rdi]
        vextractf128    xmm1, ymm0, 0x1
        vaddpd  xmm0, xmm0, xmm1
        vunpckhpd       xmm1, xmm0, xmm0
        vaddsd  xmm0, xmm1, xmm0
        vzeroupper
        ret
```

Figure 4.5: Godbolt disassembly of an alternative implementation of horizontal addition for SIMD vectors of four double precision values. Implementation taken from [27].

performance across compilers, we implement horizontal addition with a combination of elements of the vector class library and intrinsics:

```
#include<vectorclass.h>
double hadd(const Vec8d& a) {
  auto t0 = a.get_low() + a.get_high();
  auto t1 = t0.get_low() + t0.get_high();
  auto t2 = _mm_unpackhi_pd(t1, t1);
  auto t3 = _mm_add_sd(t2, t1);
  return _mm_cvtsd_f64(t3);
}
```

This implementation matches the generated assembly of Clang in figure 4.6 and is a natural extension of the AVX2 horizontal addition presented in figure 4.5.

4.3.2 Structure of Arrays/Array of Structures Transformation

Data permutations across SIMD registers are often necessary in order to enforce memory layouts that allow for maximum usage of SIMD registers. However, these permutation operations can also easily become a bottleneck of the application. It is therefore of vital importance to generate an optimal sequence of instructions for a given data permutation. Dedicated research regarding this topic exists in [104], where permutations are expressed as a sequence of elementary operations, whose execution order is then optimized. As we do not deal with general SIMD

```cpp
#include<vectorclass.h>
double hadd(const Vec8d& a) {
  return _mm512_reduce_add_pd(a);
}
```

Godbolt code: lkNB9S

Compiler: g++ 8.3

```
_Z4haddRK5Vec8d:
        vmovapd zmm0, ZMMWORD PTR [rdi]
        vextractf64x4   ymm1, zmm0, 0x1
        vaddpd  ymm1, ymm1, ymm0
        vextractf64x2   xmm0, ymm1, 0x1
        vaddpd  xmm0, xmm0, xmm1
        vhaddpd xmm0, xmm0, xmm0
        vzeroupper
        ret
```

Godbolt code: SA28vA

Compiler: clang++ 8.0

```
_Z4haddRK5Vec8d:
        vmovapd ymm0, ymmword ptr [rdi]
        vaddpd  ymm0, ymm0, ymmword ptr [rdi + 32]
        vextractf128    xmm1, ymm0, 1
        vaddpd  xmm0, xmm0, xmm1
        vpermilpd       xmm1, xmm0, 1
        vaddpd  xmm0, xmm0, xmm1
        vzeroupper
        ret
```

Godbolt code: COrts4

Compiler: icc 19.1

```
_Z4haddRK5Vec8d:
        vmovups   zmm1, ZMMWORD PTR [rdi]
        vshuff32x4 zmm0, zmm1, zmm1, 238
        vaddpd    zmm2, zmm0, zmm1
        vpermpd   zmm3, zmm2, 78
        vaddpd    zmm4, zmm2, zmm3
        vpermpd   zmm5, zmm4, 177
        vaddpd    zmm0, zmm4, zmm5
        vzeroupper
        ret
```

Figure 4.6: Godbolt disassembly of the AVX-512 intrinsic function for horizontal addition with the latest version of the GNU compiler, Clang and the Intel compiler. Generated assembly code differs a lot between compilers. The assembly generated by Clang needs the fewest μ-ops: Six, only two of which are executed on the shuffle port.

```cpp
#include<vectorclass.h>
void transpose(Vec4d& a0, Vec4d& a1, Vec4d& a2, Vec4d& a3) {
  Vec4d b0,b1,b2,b3;
  b0 = blend4d<0,4,2,6>(a0,a1);
  b1 = blend4d<1,5,3,7>(a0,a1);
  b2 = blend4d<0,4,2,6>(a2,a3);
  b3 = blend4d<1,5,3,7>(a2,a3);
  a0 = blend4d<0,1,4,5>(b0,b2);
  a1 = blend4d<0,1,4,5>(b1,b3);
  a2 = blend4d<2,3,6,7>(b0,b2);
  a3 = blend4d<2,3,6,7>(b1,b3);
}
```

Godbolt code: vZIx3D

Compiler: g++ 8.3

```
_Z9transposeR5Vec4dS0_S0_S0_:
        vmovapd ymm5, YMMWORD PTR [rdi]
        vmovapd ymm7, YMMWORD PTR [rdx]
        vunpcklpd       ymm2, ymm5, YMMWORD PTR [rsi]
        vunpcklpd       ymm3, ymm7, YMMWORD PTR [rcx]
        vunpckhpd       ymm0, ymm5, YMMWORD PTR [rsi]
        vunpckhpd       ymm1, ymm7, YMMWORD PTR [rcx]
        vinsertf128     ymm4, ymm2, xmm3, 1
        vmovapd YMMWORD PTR [rdi], ymm4
        vperm2f128      ymm2, ymm2, ymm3, 49
        vinsertf128     ymm4, ymm0, xmm1, 1
        vperm2f128      ymm0, ymm0, ymm1, 49
        vmovapd YMMWORD PTR [rsi], ymm4
        vmovapd YMMWORD PTR [rdx], ymm2
        vmovapd YMMWORD PTR [rcx], ymm0
        vzeroupper
        ret
```

Figure 4.7: Godbolt disassembly of a AoS⇒SoA register transposition of four AVX2 registers as given in figure 4.2.

permutations, but with the rather structured ones from algorithm 4.2, we instead implement them semi-manually using the vector class library from section 3.2.3: We split the permutation into a sequence of elementary permutations manually but delegate the choice of instructions to the library. The instruction selection is implemented by providing the permutation pattern as a template parameter, which allows for compile-time analysis of the pattern. In figure 4.7, we see the disassembly of the four SIMD vector transpose from figure 4.2. The generated assembly for this transposition algorithm does not change between the major compilers.

Data permutations for other values of f and s in algorithm 4.2 can be implemented analogously. For more SIMD lanes, this usually results in a hierarchic approach, e.g.

a transposition of 8 vectors can be implemented as a block transpose of the 4×4 blocks, which have been transposed exactly like in figure 4.7.

Many instructions in the disassembly in figure 4.7 are executed on the shuffle port. It is worth noting that modern Intel processors are capable of executing instructions on multiple ports within one cycle, implementing a form of ILP. Given this parallelism, it is possible to overlap the permutation algorithm with the FLOP workload of the quadrature loop. It is our goal to effectively hide the permutation in this way. Performance numbers given in chapter 5 indicate that this is indeed possible.

4.4 Integration into the Code Generator

Section 4.2 introduced a whole class of SIMD vectorization strategies. We will now turn to the important question of how to handle this wide range of vectorization opportunities. How can they be implemented in an automatic fashion from a source code generator? This will be targeted in section 4.4.1, where we will see how sum factorization kernels can be implemented as `loopy` kernels and how the C++ code for them looks like. Section 4.4.2 will provide answers to how an optimal vectorization strategy can be selected from within the code generator. This includes systematic traversal of the vectorization strategies from section 4.2 and a heuristic performance model. Section 4.4.3 introduces autotuning as a (costly) alternative to the heuristic model. Finally, we elaborate on some ways to extend the capabilities of the code generator for two use cases: Section 4.4.4 describes how custom geometry mappings are integrated into the code generator and section 4.4.5 describes how symbolic form manipulation can be used to implement matrix-free block preconditioners.

4.4.1 From Sum Factorization Kernels to Loopy Kernels

We will start off with how we symbolically represent sum factorization kernels and then move on to how these are represented as `loopy` kernels and how generated C++ code for them looks like.

Intermediate Representations

We have seen in section 3.3 that the code generation process is based entirely on AST transformations (UFL to `loopy`, `loopy` to C) that are implemented via recursive tree traversals with type-based function dispatch. These algorithms work best if the tree transformation is fully local, meaning that the visitor object is completely stateless. Our vectorization strategies - especially those based on loop fusion - are inherently non-local, as they depend on the occurence of other terms in the PDE. We therefore apply a procedure where we visit the expression twice:

- During the first tree traversal, any quantity that is calculated through a sum factorization kernel is represented by a dedicated AST node `SumfactKernel`, that stores all the relevant information, but does not yet introduce any array data structures in the `loopy` kernel.
- After the first tree traversal, an algorithm selects a vectorization strategy by providing a mapping of all the `SumfactKernel` objects in the AST to objects of `VectorizedSumfactKernel` nodes which have vectorization information attached. We will study the decision-making algorithm in detail in section 4.4.2.
- A second tree traversal is done, in which these nodes with vectorization information are realized by `loopy` statements. Only the kernels that result from this second traversal are fully functional `loopy` kernels from which code can be generated.

The `SumfactKernel` AST node contains the following pieces of information: A sequence of basis evaluation matrix objects, an object representing the input (or resp. output) tensor and additional `loopy` information about loops and conditionals that need to be wrapped around the kernel implementation, or dependencies on other `loopy` statements. The `VectorizedSumfactKernel` node implements the same interface, but internally stores a tuple of `SumfactKernel` objects that hold the actual data.

From Sum Factorization Kernels to C Code

We recall the sum factorization approach in tensor notation from equation 2.27 (limiting ourselves to $d = 3$ for readability):

$$\hat{U}_{i_0 i_1 i_2} = \sum_{j_2=0}^{n_2-1} A^{(2)}_{i_2,j_2} \sum_{j_1=0}^{n_1-1} A^{(1)}_{i_1,j_1} \sum_{j_0=0}^{n_0-1} A^{(0)}_{i_0,j_0} X_{j_0 j_1 j_2} \tag{4.13}$$

Introducing intermediate tensors T^i this series of tensor contractions can be interpreted as follows:

$$T^0_{i_0 j_1 j_2} = \sum_{j_0=0}^{n_0-1} A^{(0)}_{i_0,j_0} X_{j_0 j_1 j_2} \tag{4.14}$$

$$T^1_{j_1 j_2 i_0} = T^0_{i_0 j_1 j_2} \tag{4.15}$$

$$T^2_{i_1 j_2 i_0} = \sum_{j_1=0}^{n_1-1} A^{(1)}_{i_1,j_1} T^1_{j_1 j_2 i_0} \tag{4.16}$$

$$T^3_{j_2 i_0 i_1} = T^2_{i_1 j_2 i_0} \tag{4.17}$$

$$T^4_{i_2 i_0 i_1} = \sum_{j_2=0}^{n_2-1} A^{(2)}_{i_2,j_2} T^3_{j_2 i_0 i_1} \tag{4.18}$$

$$\hat{U}_{i_0 i_1 i_2} = T^4_{i_2 i_0 i_1} \tag{4.19}$$

As can be seen, each tensor contraction is followed by a permutation of the intermediate tensor. An implementation that strictly follows equations 4.14 to 4.19 would be quite inefficient, as the FMA operations from the contraction would outperform the permutation operations turning the latter into a bottleneck of the computation. Therefore, we aim at implicitly handling permutations while storing the result of the tensor contraction. This goal disinguishes our implementation from early work such as [22] which used level 3 BLAS for both GEMMs and permutation operations.

The memory layout of the input tensor, which in our example contains the DOFs associated with one cell, is prescribed by the `FiniteElementMap` implementation from PDELab. The code generator needs to adapt to this fixed layout. The memory layout of the basis evaluation matrices $A^{(i)}$ can be chosen to be either row- or column-major. In fact, this decision can be included into an autotuning search space in future work. Given these layouts, our goal is to store the permuted tensor T^1 from equation 4.15 with unit stride while performing the tensor contraction from equation 4.14. Additionally, the tensor T^1 should also be stored in column-major format for the next tensor contraction step to take the same form. From these constraints follows an iteration order for the tensor contraction, which results in rows of the basis evaluation matrices to be held in registers, while they are multiplied with slices of X.

We do not automatically inline the implementation of sum factorization kernels into the integration kernels. Instead, we create a separate `loopy` kernel for each sum factorization kernel and add a function call to the integration kernel. We do so for several reasons:

- The compiler is still able to always inline these kernels in our experiments, partly because of the given `always_inline` attribute.
- Readability of the generated code suffers when many kernels are inlined.
- `loopy` kernels of finer granularity allow us to write better transformations for them. This enables the composability of transformation search spaces in a divide and conquer approach.

This is what an example kernel that implements evaluation of $\partial_0 \hat{U} = (D \otimes A \otimes A)X$ for a $\mathcal{Q}_2$ finite element with an 8th order quadrature formula (three basis functions and five quadrature points per direction) looks like:

```
---------------------------------------------------------------
ARGUMENTS:
A: GlobalArg, type:'float64', shape: (5, 3), dim_tags: (N0, N1)
D: GlobalArg, type:'float64', shape: (5, 3), dim_tags: (N0, N1)
buffer0: GlobalArg, type:'float64', shape: ()
buffer1: GlobalArg, type:'float64', shape: ()
dofs: GlobalArg, type:'float64', shape: (3, 3, 3), dim_tags: (N0, N1, N2)
---------------------------------------------------------------
DOMAINS:
```

```
{ [sf_2_2] : 0 <= sf_2_2 <= 4 }
{ [sf_0_1] : 0 <= sf_0_1 <= 4 }
{ [sf_0_2] : 0 <= sf_0_2 <= 4 }
{ [sf_red_0] : 0 <= sf_red_0 <= 2 }
{ [sf_2_0] : 0 <= sf_2_0 <= 2 }
{ [sf_red_1] : 0 <= sf_red_1 <= 2 }
{ [sf_red_2] : 0 <= sf_red_2 <= 2 }
{ [sf_1_2] : 0 <= sf_1_2 <= 4 }
{ [sf_1_0] : 0 <= sf_1_0 <= 2 }
{ [sf_2_1] : 0 <= sf_2_1 <= 4 }
{ [sf_0_0] : 0 <= sf_0_0 <= 4 }
{ [sf_1_1] : 0 <= sf_1_1 <= 2 }
---------------------------------------------------------------
TEMPORARIES:
step0_out: type:'float64', shape: (3, 3, 5), dim_tags: (N0, N1, N2), base:buffer1
step1_in: type:'float64', shape: (3, 3, 5), dim_tags: (N0, N1, N2), base:buffer1
step1_out: type:'float64', shape: (3, 5, 5), dim_tags: (N0, N1, N2), base:buffer0
step2_in: type:'float64', shape: (3, 5, 5), dim_tags: (N0, N1, N2), base:buffer0
step2_out: type:'float64', shape: (5, 5, 5), dim_tags: (N0, N1, N2), base:buffer1
---------------------------------------------------------------
STATEMENTS:
  for sf_0_0, sf_2_0, sf_1_0
    step0_out[sf_1_0, sf_2_0, sf_0_0] = reduce(sum, [sf_red_0],
        D[sf_0_0, sf_red_0] * dofs[sf_red_0, sf_1_0, sf_2_0])
  end sf_0_0, sf_2_0, sf_1_0
  for sf_0_1, sf_2_1, sf_1_1
    step1_out[sf_1_1, sf_2_1, sf_0_1] = reduce(sum, [sf_red_1],
        A[sf_0_1, sf_red_1] * step1_in[sf_red_1, sf_1_1, sf_2_1])
  end sf_0_1, sf_2_1, sf_1_1
  for sf_0_2, sf_2_2, sf_1_2
    step2_out[sf_1_2, sf_2_2, sf_0_2] = reduce(sum, [sf_red_2]
        A[sf_0_2, sf_red_2] * step2_in[sf_red_2, sf_1_2, sf_2_2])
  end sf_0_2, sf_2_2, sf_1_2
---------------------------------------------------------------
```

Some names in this kernel have been changed for the sake of readability, e.g. many names in reality encode polynomial degrees and quadrature orders. The arguments of the kernel are the basis evaluation matrices A and D, which we precompute in the constructor of the `LocalOperator`, the input tensor X of DOFs and two buffers. These buffers are used to store the intermediate tensors and the code switches back and forth between the two buffers: The first contraction writes into the first buffer, the second contraction writes into the second buffer and the third contraction can again use the first buffer, as the result of the first contraction is not needed anymore. We allocate these buffers on the stack in the calling scope, calculating their size at code generation time.

The given temporary variables used for the intermediate tensors have an additional field `'base'`, which indicates a `loopy` mechanism not being used so far. It allows temporary variables to alias other temporary variables while their shape, array axis

implementation tags or underlying data type[3] may vary. We leverage this concept in order to choose the correct buffer for the current contraction step.

This is the resulting generated code of the above example kernel:

```
void sfimpl(double const *__restrict__ dofs, double* buffer0, double* buffer1)
{
  double acc_sf_red_0;
  double acc_sf_red_1;
  double acc_sf_red_2;
  double *step0_out = (double *)buffer1;
  double *step1_in = (double *)buffer1;
  double *step1_out = (double *)buffer0;
  double *step2_in = (double *)buffer0;
  double *step2_out = (double *)buffer1;

  for (int sf_2_0 = 0; sf_2_0 <= 2; ++sf_2_0)
    for (int sf_1_0 = 0; sf_1_0 <= 2; ++sf_1_0)
      for (int sf_0_0 = 0; sf_0_0 <= 4; ++sf_0_0)
      {
        acc_sf_red_0 = 0.0;
        for (int sf_red_0 = 0; sf_red_0 <= 2; ++sf_red_0)
          acc_sf_red_0 = acc_sf_red_0 +
            D[sf_0_0 + 5 * sf_red_0] * dofs[sf_red_0 + 3 * sf_1_0 + 9 * sf_2_0];
        step0_out[sf_1_0 + 3 * sf_2_0 + 9 * sf_0_0] = acc_sf_red_0;
      }
  for (int sf_2_1 = 0; sf_2_1 <= 4; ++sf_2_1)
    for (int sf_1_1 = 0; sf_1_1 <= 2; ++sf_1_1)
      for (int sf_0_1 = 0; sf_0_1 <= 4; ++sf_0_1)
      {
        acc_sf_red_1 = 0.0;
        for (int sf_red_1 = 0; sf_red_1 <= 2; ++sf_red_1)
          acc_sf_red_1 = acc_sf_red_1 +
            A[sf_0_1 + 5 * sf_red_1] * step1_in[sf_red_1 + 3 * sf_1_1 + 9 * sf_2_1];
        step1_out[sf_1_1 + 3 * sf_2_1 + 15 * sf_0_1] = acc_sf_red_1;
      }
  for (int sf_2_2 = 0; sf_2_2 <= 4; ++sf_2_2)
    for (int sf_1_2 = 0; sf_1_2 <= 4; ++sf_1_2)
      for (int sf_0_2 = 0; sf_0_2 <= 4; ++sf_0_2)
      {
        acc_sf_red_2 = 0.0;
        for (int sf_red_2 = 0; sf_red_2 <= 2; ++sf_red_2)
          acc_sf_red_2 = acc_sf_red_2 +
            A[sf_0_2 + 5 * sf_red_2] * step2_in[sf_red_2 + 3 * sf_1_2 + 15 * sf_2_2];
        step2_out[sf_1_2 + 5 * sf_2_2 + 25 * sf_0_2] = acc_sf_red_2;
      }
}
```

In this example, we only show a simple, scalar example. For the SIMD-vectorized kernels that result from the strategies in section 4.2, all arrays in the generated

[3] In this case, the resulting code would violate the strict aliasing rule and should be compiled using `-fno-strict-aliasing`.

code would use a SIMD base type, such as `Vec4d` and additional operations like SIMD broadcasts and horizontal additions would appear in the code.

4.4.2 Cost Model-based Selection of Vectorization Strategies

We will now give details about the algorithm used to select a vectorization strategy. The search space of admissible vectorization strategies described in section 4.2 for a given set of sum factorization kernels is large and it is not known a priori which strategy delivers optimal performance. We mention two scenarios arising in the examples in chapter 5 to illustrate that trade off decisions need to be made:

- For a simple Poisson problem in 3D, $\partial_i u$ are needed, but not the evaluation of u itself. Given a SIMD width of 256 bits, is it better to fuse three kernels and ignore the forth SIMD lane or to apply a (partly) splitting based vectorization strategy? How does the quadrature order affect this?
- For the implementation of the $\mathbb{Q}_k/\mathbb{Q}_{k-1}$ DG scheme for the Stokes equation from section 5.3, the evaluation of pressure cannot be parallelized with any other necessary evaluations. Vectorizing pressure evaluation by splitting may come at the cost of increasing the number of quadrature points for the whole algorithm though. When is it better to not vectorize pressure evaluation?

A cost model based approach is required in order to make optimal vectorization decisions. Such an approach consists of two core components: A function that systematically traverses all vectorization opportunities to find a minimum and an actual cost function. In order to handle the exponential complexity of the traversal of vectorization opportunities, we employ a divide and conquer strategy splitting the optimization problem into several subproblems:

- Starting from the quadrature point tuple $(m_0, \ldots, m_{d-1})$ that was specified by the user or deduced from the problem formulation, we list all possible tuples with increased number of quadrature points, that enable other vectorization strategies. For each of those we find an optimal vectorization strategy and find the minimum among these:

$$\begin{aligned} \underset{q}{\text{minimize}} \quad & \textsc{cost}(\textsc{FixedQPMinimalStrategy}(\mathit{sumfacts}, \mathit{width}, q)) \\ & q \in \left\{ \left(\left\lceil \frac{m_0}{i} \right\rceil \cdot i, \ldots, m_{d-1} \right) \;\middle|\; i = 1, 2, 4, \ldots, w \right\}. \end{aligned} \tag{4.20}$$

  Here, w again denotes the SIMD width.
- When finding an optimal strategy for a given fixed quadrature point tuple, we first divide the given set of sum factorization kernels into smaller subsets, which may potentially be subject to a loop fusion based vectorization approach

Algorithm 4.4: Finding an optimal vectorization strategy: Given a set of sum factorization kernels, the SIMD width and a fixed quadrature point tuple, a vectorization strategy is returned as a mapping of sum factorization kernels to vectorized kernels. The VectorizationStrategies function will be provided by algorithm 4.5.

```
1 function FixedQPMinimalStrategy(sumfacts, width, q)
2     groups ← ∅
3     for all sf ∈ sumfacts do
4         insert sf in groups[ClassifyBounds(sf)]
5     results ← ∅
6     for all groupsumfacts ∈ values(groups) do
7         vecsf ← VectorizationStrategies(groupsumfacts, width, q)
8         insert argmin_{v∈vecsf} COST(v) in results
9     return COMBINE(results)
```

i.e. they share the same loop bounds. Minimal solutions w.r.t. the defined cost function for these subsets are then combined into a full vectorization strategy.

We define a function ClassifyBounds(sf) such that two sum factorization kernels that yield the same value are potentially vectorizable via loop fusion. Likewise, we define a function ClassifyInput(sf) such that two kernels yielding the same result operate on the same input tensor. To wrap up the divide and conquer approach, a function COMBINE that merges the minimal solutions on subsets is used. Algorithm 4.4 illustrates the overall optimization algorithm and algorithm 4.5 shows the algorithm within one of the subsets.

Algorithms 4.4 and 4.5 establish a framework to explore different cost functions. Equation 4.23 provides a heuristic cost model function. It reproduces our practical experiences quite well, but does not take into account specific hardware features beyond the SIMD width. We mainly use it as a drop-in replacement if the autotuning approach from section 4.4.3 is infeasible. A good heuristic cost model function is also beneficial to guide a non-backtracking search space traversal method for the autotuning process. The cost function depends on the following quantities:

- the number of issued FLOP instructions is given as FLOPS(sf). This counts SIMD operations only once.
- a heuristic penalty function ILP(sf) describing the instruction level parallelism potential of a sum factorization kernel depending on the size of the splitting s as used in section 4.2.2, where we observe that loop fusion based vectorization should always be preferred over splitting based vectorization when applicable.

$$\text{ILP}(sf) = 1 + c_0 \log_2(s) \tag{4.21}$$

Algorithm 4.5: For a given set of sum factorization kernels, that are pairwise implementable in parallel, return all vectorization strategies from the pool of implemented methods in section 4.2. The COMBINE function merges the given mappings into one large mapping.

```
1  function VectorizationStrategies(sumfacts, width, q)
2      strategies ← ∅
3      input_groups ← ∅
4      for all sf ∈ sumfacts do
5          insert sf in input_groups[ClassifyInput(sf)]
6      for all num_inputs ∈ {1, 2} do
7          if num_inputs > len(input_groups) then
8              break
9          for all f ∈ {1, 2, 4, ..., w/num_inputs} do
10             if (w/(f ∗ num_inputs)) mod m_0 ≡ 0 then
11                 kernels ← ∅
12                 for all i ∈ {0, ..., num_inputs − 1} do
13                     insert input_groups[i][0 : f − 1] in kernels
14                 s ← ∅
15                 for all sf ∈ kernels do
16                     s[sf] = VectorizedSumfactKernel(kernels)
17                 for all other ∈VectorizationStrategies(sumfacts−kernels,
   width, q) do
18                     extend(strategies,combine(other, s))
19     return strategies
```

- a heuristic penalty function LOADS(sf) for the necessary load instructions. This depends on the number of input coefficients p used for loop fusion in the kernel.

$$\text{LOADS}(sf) = 1 + c_1 \log_2(p) \tag{4.22}$$

The resulting cost function is the product of these terms:

$$\text{COST}(sf) = \text{FLOPS}(sf) \cdot \text{ILP}(sf) \cdot \text{LOADS}(sf) \tag{4.23}$$

In practice, we have chosen $c_0 = 0.5$ and $c_1 = 0.25$ and achieved good results. We will validate this choice in more detail in section 5.2. While it is definitely useful to have a very cheap cost model at hand in order to quickly generate code or to discriminate slow variants, we will use the established minimization procedure for an autotuning approach in the following section.

Although necessary in order to handle the complexity of the vectorization strategy search space, the divide and conquer approach mentioned above bears an intrinsic problem. When performing the minimization over the set of possible quadrature

point tuples at the outermost divide and conquer level in equation 4.20, the COST function cannot account for any costs that an increased number of quadature points causes outside of the sum factorization kernel. In our case, this occurs in the quadrature loop, where the additional work depends on the number of floating point operations per quadrature point, which is prescribed by the PDE problem. We fix this issue by introducing a penalized cost function PCOST that takes into account the total number of FLOPs. We obtain that number *ops* from introspection of the ASTs of the quadrature loop:

$$\text{PCOST}(sf, q, q^{min}) = \frac{\text{FLOPS}(sf) + ops \cdot \prod_i q_i}{\text{FLOPS}(sf) + ops \cdot \prod_i q_i^{min}} \text{COST}(sf) \tag{4.24}$$

Here, q^{min} denotes the quadrature tuple that is the minimum requirement for the problem to be correctly solved. Another cost not correctly reflected by COST is the overhead of the transposition operations described in figure 4.2. This is assuming that these can be effectively hidden behind the floating point workload of the quadrature loop, as we described in section 4.3. This assumption held true for our experiments in section 5.2, but it might fail for PDE problems that do not require much floating point operations in the quadrature loop or for future architectures with even wider SIMD widths.

4.4.3 Autotuning

Autotuning is a technique in computer science that is getting more and more important in HPC due to the complexity of modern architectures [120]. The performance of different program realizations often varies drastically and a priori performance models cannot be found. In these cases, autotuning can be used to evaluate program variants from a search space in a preprocessing step to perform a minimization of needed ressources (e.g. runtime). This preprocessing step perfectly integrates into a code generation tool chain. Implementation of GEMMs on modern architectures has been subject to autotuning by many authors, e.g. [2], [45]. Software frameworks that implement generic tuning algorithms on top of user-provided search spaces have been introduced, e.g. Opentuner [9] or CLTune [92].

Autotuning requires the following key components: Definition of a search space that describes available program variants, a way of traversing this search space and a facility to generate, compile, execute and measure program variants. In order to handle the complexity of the search space, divide and conquer approaches should be used wherever possible. In the context of complex applications, such as simulations with PDEs, this starts with identifying the correct granularity level for autotuning. In our case, the natural choice of granularity level is given by the sum factorization kernels in the problem. The vectorization strategies provided in section 4.3 and the traversal algorithm from section 4.4 implement a custom search space which can be explored by means of an autotuner. We currently use backtracking to traverse this search space.

We implement the following toolchain for generation, compilation, execution and measurement of program variants:

- We implement an additional code path in the source code generator that generates stand-alone benchmarking code for a `loopy` kernel. These kernels may have a smaller scope, such as one individual sum factorization kernel implementation. Input and output arrays of such kernels are mocked within the benchmark program.
- The CMake integration of our code generator described in section 3.4.2 is used to determine compiler paths and flags. To this end, a dummy target in the dune-codegen project is configured with all the necessary flags for autotuning. The code generator retrieves these flags by accessing CMake-internal caches. Optionally, a build environment-specific wrapper is used. This is e.g. helpful if the compilation node uses a module system to access compilers.
- Execution of the benchmark programs is again subject to some build environment-specific tweaks. We therefore allow an execution wrapper script to be passed to the code generator. We have used this in the past to offload benchmark execution onto compute nodes via a scheduler in a cluster environment. Compilation and execution results are cached on disk in order to save resources.
- Measurement of benchmark execution is done through the Google benchmark library [49]. The above mentioned generated stand-alone code already contains the necessary benchmarking code. The benchmark library controls repeated execution of the kernel until a good runtime estimate is available and writes out the result as a JavaScript Object Notation (JSON) file.

The above toolchain is hooked into the code generation process through a cost function, which generates code, compiles it, runs it and reads the written JSON file. The minimization procedure from algorithm 4.5 then finds the code variant that minimizes the total execution time. This cost function is selected through the form compiler option `vectorization_strategy=autotune`.

This autotune approach delivers very good results and we generally use it for our performance measurements. Depending on how large the search space is, code generation may take a substantial amount of time though. While this is not important for an HPC application to be run at large scale, it is unfeasible during development cycle. The necessity to penalize cost functions in the divide and conquer setting described in section 4.4 also applies to autotuned code.

In future work, the use of autotuning in the code generator can be greatly extended. The `loopy` IR that we use to represent our loop nests allows us to define search spaces for different program variants of a given (vectorized) sum factorization kernel. This search space is formulated in terms of `loopy` transformations. The following techniques are a promising starting point for optimization of the sum factorization kernel code realization:

- *Loop reordering:* Tensor contractions of the form $A_{ij} = \sum_k B_{ik}C_{kj}$, although typically expressed in i-j-k order - can be nested in many different ways, which can be enforced by `loopy` transformations.
- *Loop tiling:* Tensor contraction loops can be tiled such that the working set size is optimized for the L1 cache of the given architecture. Especially for very high polynomial order, this is a promising approach.
- The *memory layout* of basis evaluation matrices could be changed between row-major and column-major format.
- *Loop unrolling* is an optimization expressable as a `loopy` transformation that is quite important for maximum performance. However, in our experience, the compiler does sufficiently unroll our sum factorization kernel implementation even without explicit guidance.

4.4.4 Geometry Evaluations

Optimal implementations of sum factorization kernels for facet integrals depend on the embedding of the reference facet into the reference element of the neighboring cell, as it allows the reordering of tensor contractions in order to minimize the total number of executed FLOPs. We provide one implementation per facet of the reference element for boundary integrals and one implementation per combination of facets of the two adjacent cells for interior facets. The number of actually releveant combinations in the skeleton case is reduced drastically on structured grids though. Our generated code determines the facet embedding and dispatches to the correct implementation. Having this facet-specific implementations in place also has a beneficial impact on the implementation of geometry evaluations. We therefore provide dedicated mixin classes for sum factorized code which extend the choice of mixins given in figure 3.13. The mixin classes identified by `'sumfact_axiparallel'` and `'sumfact_equidistant'` inherit from their base version and implement the following additional optimization on a facet whose normal vector is parallel to $\boldsymbol{e}_j$:

$$\boldsymbol{n}_i = \pm\delta_{ij} \tag{4.25}$$

With the code generation time unrolling of reduction indices described in section 3.3.2 being applied, this completely removes the `FacetNormal` node from the generated code. Also, the zero entries of the normal vector are propagated through the AST and guarantee a FLOP-minimal code variant.

Semistructured Geometry Mapping

We will now present a custom domain-specific geometry mapping for simulations of soil physics applications. We do so to demonstrate how the composition pattern of the main UFL to `loopy` visitor allows users to provide custom implementations tailored to application requirements. We assume the grid to be axiparallel and equidistant within the $x-y$ plane, but allow z-coordinates to vary non-equidistantly.

We do so by prescribing elevation functions $z^t(x, y)$ and $z^b(x, y)$ at top and bottom and use an equidistant meshing along each vertical line. In DUNE such a grid can be realized by the `GeometryGrid` class, which is a *meta grid*. Meta grids do wrap the implementation of a *host grid* but may provide additional functionality. In the case of `GeometryGrid`, the vertices of the host grid are remapped with a custom multilinear map μ_T^*. For our semi-structured grid, a structured grid is transformed with the following transformation μ_T^*:

$$\mu_T^*(\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}) = \begin{pmatrix} x' \\ y' \\ z^b(x', y') + z'(z^t(x', y') - z^b(x', y')) \end{pmatrix}. \tag{4.26}$$

The function μ_T^* assumes the host grid to be using the unit interval as its extension in z direction. The geometry evaluations provided by the `GeometryGrid` instance would have the same computational complexity as a fully unstructured mesh though, because it does not grasp the special structure in the x-y plane. We therefore only use this grid for the purposes of visualization and perform the evaluation of any semistructured geometric quantity in the code generator.

The geometry mapping μ_T^* exhibits some special structure that can be exploited at code generation time to reduce the computational complexity of the geometry evaluation:

- The side facets of the transformed hexahedra are parallel to the $x-z$ or $y-z$ plane. This results in the normal vector being axiparallel: $\boldsymbol{n} = \pm\boldsymbol{e}_x$ (or $\boldsymbol{e}_y$). In the sum factorization approach we are using different implementations for all facets of the reference cube which greatly simplifies two thirds of our integrals over interior facets.
- The jacobian of the geometry mapping μ_T^* from equation 4.26 exhibits a special structure:

$$\nabla\mu_T^* = \begin{bmatrix} h_x & 0 & 0 \\ 0 & h_y & 0 \\ \delta_0 & \delta_1 & \delta_2 \end{bmatrix} \tag{4.27}$$

  with $\delta_i := \partial_i \mu_{T,z}^*((x', y', z')^T)$. The jacobian matrix is a scaled version of a Frobenius matrix and as such, its inverse shares the original jacobian's sparsity pattern:

$$(\nabla\mu_T^*)^{-1} = \begin{bmatrix} h_x^{-1} & 0 & 0 \\ 0 & h_y^{-1} & 0 \\ -\delta_0\delta_2^{-1}h_x^{-1} & -\delta_1\delta_2^{-1}h_y^{-1} & \delta_2^{-1} \end{bmatrix} \tag{4.28}$$

  Due to its way of unrolling operations on small tensors, the code generator can exploit this sparsity pattern and reduce the complexity of the resulting expression. Furthermore, the determinant of the inverted jacobian is as simple as $(h_x h_y \delta_2)^{-1}$.

- Evaluation of the above quantities δ_i can be implemented using sum factorization techniques. In order to do so, we again calculate the quantity at all quadrature points at the same time. The basis functions are chosen as $\mathcal{Q}_1$ Lagrange functions and the input tensor is given by the tensor that collects the z-coordinates of all the vertices of the cube. Within the code generator, these sum factorization kernels are handled in the same way as the ones arising from finite element function evaluation e.g. SIMD vectorization strategies are applied.

The evaluation of δ_i through a sum factorization kernel can also be done for all entries of the jacobian if the geometry mapping is multilinear and does not exhibit any special structure. Such geometry mappings have also been implemented into our code generator though we do not show any details here. Compared to fully multilinear geometry mappings, semistructured geometries provide a great tradeoff between computational efficiency and flexibility w.r.t. the computational domain.

4.4.5 Block Preconditioners

The matrix-free solution procedure described in section 2.7 requires the implementation of building blocks beyond the `LocalOperator` for the given PDE. One such building block is an operator that applies the block-diagonal part of the DG operator in a potentially matrix-free fashion. For the overall complexity and performance to be preserved, the same performance optimization strategies need to be applied for these operations. In [89], this was realized by adding compile-time conditionals to the local operator implementation that restricted the full operator to the block diagonal contributions. In a code generation procedure, we can instead formulate a form transformation that restricts the given form to the block diagonal part and then generates code for this modified input. This way, the implementations of the preconditioner and the standard code generator are separated perfectly.

Given the jacobian form, the form transformation to restrict a given problem to the block diagonal works as follows: Any occurence of the pattern

```
PositiveRestricted(Argument(...))
```

is mapped to `Zero`, while all other expressions are not altered. Note that this is only true as we are generating local operators which are meant for two-sided assembly, meaning that we expect each intersection $F \in \mathcal{F}_h$ to be visited twice - once from each neighboring cell. Normally, in PDELab each intersection is only visited once and we would be required to also take into account those terms where both arguments are restricted to the outside cell in the form transformation. However, in the solution procedure in section 2.7, we are required to calculate the diagonal jacobian block associated with a cell locally, i.e. without iterating over all the grid.

Another quantity that is required in the matrix-free solver is the diagonal of the diagonal blocks. It is used to precondition the matrix-free inversion of a diagonal

block. Our goal is to assemble the diagonal directly into a vector and never assemble the full diagonal block. Such an assembly procedure would preserve the optimal $\mathcal{O}(k^{d+1})$ complexity of the overall algorithm. Given the jacobian form $j(u; v, w)$, we study how $d(u; v^2) := j(u; v, v)$ can be implemented directly. As d is linear in v^2, we are interested in how to efficiently evaluate these squared basis functions. We do so by reordering the one dimensional functions such that we formulate the overall evaluation in terms of products of these:

$$(\phi(\boldsymbol{x}))^2 = \left(\prod_{k=0}^{d-1} \phi_k^{1D}(x_k)\right)^2 = \prod_{k=0}^{d-1} (\phi_k^{1D}(x_k))^2. \tag{4.29}$$

This gives rise to the following sum factorized assembly of $d(u; v^2)$ from the tensor $\hat{R}$ which describes the evaluations at all quadarature points which needs to be multiplied with v^2:

$$\hat{J} = \left(A^T \circ A^T \otimes A^T \circ A^T \otimes A^T \circ A^T\right) \hat{R} \tag{4.30}$$

Here, $\circ$ denotes the element-wise product of two matrices. For the sake of simplicity, we omitted any gradients of v in the above description. Taking these into account requires us to consider a lot of combinations arising from test and trial functions having different partial derivatives applied. Where in residual assembly, we had up to $d+1$ sum factorization kernels in this step (multiplication with the test function and the d components of its gradient), we need to consider $(d+1)^2$ combinations here. Fortunately, many of these can be discarded due the multiplication of basis evaluation matrices being commutative. While implementing this diagonal operator manually might be a very tedious procedure, the code generation approach allows us to do this in a generic, problem-independent fashion.

CHAPTER 5

Performance Experiments

We have introduced an HPC-enabled code generation toolchain in section 3 which allows us to generate code that uses sum factorization to exploit the tensor product structure of finite elements. In section 2.7 we highlighted that such an implementation is especially valuable when performing matrix-free evaluation of an operator for DG elements. Together with the SIMD vectorization strategies from section 4.2, an implementation that achieves a substantial amount of the machine's theoretical floating point peak performance is possible. We will now back this claim with performance measurements of our generated code. In order to give the reader full insight into how these numbers were measured, we will first describe the benchmark setup in detail in section 5.1. We will then move on to perform measurements with the diffusion-reaction problem which we investigated in a lot of places throughout this thesis. Sections 5.3 and 5.4 will apply the techniques to more complex systems of PDEs: the Stokes equations and Maxwell's equations. We will seize the opportunity of introducing these PDEs to also provide the full implementation of those examples in the UFL DSL, illustrating the desirability of the code generation approach for the user.

5.1 Benchmark Setup

It is our intent to be as transparent as possible about how the performance numbers from sections 5.2 to 5.4 were produced. We will therefore give all necessary details about the methodology in this section. This spans explanation of performance metrics, measurement of floating point operations and runtimes, hardware configurations, compilers and how benchmark programs are executed. This setup description is partly taken from [63].

We use the following two performance metrics in this work:

- FLOPs per second expressed in GFLOPs/s and often given as a percentage of the machine's maximum floating point performance. We will often refer to this as *machine utilization*.
- DOFs per second processed during a full operator application. Note that we prefer this measure over its inverse (time per DOF). We will often call this *throughput* in the following.

Good results on the latter are always more important from the application point of view, as it gives an accurate measure of how fast a real problem can be solved. However, the former is still an interesting measure that allows reasoning about how good a code is suited for a given hardware. As implementations that are not minimal w.r.t. the total number of FLOPs executed do artificially increase the utilization, the throughput measure becomes even more important in those case.

In order to accurately measure the FLOPs per second, the number of performed FLOPs needs to be measured exactly. This can in principle be done by reading special registers provided in Intel architectures. However, speculative branch execution and other sorts of ILP effects make this number differ from the minimal number of FLOPs required by the mathematical problem. We therefore pursue a different approach exploiting the fact that we are generating code for dune-pdelab, which uses C++ templates to the extent that we can replace the underlying floating point type throughout all our simulation workflow. Instead of using `double`, we use a custom type templated to `double`, which has overloads for all arithmetic operations that increase a global counter and forward the operation to the underlying template type. This floating point type is provided in a separate module dune-opcounter [66] which is also usable outside of the DUNE context. In order to also count FLOPs executed with SIMD types from the vector class library from section 3.2.3, we provide a vector type that mimics its interface but forwards all operations to the underlying scalar counter type. This counting of course introduces a non-negligible performance overhead. We therefore compile different executables from the same source for operation counting and time measurement. The approach bears another disadvantage in that the compiler cannot perform CSE on the operation counting executable. We therefore need to assume that our code is minimal in terms of executed FLOPs, which is quite often either a reasonable assumption or the overall work is dominated by the FLOP-minimal work in sum factorization kernels.

Apart from counting FLOPs, accurate time measurements are needed. We instrument our code with C macros to start and stop timers using the Time Stamp Counter (TSC) registers. The performance overhead of starting and stopping a timer is measured at runtime and the measurements are calibrated accordingly. Processes are pinned to physical cores to prevent spoiled TSC measurements and unnecessary context switches. To gain further insight into the performance bottlenecks of our implementation, we measure time and FLOP rates at different levels of granularity:

Full operator application, cell-local integration kernel and individual steps of algorithm 2.3. For all these granularity levels, separate executables are compiled to assure that no measurement is tampered by additional measurements taken within the measuring interval. In our experience, the finest granularity level is subject to measurement artifacts at very low polynomial degrees, as the total number of FLOPs within the measurement interval becomes small. We therefore take these measurements for $\mathcal{Q}_1$ elements with a grain of salt. All time measurements are repeated at least ten times and the minimum of these runtimes is reported.

We study the node-level performance of our generated code on two Intel micro architectures: Haswell and Skylake. Details about these CPUs can be found in appendix A. The theoretical peak performance is given as the product of the processor frequency, the number of SIMD lanes for double precision values, the number of ports capable of executing arithmetic instructions (here: two in both cases) and a factor of two to account for FMA. A survey of issues in determining a machine's theoretical peak performance is given in [36]. It is worth noting, that the additional frequency reduction moving to AVX-512 on Intel Skylake results in the theoretical throughput increase moving from AVX2 to AVX-512 being as low as 1.83, instead of the naively expected factor of 2. We are always seeking to saturate the full machine with our computation. Otherwise, some processors may have privileged access to resources such as memory controllers and tamper results. We do so by doing MPI parallel computations with one rank per core and an identical workload size on each of these ranks.

We choose the problem size of the benchmark program such that one vector of DOFs exceeds the level 3 cache of the machine. While this may not be a realistic setting when doing strong scaling of simulation codes, it gives a good worst case estimate of our code's node level performance. The time for communication of overlap data via MPI is not included in our measurement, as the task of hiding that communication behind computation is not the subject of this work. Measurements are presented for volume and skeleton integrals only, assuming a periodic grid. Boundary integrals can of course be generated as well, but their impact on the overall performance is negligible. Unless otherwise mentioned, we use the GNU compiler in version 8.1 for our results.

5.2 Diffusion-Reaction Equation

The diffusion-reaction equation was used as an example problem throughout this thesis. We will now move to provide detailed performance results for its implementation. Before showing results for the Intel Haswell and Skylake architectures we will recap the mathematical formulation of the DG discretization and provide its full implementation in the UFL DSL.

Mathematical Formulation

We recall the SWIP DG formulation for the diffusion-reaction equation from equation 2.13 and recast it into residual formulation:

$$\begin{aligned} r_h(u_h, v_h) = &\sum_{T\in\mathcal{T}_h}\int_T k(\boldsymbol{x})\nabla u\cdot\nabla v + c(\boldsymbol{x})uv - fv\ dx \\ &- \sum_{F\in\mathcal{F}_h}\int_F (\{\boldsymbol{k}(\boldsymbol{x})\nabla u\}_\omega, \boldsymbol{n})[\![v]\!] + [\![u]\!](\{\boldsymbol{k}(\boldsymbol{x})\nabla v\}_\omega, \boldsymbol{n}) - \gamma_F[\![u]\!][\![v]\!]\ ds \\ &- \sum_{F\in\mathcal{B}_h\subseteq\Gamma_D}\int_F (\boldsymbol{k}(\boldsymbol{x})\nabla u, \boldsymbol{n})v + (u-g)(\boldsymbol{k}(\boldsymbol{x}\nabla v, \boldsymbol{n}) - \gamma_F(u-g)v\ ds \end{aligned} \tag{5.1}$$

where the penalty parameter γ_F is chosen exactly as in equation 2.15. In this section, we will fix the physical parameters in equation 5.1 to the following simple choices:

$$\boldsymbol{k}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{x}^T + \boldsymbol{I} \tag{5.2}$$
$$c(\boldsymbol{x}) = 10 \tag{5.3}$$
$$f(\boldsymbol{x}) = -6 \tag{5.4}$$
$$g(\boldsymbol{x}) = \|\boldsymbol{x}\|_2^2 \tag{5.5}$$

These parameters are chosen such that $u(\boldsymbol{x}) = \|\boldsymbol{x}\|_2^2$ solves the original PDE. This procedure is called *method of manufactured solution* and allows code verificiation for PDE codes [105].

UFL Implementation

We will now provide the full UFL implementation of the diffusion-reaction problem. We do so both in order to exemplify the usage of the DSL and to provide full details about the given benchmark in case the reader wants to compare performance results against different implementations.

We start off with defining some basic geometric quantities:

```
dim = 3
cell = hexahedron
x = SpatialCoordinate(cell)
n = FacetNormal(cell)('+')
```

Given these, we can implement the physical parameters from equations 5.2 to 5.5:

```
g = x[0]*x[0] + x[1]*x[1] + x[2]*x[2]
I = Identity(3)
k = as_matrix([[x[i]*x[j] + I[i,j] for j in range(3)] for i in range(3)])
f = -6.
c = 10.
```

We continue to define the used finite element space and the test and trial functions taken from it. The polynomial degree is not fixed here, but it can be set to any non-zero value.

```
V = FiniteElement("DG", cell, degree)
u = TrialFunction(V)
v = TestFunction(V)
```

The penalty parameter for the DG method is implemented exactly as defined in equation 2.15:

```
alpha = 3.0
h_ext = CellVolume(cell) / FacetArea(cell)
gamma_ext = (alpha * degree * (degree + dim - 1)) / h_ext
h = Min(CellVolume(cell)('+'), CellVolume(cell)('-')) / FacetArea(cell)
gamma_int = (alpha * degree * (degree + dim - 1)) / h
theta = -1.0
```

Finally, the residual form is written down in UFL notation which resembles the mathematical notation from equation 5.1. Details about the used UFL functionality are provided in section 3.2.1:

```
r = (inner(k*grad(u), grad(v)) + (c*u-f)*v)*dx \
  - inner(n, k*avg(grad(u)))*jump(v)*dS \
  + gamma_int*jump(u)*jump(v)*dS \
  + theta*jump(u)*inner(k*avg(grad(v)), n)*dS \
  - inner(n, k*grad(u))*v*ds \
  + gamma_ext*u*v*ds \
  + theta*u*inner(k*grad(v), n)*ds \
  - theta*g*inner(k*grad(v), n)*ds \
  - gamma_ext*g*v*ds
```

Intel Haswell Results

We will now turn to performance results for the diffusion-reaction equation on the Intel Haswell architecture. The equation is solved as given in equation 5.1 on a structured equidistant mesh with varying polynomial degree. We used autotuning to select a suitable vectorization strategy, though minimization of the cost model function from equation 4.23 yields the same results for the diffusion-reaction equation. Figure 5.1 shows the performance measurements for a full operator application, as well as the volume and skeleton integrals individually. We make several obervations:

- Given a polynomial degree $k > 2$, the total machine utilization is at roughly 40%.
- Volume integrals are capable of utilizing the machine even better peaking at 60% of the machine's floating point capabilities for $k = 3$. Skeleton integrals

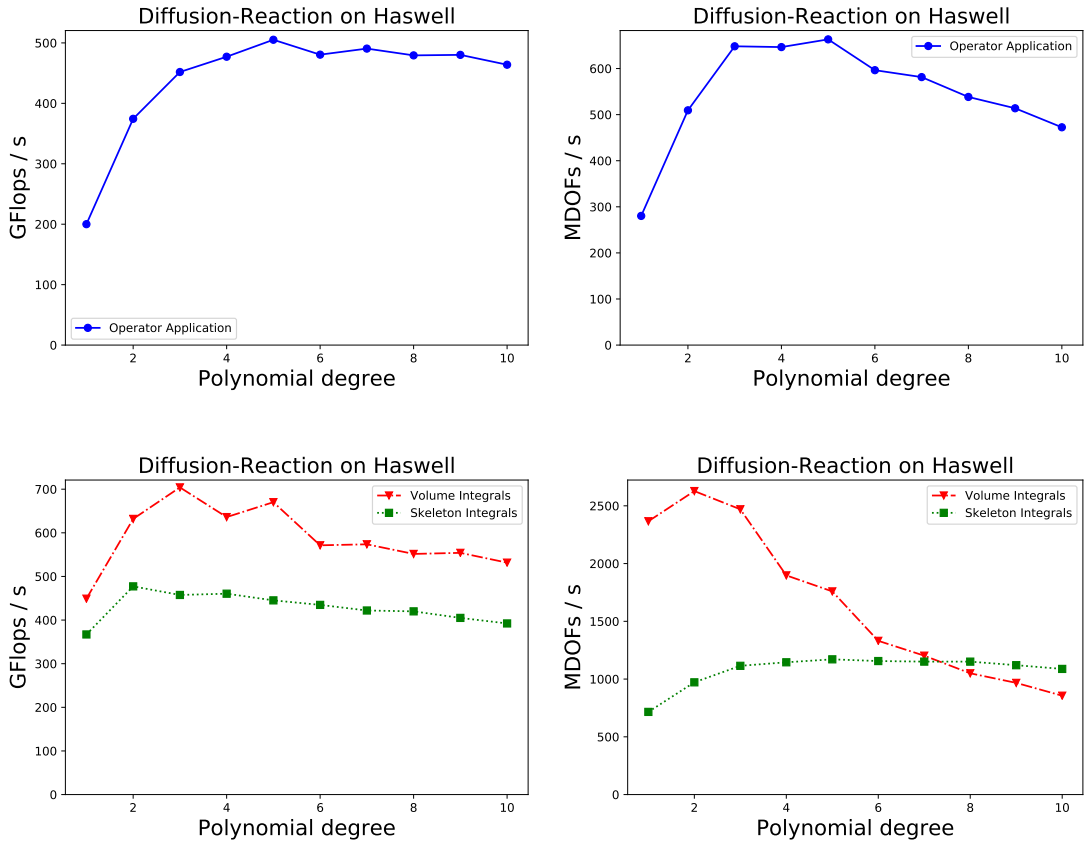


Figure 5.1: Performance of the diffusion-reaction equation on Intel Haswell: The upper row of plots show the machine utilization and the DOF throughput for a full application of the DG operator. The lower row shows the same numbers with measurements restricted to the integration kernels for volume and skeleton integrals. We show performance numbers for the full operator application in a separate plot in order to have an insightful scaling in the throughput plot. Our implementation achieves 40% machine utilization for medium to high polynomial degrees.

on the other hand only run at roughly 35%. This gap can be explained by the much higher flop-per-byte-ratio in volume integrals and the amount of ILP between the two floating point ports that is enabled by these additional FLOPs.

- The vectorization strategy that enables these performance numbers is a fully fusion-based one. A comparison with other strategies will be given in figure 5.3.
- The maximum total DOF throughput of a full operator application is roughly 650 MDOFs/s. The thoughput of skeleton integrals is independent of the polynomial degree, because the FLOPs per DOF ratio is constant. For volume integrals however, the number of FLOPs per DOF increases with the polynomial degree, making them increasingly dominant for higher polynomial degree.

This property holds independently of the PDE as it results from a surface to volume ratio effect.

- For $k = 1$, performance is drastically reduced. Additionally, performance numbers from the two shown granularity level do not add up in a coherent way. We suspect that the very short time span of the measurement interval yields significant errors in our measurements. This might be related to the measuring instruction introducing a memory barrier which has more impact on the measurement at low polynomial orders. With the shown technology being geared towards high order methods, we did not attach importance to the $\mathcal{Q}_1$ case though.

In figure 5.2 we have an even more detailed look at how individual steps from algorithm 2.3 perform. Step one performs a sequence of tensor contractions with a SIMD broadcast of the input tensor. This operation is executed at up to 60% of machine peak. In contrast, step three only runs at 30% (on skeleton integrals) to 40% (on volume integrals) of the machine peak. The main difference between these is the necessity of horizontal addition of the tensor contraction result (instead of the SIMD broadcast in step one). This illustrates well, how much of a bottleneck the horizontal addition may be. Despite its very high machine utilization, the quadrature loop step does not contribute much to the overall performance, because of the small number of total FLOPs executed in it.

Comparison of Vectorization Strategies

As noted in the previous section, the best performing code variants from our vectorization search space were purely fusion based ones. We will now turn to investigate how other strategies perform. Figure 5.3 shows a comparison of three vectorization strategies:

- The purely fusion based one as described in section 4.2.1, which fuses the sum factorization kernels for the evaluation of u and ∇u into one SIMD-vectorized sum factorization kernel.
- The splitting strategy described in section 4.2.2. This strategy vectorizes each sum factorization kernel individually but requires a divisibility constraint of 4 on q_0. If this constraint is not satisfied by the specified minimal number of quadrature points, q_0 is artificially increased.
- A hybrid strategy following the ideas of section 4.2.3. Here, this strategy fuses pairs of sum factorization kernels and introduces a relaxed divisibility constraint of 2 on q_0.

We observe that strategies based on kernel fusion perform scrictly better in terms of DOFs throughput. This can be explained with the high total number of FMA chains needed to saturate the two floating point ports of the processor: The pipeline depth

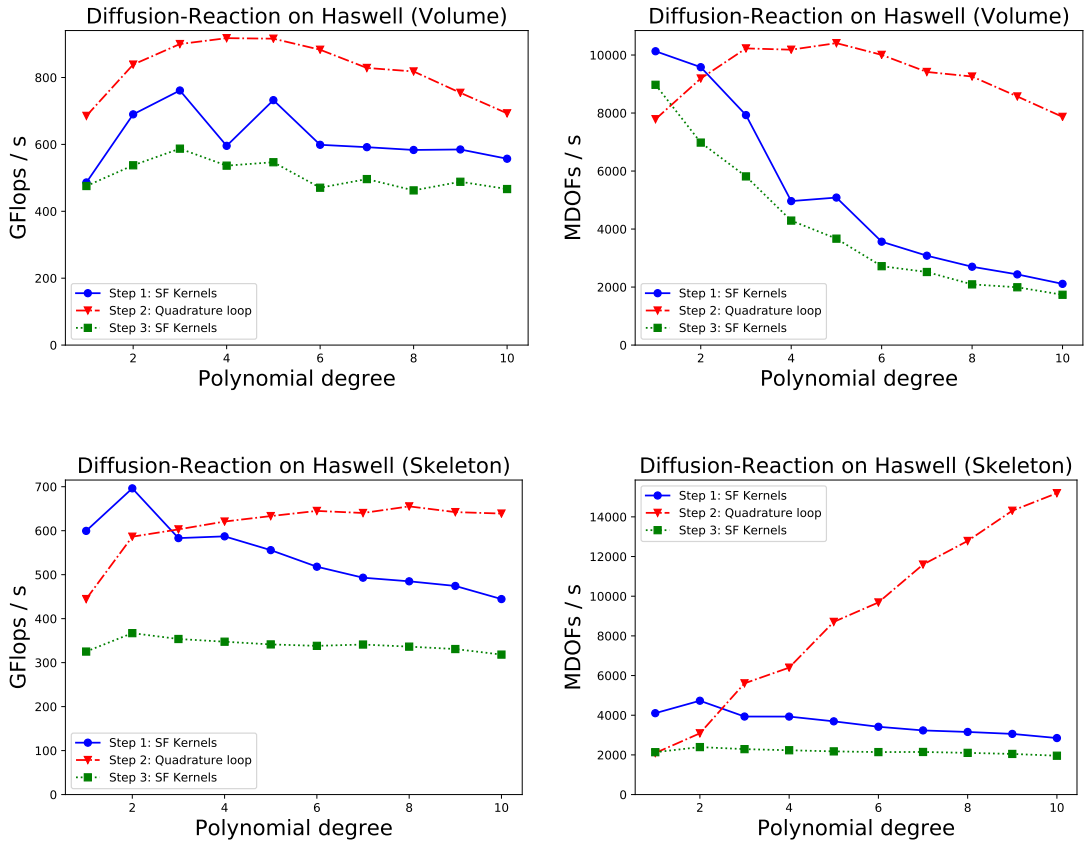


Figure 5.2: Fine grained performance measurements of diffusion-reaction integration kernels on Intel Haswell: Machine utilization and throughput for the three main steps from algorithm 2.3 are shown for volume and skeleton integrals.

of the processor requires long, unrolled chains of FMA operations in order to run at full utilization. Applying a fusion strategy effectively increases the number of such chains in a sum factorization kernel implementation by a factor of the SIMD width. This necessity of having a sufficient amount of independent FMA chains is also one reason that makes the difference in throughput so much more pronounced for skeleton integrals. Another reason was already described in section 4.2.2: FLOP-minimal implementations of sum factorization kernels reorder the tensor contraction sequence such that the contraction associated to the normal direction is executed first. However, this contraction cannot be vectorized with a splitting strategy, as there is only one quadrature point for the normal direction. This results in the first tensor contraction being executed in a non-vectorized way. The mismatch between the DOFs throughput and the machine utilization measurements can be explained with an increased number of quadrature points: In order to make the fully splitting strategy work for $k = 1$, the number of quadrature points is doubled ($q_0 = 4$ instead of $q_0 = 2$). While this increased workload achieves a higher machine utilization, those additional FLOPs are not necessary in a fusion strategy that achieves a higher

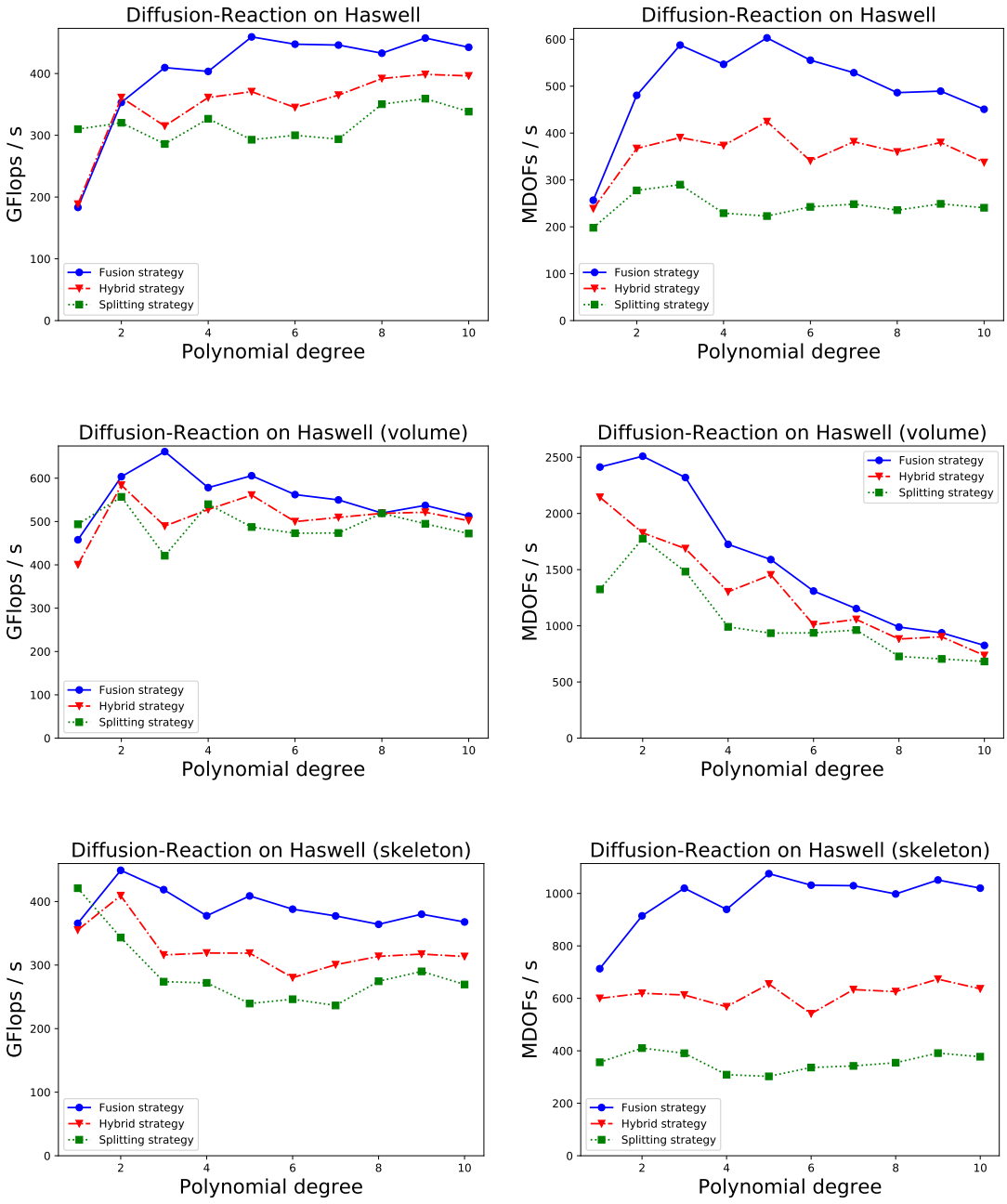


Figure 5.3: Performance comparison of multiple SIMD vectorization strategies for the diffusion-reaction equation on Intel Haswell: The upper rows shows the measurements for a full application of the DG operator, the lower rows show volume and skeleton integrals. The fusion strategy was described in section 4.2.1 and fuses the evaluations of u and ∇u. The splitting strategy from section 4.2.2 vectorizes each sum factorization kernel individually potentially requiring an increase of the quadrature points to satisfy the associated divisibility constraint. The hybrid strategy from section 4.2.3 fuses two quantities and introduces a more moderate divisibility constraint of 2 on q_0.

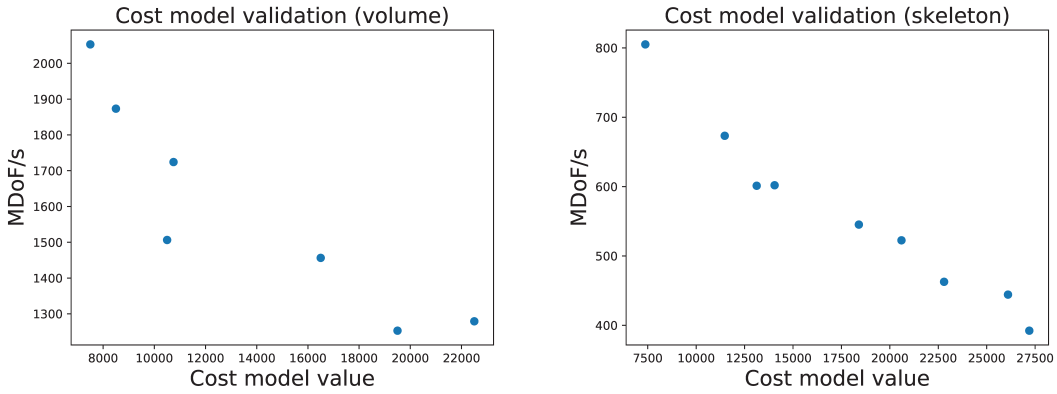


Figure 5.4: Validation of the cost model function from equation 4.23 given the diffusion-reaction problem: Each data point represents one vectorization strategy. The cost model function correlates with the measured overall performance and most importantly captures the maximum performance correctly.

DOFs throughput. For this reason, only the combination of these two measures gives good insight into our kernels' performance.

Validation of the Cost Model

The cost model function from equation 4.23 is a very cheap alternative to the autotuning procedure. However, the construction of the cost function is purely heuristic. Figure 5.4 shows how the cost function value and the measured performance correlate. The plots are obtained by restricting the DG formulation to one integral type and adding one data point per vectorization strategy for that integral. Most importantly, the cost function minimum and the maximum measured performance agree.

Comparison of Horizontal Addition Implementations

In section 4.3 we studied performance critical SIMD operations and provided several implementations of horizontal addition. Figure 5.5 compares our improved implementation from figure 4.5 with the default implementation of the vector class library from figure 4.4. In this measurement, we restrict ourselves to the volume integral of the diffusion-reaction equation, which is vectorized with a fusion-based strategy. The figure shows the speedup of our implementation for both the full volume integral and for step three of algorithm 2.3 in isolation. We observe a consistent speedup across all polynomial degrees with the total gain being around 2%. The fact that this overall gain is achieved by merely avoiding the `vhaddpd` instruction in one place illustrates how delicate SIMD performance bottlenecks are.

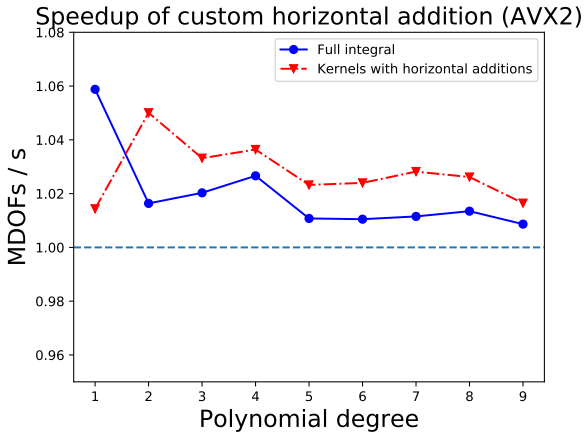


Figure 5.5: Speedup of the custom implementation of AVX2 horizontal addition from figure 4.5 compared to the vector class library implemenetation from figure 4.4. Both the speedup for the full evaluation of the volume integral and for step three of algorithm 2.3 are shown.

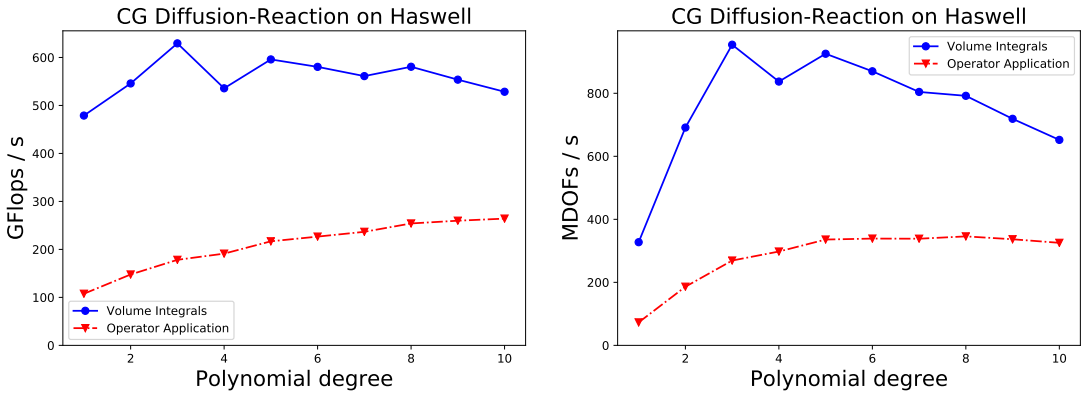


Figure 5.6: Performance of Continuous Galerkin (CG) implementation of the diffusion-reaction equation using sum factorization. As expected, the volume kernel performs similar to its DG counterpart from figure 5.1, but the performance is lost in the data gathering and scattering.

Sum Factorized Continuous Finite Elements

We motivated our use of sum factorization with DG methods. However, local CG bases exhibit the same tensor product structure and can be evaluated with sum factorization. A CG formulation of the diffusion-reaction equation was given in equation 2.4. In contrast to DG, the CG formulation consists of volume integrals only. However, the local integration kernels cannot operate directly on the global data structures. Instead, PDELab provides dense containers for DOFs and residuals which are gathered from and scatter to global memory in a pre- and postprocessing step. Figure 5.6 shows the performance of these local integration kernels which is on par with the DG volume integrals from figure 5.1. The embedding of this local integration into a global operator application does seriously hamper the performance though, as the costly data movement operations cannot be hidden. As a result, the DOF thoughput of the CG implementation is drastically reduced. It is lower than the overall DG DOF throughput although it performs much fewer FLOPs. A 20% machine utilization is still a competitive value for CG methods though. Our CG implementation is currently limited to structured grids, as continuous $\mathcal{Q}_k$ finite elements on hexahedra with $k > 2$ require special care [3].

Single Precision Results

For the performance numbers presented so far, a fusion based strategy was always possible. We will now present measurements of single precision computations for the diffusion-reaction equation on the Intel Haswell architecture. With a SIMD widths of 8 lanes, volume integrals cannot be vectorized by fusion due to a lack of suitable quantities. Consequently, a hybrid splitting strategy with a divisibility constraint of 2 for the number of quadrature points in x-direction is used. In the skeleton case, fusion is possible when allowing two distinct input tensors as there appear exactly eight quantities: u^-, u^+, ∇u^+ and ∇u^-. Figure 5.7 shows the resulting performance and figure 5.8 shows the achieved speed-up compared to the double precision case from figure 5.1.

The theoretical maximum utilization increase is a factor of 2 when moving from double to single precision operations, although the increase in DOFs throughput might be lower due to an artificially increased number of FLOPs. However, in practice we only achieve a speed-up of 1.8 for volume integrals at high polynomial degrees and no speed-up at all at low polynomial degrees and for skeleton integrals. One reason for this lack of performance is that the complexity of critical SIMD operations like horizontal addition and AoS to SoA transformations is increased. Furthermore, we have not provided custom implementations of horizontal addition for single precision calculations like we did in section 4.3 for the double precision case. A look at more fine-grained measurements exhibits that step one of algorithm 2.3 delivers 2.5 the DOFs throughput stage three does, meaning that the horizontal addition is becoming a severe bottleneck.

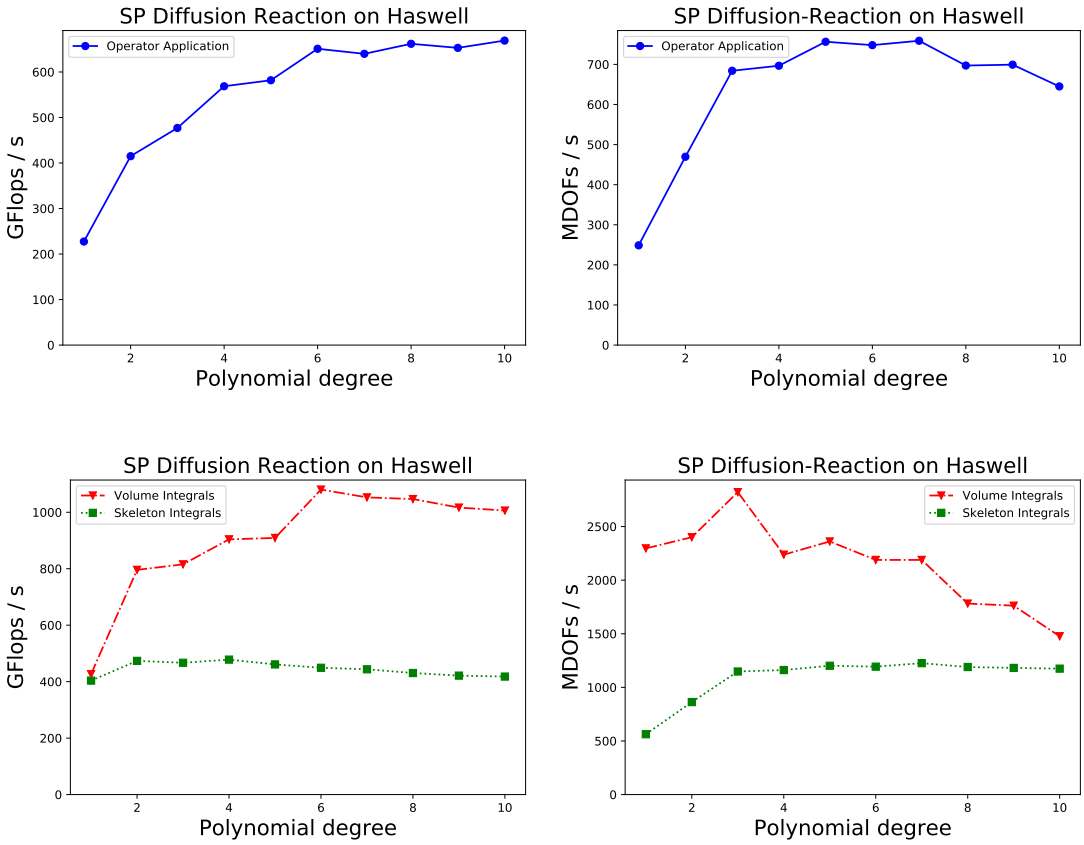


Figure 5.7: Single precision performance of the diffusion-reaction equation on Intel Haswell. On volume integrals, the selected vectorization strategy is of hybrid nature: Four quantities are fused together and a splitting of 2 is applied. Skeleton integrals are vectorized by a fusion-based strategy.

Intel Skylake Results

The single precision results on the Intel Haswell architecture above were the first results with 8 SIMD lanes. We now move to performance measurements on the Intel Skylake architecture which features the AVX-512 instruction set allowing 8 double precision values in one SIMD register. Figure 5.9 shows the results. In order to distinguish effects of the increased SIMD width from effects of the advanced technology in the Intel Skylake architecture, figure 5.10 also shows the speed-up obtained moving from AVX2 to AVX-512 on the machine. The theoretical utilization increase for this case is 1.83 and was described in section 5.1.

A machine utilization of roughly 30% is achieved and the maximum achieved DOFs throughput is around 1GDOFs/s. The performance gaps between volume and skeleton integrals resemble those of the single precision measurements on Intel Haswell. In fact, the superior performance of volume integrals is even more pronounced on the Intel Skylake architecture. While the utilization for volume

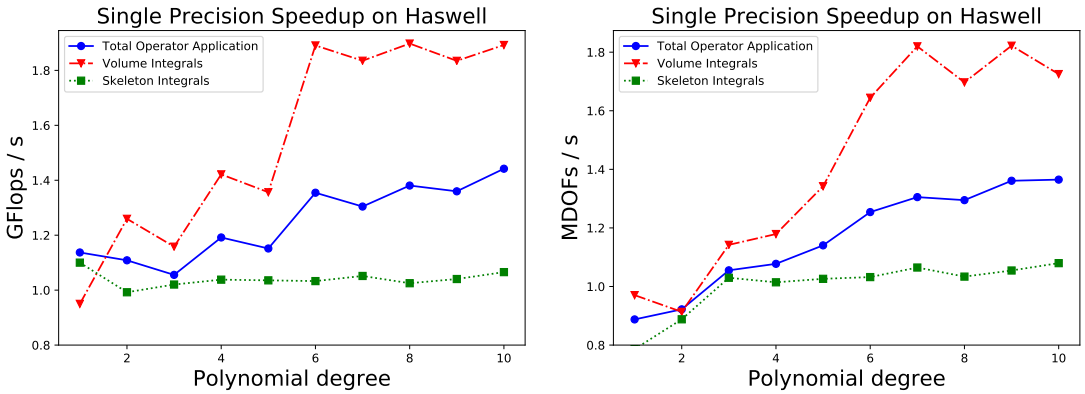


Figure 5.8: Speed-up of the single precision results for the diffusion-reaction equation on Intel Haswell compared to above double precision results. Numbers for volume and skeleton integrals, as well as the full operator application are shown. The perfect speed-up would be 2 which is feasible for volume integrals, but completely out of scope for skeleton integrals.

integrals is comparable with the Haswell single precision results from figure 5.7, skeleton integrals show worse results.

Another interesting number to look at when comparing performance of the examined Intel Haswell and the Intel Skylake processors is the total node performance. While our Intel Haswell node was able to achieve 500 GFLOPs/s, the Skylake node performs up to 850 GFLOPs/s. A number of factors play an important role when comparing these numbers:

- The Skylake node has 40 cores compared to the 32 cores of the Haswell node running at roughly the same frequency.
- The Skylake architecture is an improved successor of the Haswell architecture providing e.g. a more powerful front-end. This becomes apparent when directly comparing AVX2 results between Skylake and Haswell.
- The introduction of AVX-512 and the theoretical utilization increase it offers.

Although performance numbers of our skeleton kernels do not scale with the additional available SIMD width, the overall numbers exhibit a drastically increased node performance and make the Intel Skylake architecture a desirable hardware platform for HPC implementations of DG methods.

5.3 Stokes Equations

We will now move to the steady state Stokes equations, adding the complexity of treating a system of PDEs. The Stokes equations are a linearization of the more general Navier-Stokes equations which is valid in the low Reynolds number regime.

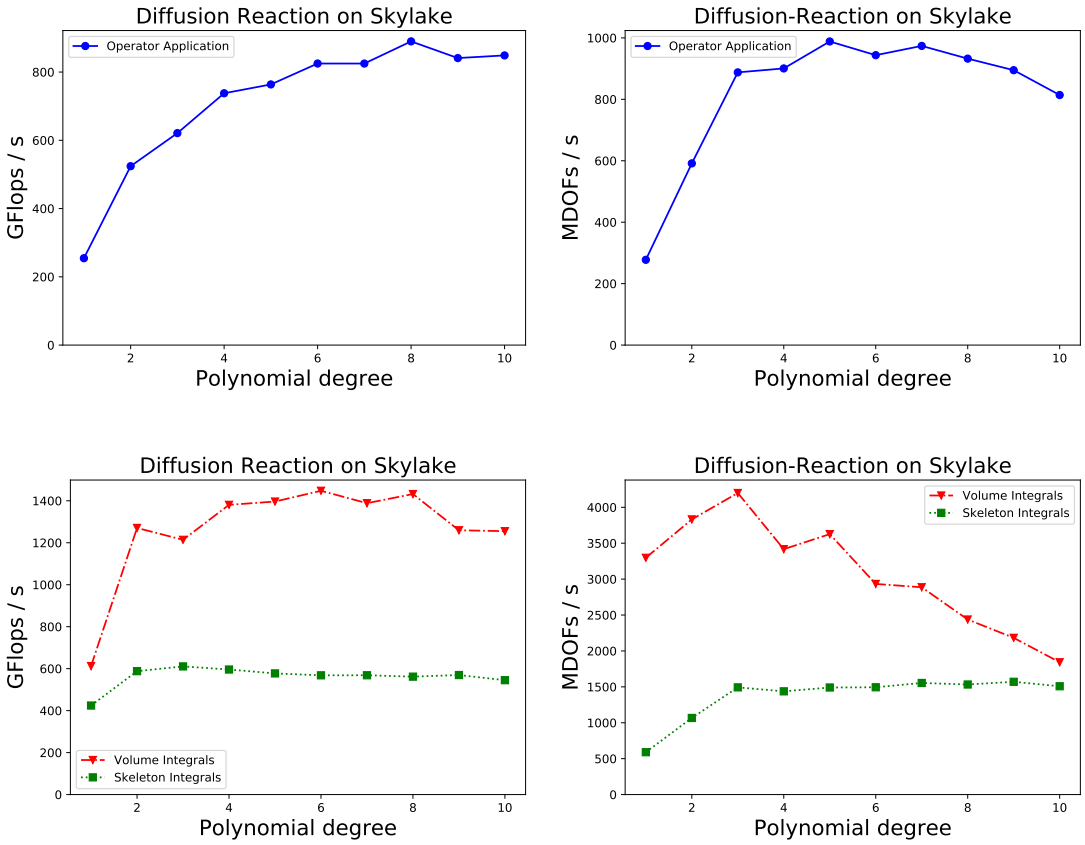


Figure 5.9: Performance of the diffusion-reaction equation on Intel Skylake using AVX-512: We achieve around 30% machine utilization for a full operator application. Volume integrals perform much better and run at 50% whereas skeleton integrals only run at 20%.

Again, the full mathematical formulation and implementation in UFL are provided before presenting performance results for our implementation.

Mathematical Formulation

Both the velocity field $\boldsymbol{u} \in (V_h^k(\Omega))^3$ and the pressure $p \in V_h^{k-1}$ are unknowns of the Stokes equations. The strong formulation is given by

$$-\Delta \boldsymbol{u} + \nabla p = \boldsymbol{f} \qquad \text{in } \Omega \tag{5.6}$$

$$\nabla \cdot \boldsymbol{u} = 0 \qquad \text{in } \Omega \tag{5.7}$$

$$\boldsymbol{u} = \boldsymbol{g} \qquad \text{on } \Gamma_D \tag{5.8}$$

$$\nabla \boldsymbol{u}\, \boldsymbol{n} - p\boldsymbol{n} = 0 \qquad \text{on } \Gamma_N \tag{5.9}$$

for Dirichlet boundary $\Gamma_D \subset \partial\Omega$, $\Gamma_D \neq \{\emptyset, \partial\Omega\}$ and Neumann boundary $\Gamma_N = \partial\Omega \setminus \Gamma_D$. Again, we omitted physical constants such as viscosity, because they are not important for our goal of measuring performance.

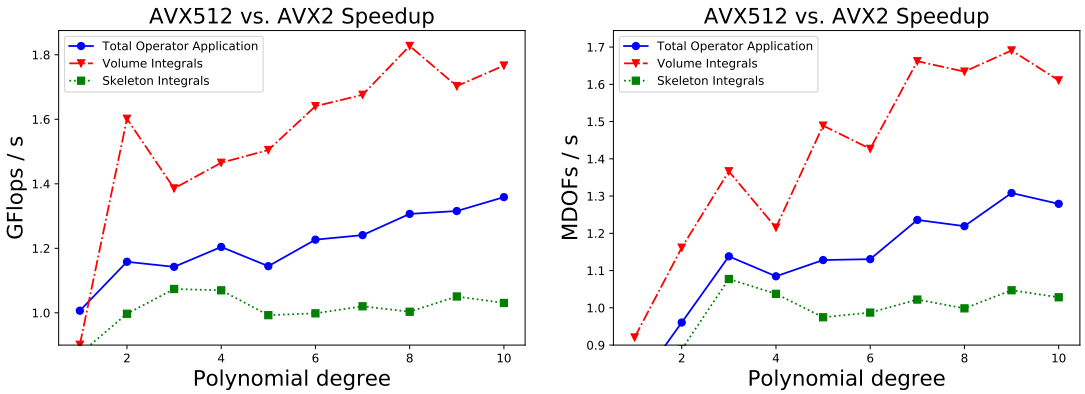


Figure 5.10: Speedup of AVX-512 vs. AVX2 for the diffusion-reaction equation on Intel Skylake. Both results have been measured on the same machine restricting it to the AVX2 instruction subset for the base numbers. The theoretical speedup of such computation is 1.83 due to the reduced base frequency issue described in section 5.1.

The residual function for the DG discretization of this problem is

$$
\begin{aligned}
r(\boldsymbol{u}_h, p_h, \boldsymbol{v}_h, q_h) = &\sum_{T\in\mathcal{T}_h}\int_T \nabla\boldsymbol{u}_h : \nabla\boldsymbol{v}_h - p_h(\nabla\cdot\boldsymbol{v}_h) - (\nabla\cdot\boldsymbol{u}_h)q_h \; dx \\
&+ \sum_{F\in\mathcal{F}_h^i\cup\mathcal{F}_h^D}\int_F -\{\nabla\boldsymbol{u}_h\}\boldsymbol{n}_F\cdot[\![\boldsymbol{v}_h]\!] - [\![\boldsymbol{u}_h]\!]\cdot\{\boldsymbol{v}_h\}\boldsymbol{n}_F + \gamma_F[\![\boldsymbol{u}_h]\!]\cdot[\![\boldsymbol{v}_h]\!] \; ds \\
&+ \sum_{F\in\mathcal{F}_h^i\cup\mathcal{F}_h^D}\int_F \{p_h\}[\![\boldsymbol{v}_h]\!]\cdot\boldsymbol{n}_F + [\![\boldsymbol{u}_h]\!]\cdot\boldsymbol{n}_F\{q_h\} \; ds \\
&+ \sum_{F\in\mathcal{F}_h^D}\int_F \boldsymbol{g}\cdot\nabla\boldsymbol{v}_h\boldsymbol{n}_F - \gamma_F(\boldsymbol{g}\cdot\boldsymbol{v}_h) - (\boldsymbol{g}\cdot\boldsymbol{n}_F)q_h \; ds, \qquad (5.10)
\end{aligned}
$$

where $\mathcal{F}_h^D = \mathcal{B}_h \cap \Gamma_D$ is the set of boundary faces with Dirichlet boundary condition. The penalty term γ_F is chosen exactly as before for the diffusion-reaction equation.

UFL Implementation

The UFL implementation of the Stokes equations is a good example how UFL's property to closely resemble the mathematical formulations is preserved even for complex systems. We only show those parts here that differ from the diffusion-reaction equation. Test and trial functions are taken from a suitable product space which was introduced in section 3.2.1:

```
V = VectorElement("DG", cell, v_degree)
P = FiniteElement("DG", cell, p_degree)
TH = V * P
```

```
v, q = TestFunctions(TH)
u, p = TrialFunctions(TH)
```

Finally, we provide the full residual formulation, which closely resembles the discrete weak formulation from equation 5.10:

```
r = inner(grad(u), grad(v))*dx \
 - p*div(v)*dx \
 - q*div(u)*dx \
 - inner(avg(grad(u))*n, jump(v))*dS \
 + gamma_int * inner(jump(u), jump(v))*dS \
 - inner(avg(grad(v))*n, jump(u))*dS \
 + avg(p)*inner(jump(v), n)*dS \
 + avg(q)*inner(jump(u), n)*dS \
 - inner(grad(u)*n, v)*ds \
 + gamma_ext * inner(u-g_v, v)*ds \
 - inner(grad(v)*n, u-g_v)*ds \
 + p*inner(v, n)*ds \
 + q*inner(u-g_v, n)*ds
```

Here, the velocity boundary condition `g_v` was not further specified as our performance measurements are again omitting boundary integrals.

Intel Haswell Results

Performance measurements of the Stokes equations on the Intel Haswell architecture are presented in figure 5.11. We will start with a discussion of the vectorization strategies that are selected by the autotuning process for this problem. This is interesting because a number of trade off decisions are involved. As we do not need evaluations of ∇p, evalutions of the pressure p cannot be vectorized by a pure fusion strategy, though the evaluation of p^+ and p^- in skeleton integrals can be fused. Vectorization of pressure evaluation therefore depends on the number of quadrature points: If it exhibits suitable divisibility, a splitting strategy is applied to the evaluation of pressure. If the number of quadrature points does not exhibit this divisibility, a non-vectorized implementation is generated. This is indeed optimal, as artificially increasing the number of quadrature points would introduce a global cost increase that outweighs the gains of vectorizing the pressure evaluation. The cost function penalization from equation 4.24 (which is also applied to autotuning measurements) correctly captures this phenomenon. The evaluation of $\nabla \boldsymbol{v}$ on volume integrals requires a total of nine sum factorization kernels with triplets sharing the input tensor. There are two routes to follow in vectorizing this evaluation: Three kernels sharing the same input tensor (e.g. $\partial_i \boldsymbol{v}_j$ for $i = 0, 1, 2$) can be vectorized by fusion ignoring the fourth SIMD lane. On the other hand, the nine kernels could be realized by a number of vectorized implementations using splitting-based and hybrid strategies. Again, the strategy selection depends on the number of quadrature points: For even polynomial degrees we have an odd number

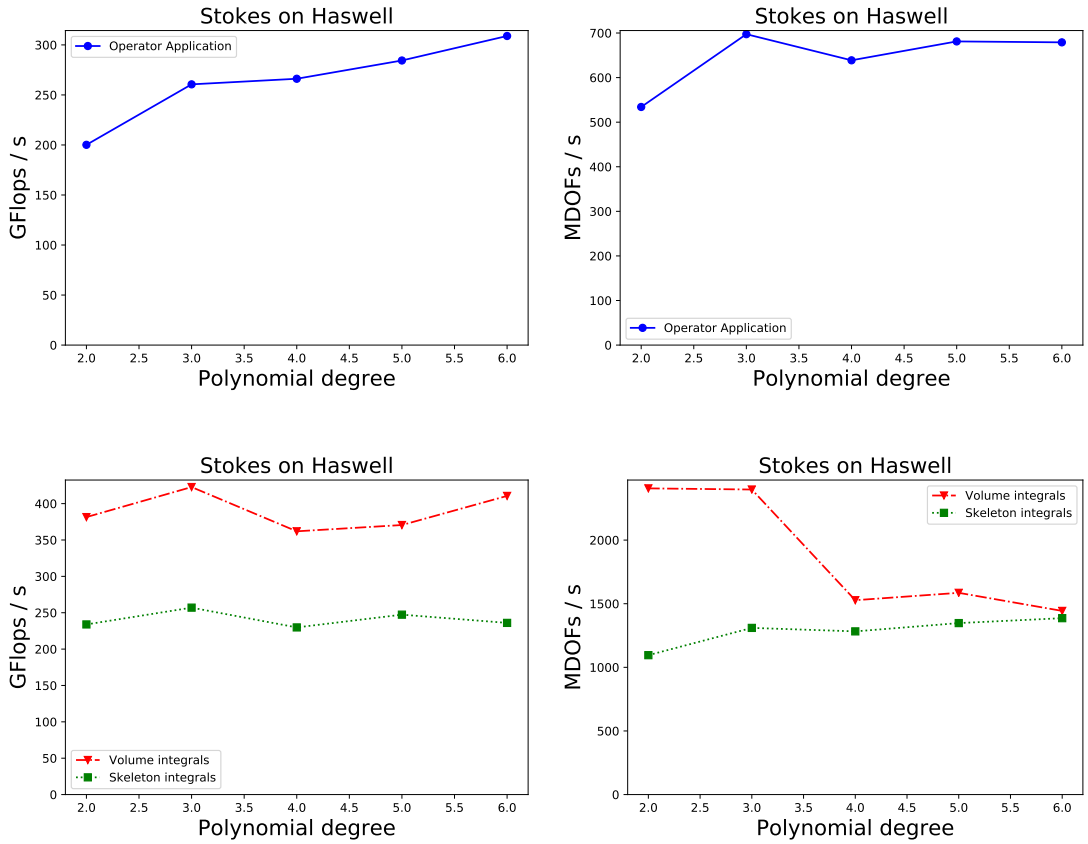


Figure 5.11: Performance measurements of the Stokes equations on Intel Haswell: The overall machine utilization is between 20% and 25% whereas the DOF throughput is slightly increased compared to the diffusion-reaction equation. The given polynomial degrees are used for the velocity components.

of 1D quadrature points and the padded strategy is used. For odd polynomial degrees however, a combination of splitting and hybrid strategies is selected. On facet integrals, the choice of strategy is more straight-forward: As we are using an axiparallel grid, we only need the partial derivatives of the velocity components in normal direction. This allows us to always apply a fusion based vectorization of the form $v_i^- |\partial_n v_i^- | v_i^+ | \partial_n v_i^+$.

With the optimal, purely fusion-based strategies from the diffusion-reaction equation being inapplicable for the Stokes equations, the total machine utilization is decreased in comparison. We still achieve 20 - 25% of machine utilization though. The DOFs throughput is slightly increased in comparison to the diffusion-reaction equation though which can be explained with the fewer FLOPs per DOF executed. This stems from less FLOPs being executed in the quadrature loop and pressure DOFs requiring much less FLOPs. Looking at the more fine-grained measurements for volume and skeleton integrals from figure 5.11, we see that both integral types suffer equally from the inavailability of optimal strategies.

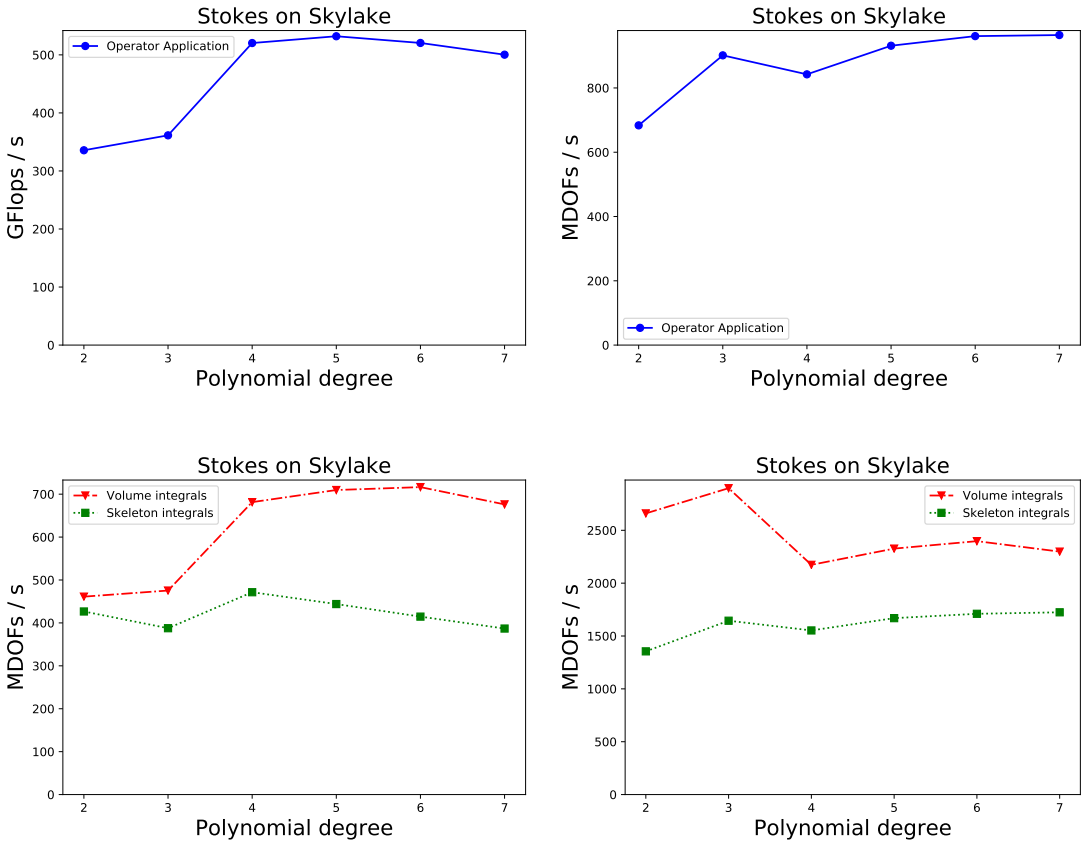


Figure 5.12: Performance measurements of the Stokes equations on Intel Skylake: The machine utilization with the AVX-512 instruction set is around 20%.

Intel Skylake Results

Figure 5.12 shows the measurements for the Stokes equations on the Intel Skylake architecture. The selected vectorization strategies are similar to the ones in the Haswell case with an additional splitting factor of two applied. In order to guarantee the applicability of that strategy, the number of quadrature points needs to be increased in many cases. This explains that despite a clearly visible utilization increase compared to figure 5.11, the DOFs throughput does not increase by the same factor. Interestingly, the performance gap between volume and skeleton integrals is not as pronounced in figure 5.12 as it was before. This can be explained with the selected vectorization strategies for volume integrals not achieving the same utilization as in the diffusion-reaction equation.

5.4 Maxwell's Equations

Maxwell's equation in 3D is a hyberbolic conservation law with six conserved quantities. We will provide a DG discretization for a generic hyperbolic conservation

law and provide a UFL implementation that separates the implementation of the conservation law and the implementation of the numerical flux function. This is similar to an approach pursued lately on top of the FEniCS framework [56]. A similarly modular implementation of DG methods for hyperbolic conservation laws was provided as a tutorial for PDELab in [97]. The description of the treatment of Maxwell's equations in this section follows the latter.

Mathematical Formulation

Hyperbolic conservation laws are time-dependent first order PDE systems of the following form:

$$\partial_t \boldsymbol{u}(\boldsymbol{x},t) + \nabla \cdot \boldsymbol{F}(\boldsymbol{u}(\boldsymbol{x},t),\boldsymbol{x},t) = \boldsymbol{f}(\boldsymbol{u}(\boldsymbol{x},t),\boldsymbol{x},t) \qquad \forall \boldsymbol{x} \in \Omega \times \Sigma \tag{5.11}$$

$$\boldsymbol{u}(\boldsymbol{x},t) = \boldsymbol{u}_0(\boldsymbol{x}) \qquad t = 0 \tag{5.12}$$

Here, Ω is the spatial domain, $\Sigma = [0,T]$ is the temporal interval and $u(\boldsymbol{x},t)$ is the solution. The function $\boldsymbol{F} : \mathbb{R}^m \times \Omega \times \Sigma \to \mathbb{R}^{m \times d}$ is called *flux function* and uniquely describes the PDE problem in the absence of source terms $\boldsymbol{f}$. Discretizing equation 5.11 with a discontinuous function space and applying integration by parts, we end up with the following residual evaluation:

$$r_h(\boldsymbol{u}_h,\boldsymbol{v}_h) = -\sum_{T\in\mathcal{T}_h} \int_T \boldsymbol{F}(\boldsymbol{u}_h(\boldsymbol{x},t),\boldsymbol{x},t) : \nabla \boldsymbol{v}_h \, dx \tag{5.13}$$

$$+ \sum_{F\in\mathcal{F}_h} \int_F \Psi(\boldsymbol{u}_h^+(\boldsymbol{x},t),\boldsymbol{u}_h^-(\boldsymbol{x},t)) \cdot [\![\boldsymbol{v}_h]\!] \, ds \tag{5.14}$$

$$+ \sum_{F\in\mathcal{B}_h} \int_F \Psi(\boldsymbol{g}(\boldsymbol{x},t),\boldsymbol{u}_h(\boldsymbol{x},t)) \cdot [\![\boldsymbol{v}_h]\!] \, ds \tag{5.15}$$

Here, Ψ is a *numerical flux function* that provides the evaluation of the flux on the two-valued interface. For scalar problems, the simplest possible flux function is an upwind function, which selects either $\boldsymbol{u}^+$ or $\boldsymbol{u}^-$ depending on the sign of the flux function. With Maxwell's equations being a system of PDEs, we apply a generalization of upwinding called *flux vector splitting*. This method relies on the system being linear and the reformulation of the flux function in terms of a matrix $\boldsymbol{B}$:

$$\boldsymbol{F}(\boldsymbol{u}(\boldsymbol{x},t),\boldsymbol{x},t) \cdot \boldsymbol{n} = \boldsymbol{B}(\boldsymbol{n})\boldsymbol{u}(\boldsymbol{x},t) \tag{5.16}$$

From the hyperbolicity of the system 5.11 follows that $\boldsymbol{B}(\boldsymbol{n})$ is real diagonalizable with eigenvalues λ_i. The corresponding eigenvectors are the columns of a matrix $\boldsymbol{R}$, such that $\boldsymbol{B} = \boldsymbol{R}\boldsymbol{D}\boldsymbol{R}^{-1}$. Multiplication with $\boldsymbol{R}^{-1}$ transforms a state $\boldsymbol{u}$ into characteristic variables, where the flux function is diagonal and upwinding can

be applied for each component individually. We express the whole splitting in a condensed matrix notation:

$$\begin{aligned} \boldsymbol{D}^+ &= diag\left(\max\{0, \lambda_1\}, \ldots, \max\{0, \lambda_m\}\right) \\ \boldsymbol{D}^- &= diag\left(\min\{0, \lambda_1\}, \ldots, \min\{0, \lambda_m\}\right) \\ \boldsymbol{B}^+ &= \boldsymbol{R}\boldsymbol{D}^+\boldsymbol{R}^{-1} \\ \boldsymbol{B}^- &= \boldsymbol{R}\boldsymbol{D}^-\boldsymbol{R}^{-1} \\ \Psi(\boldsymbol{n}, \boldsymbol{u}^+, \boldsymbol{u}^-) &= \boldsymbol{B}^+(\boldsymbol{n})\boldsymbol{u}^- + \boldsymbol{B}^-(\boldsymbol{n})\boldsymbol{u}^+ \end{aligned} \tag{5.17}$$

UFL Implementation

Formulating the discretization of Maxwell's equations directly in UFL can be easily done. For the sake of composability of the resulting implementation, we will split it into three main building blocks: A generic DG discretization of hyperbolic conservation laws in UFL, a description of the conservation law and a description of the numerical flux function. Introducing interfaces for these blocks allows reuse of these components in a composable way.

A description of a hyperbolic conservation law needs to provide the following information: The number of conserved quantities in the system, the flux function $\boldsymbol{F}$ and a boundary condition. On top of this numerical flux functions might require additional information, such as eigenvalues and eigenvectors of the matrix $\boldsymbol{B}(\boldsymbol{n})$ from equation 5.16. The resulting interface and its implementation for Maxwell's equation look as follows:

```
class MaxwellProblem(GenericHyperbolicProblem):
    @property
    def components(self):
        return 6

    def flux(self, state):
        E0, E1, E2, B0, B1, B2 = state
        return as_matrix([[0.,   -B2,  B1 ],
                          [B2,   0.,   -B0],
                          [-B1,  B0,   0. ],
                          [0.,   E2,   -E1],
                          [-E2,  0.,   E0 ],
                          [E1,   -E0,  0. ]])

    def eigenvalues(self, state):
        return (0.0, 0.0, -1.0, -1.0, 1.0, 1.0)

    def eigenvectors(self, state):
        n = FacetNormal(self.cell)('+')
```

```
        a = [conditional(abs(n[2]) < 0.5, n[0]*n[2],   n[0]*n[1]   ),
             conditional(abs(n[2]) < 0.5, n[1]*n[2],   n[1]*n[1]-1 ),
             conditional(abs(n[2]) < 0.5, n[2]*n[2]-1, n[1]*n[2]   )]

        b = [conditional(abs(n[2]) < 0.5, -n[1], n[2]  ),
             conditional(abs(n[2]) < 0.5, n[0],  0.    ),
             conditional(abs(n[2]) < 0.5, 0.,    -n[0] )]

        return as_matrix(
            [[n[0], 0,    b[0],  a[0], a[0], -b[0]],
             [n[1], 0,    b[1],  a[1], a[1], -b[1]],
             [n[2], 0,    b[2],  a[2], a[2], -b[2]],
             [0,    n[0], a[0], -b[0], b[0],  a[0]],
             [0,    n[1], a[1], -b[1], b[1],  a[1]],
             [0,    n[2], a[2], -b[2], b[2],  a[2]]])

    def boundary_state(self, state):
        return (0.0,) * self.components
```

For the sake of simplicity, this example assumes the physical constants present in Maxwell's equation to be 1. This does not notably affect any performance numbers presented in this section.

Numerical flux functions are implemented as Python callables which accept the outer and inner state as their only arguments. For Maxwell's equations, we implement the flux vector splitting from equation 5.17. With the problem class and the flux function being defined, the generic spatial DG discretization can be obtained in the following way:

```
def generic_hyperbolic_operator(problem, numerical_flux, degree=1):
    V = FiniteElement("DG", problem.cell, degree)
    MV = MixedElement(*tuple(V for i in range(problem.components)))

    u = TrialFunction(MV)
    v = TestFunction(MV)

    state = split(u)
    outer_state = tuple(s('+') for s in state)
    inner_state = tuple(s('-') for s in state)
    boundary_state = problem.boundary_state(state)

    r = -1. * inner(problem.flux(state), grad(v))*dx \
      + inner(numerical_flux(outer_state, inner_state), jump(v))*dS \
      + inner(numerical_flux(boundary_state, state), v)*ds

    return r
```

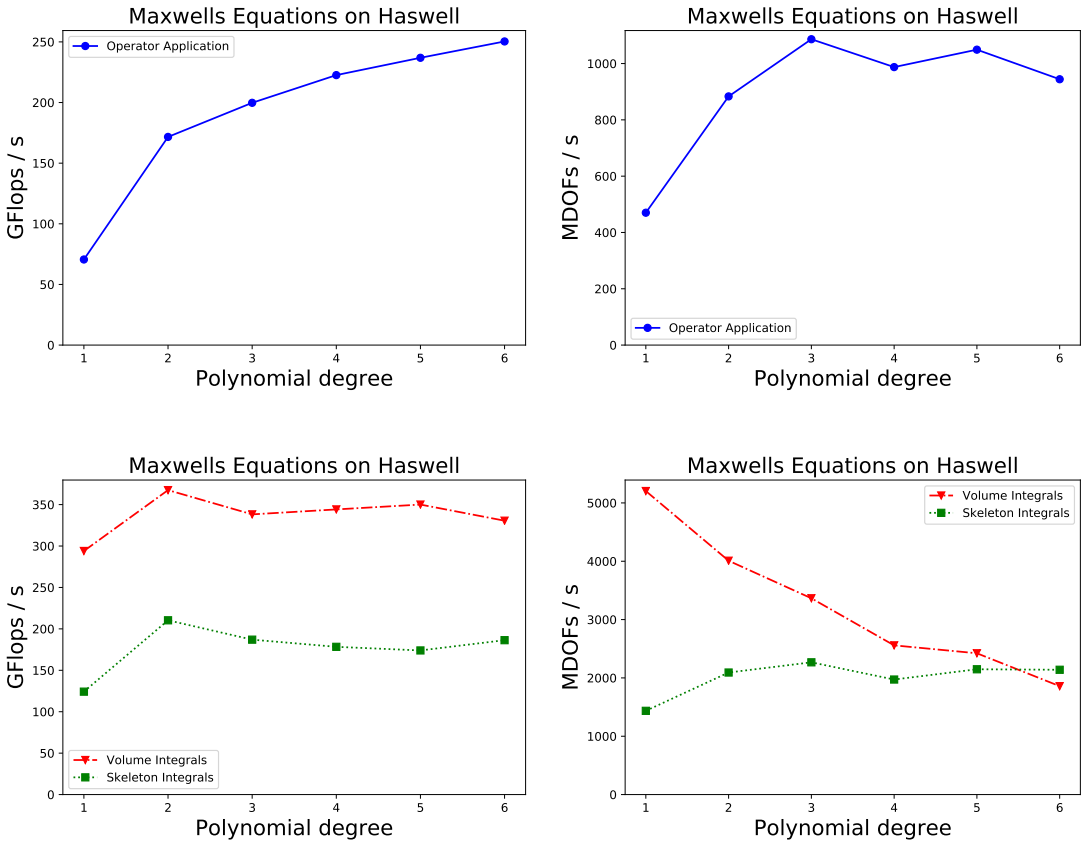


Figure 5.13: Performance results for 3D Maxwells equations on the Intel Haswell architecture: The upper plots show GFLOPs/s and MDOF/s for a full residual evaluation. The lower plots split these results into volume and skeleton integrals. 20% of theoretical machine peak are reached for this system of six components.

A UFL form for the necessary mass operators can easily defined as `u*v*dx`.

Intel Haswell Results

We will now present the performance result of Maxwells equations in 3D on the Intel Haswell architecture. As the time discretization of hyperbolic systems is often done with explicit Ordinary Differential Equation (ODE) solvers, we measure evaluation of the residual instead of matrix-free applications of the jacobian. Figure 5.13 summarizes the results for varying polynomial degree. Across a full residual evaluation, we can achieve roughly 20% of the machine's theoretical peak performance and sustain a total of roughly 1 GDOF/s.

Comparing these results to the performance numbers achieved for the diffusion-reaction equation on the same machine from figure 5.1 we see that while the machine utilization is much lower (approximately half), the DOF thoughput is much

higher (approximately doubled). The following two paragraphs will describe two aspects that allow explanation of this discrepancy.

The integrals in the generic DG problem from equation 5.11 depend only on the state $\boldsymbol{u}$ and not on its gradient, so all sum factorization kernels in step one of algorithm 2.3 stem from evaluation of components of $\boldsymbol{u}$. The fact that we did only allow a maximum of two different input kernels when fusing sum factorization kernels into SIMD batches rules out many fusion-based vectorization strategies. Then again, these strategies worked optimally for the diffusion-reaction equation as shown in figure 5.3. This partly explains the reduced machine utilization observed in figure 5.13. In future work, this could be remedied by allowing fusion of sum factorization kernels with four different input tensors. Fused kernels of this kind would utilize SIMD gather instructions (instead of SIMD broadcasts) to load data in step 1 of algorithm 2.3 and extract scalar values from SIMD lanes in the accumulation step.

With the Maxwell implementation achieving roughly twice the throughput while having roughly half the machine utilization, it is obvious that it performs only a quarter of the FLOPs compared to the diffusion-reaction equation. Partly, this can be credited to the reduced number of sum factorization kernels described before due to the absence of gradients in the equations (note the volume integral still depends on $\nabla \boldsymbol{v}$). However, there is another important aspect that reduces the number of executed FLOPs: As the matrices $\boldsymbol{B}(\boldsymbol{n})$ and $\boldsymbol{R}(\boldsymbol{n})$ only depend on the normal vector and normal vectors are explicitly known at code generation time in the axiparallel sum factorization setting, the inverse matrix $\boldsymbol{R}^{-1}(\boldsymbol{n})$ can be calculated at code generation time. As a consequence the evaluation of the numerical flux function from equation 5.17 becomes a linear combination of components of $\boldsymbol{u}$. Furthermore, the special structure of the Maxwell problem results in those fluxes only depending on four of the six components of $\boldsymbol{u}$. With this being code generation time information, the other components never need to be evaluated. This minimization of executed FLOPs is a genuine advantage of the code generation approach which is infeasible to implement in C++ implementations like [97]. Those codes necessarily need to assemble, invert and apply the 6×6 matrix $\boldsymbol{R}(\boldsymbol{n})$ in each skeleton integral. The code generation toolchain can also implement such procedure, if code generation time evaluation is impossible, e.g. in the case of spatially varying material properties that affect $\boldsymbol{B}(\boldsymbol{n})$.

With the fusion based approach being only partly applicable even on AVX2, moving to AVX-512 will exhibit even less machine utilization with the currently implemented vectorization strategies. We conclude that scaling Maxwells equations beyond AVX2 requires inclusion of additional strategies into our search space. One such strategy is the inclusion of fusions with four different input tensors that can be implemented by vector gather instructions. Another class of vectorization strategies worth exploring is cross-element vectorization and hybrid strategies of our approach and cross-element vectorization.

CHAPTER 6

Closing Remarks

6.1 Summary

Throughout this thesis and its performance experiments we have seen that DG methods can be a good fit to achieve high performance on HPC systems. This especially holds true for matrix-free evaluations of the DG operator that exploit the tensor product structure of basis functions and quadrature formulae through sum factorization. These algorithms are able to perform both finite element assembly and the necessary sparse linear algebra operations in the FLOP-bound regime. The availability of such algorithms is an important fact for the HPC community.

Regarding the efficient implementation of these methods on modern architectures, we observe that many of the programmability issues with heterogeneous computing environments even apply to different CPU generations. This is due to a lack of toolchain support for codes that explicitly employ SIMD vectorization. We therefore advocate the use of generative programming to achieve performance portability across CPU instruction sets with increasing SIMD width. This allows flexibility in how to vectorize and removes the burden of writing hardware-specific code from the user.

This thesis has established an HPC-enabled toolchain that allows to generate code for finite element integration kernels. It does so by leveraging existing projects such as the UFL DSL for finite element problems and the `loopy` IR that allows us to apply performance-oriented code transformation. The same IR is used to express sum factorization kernels and generate code for them. We embed this toolchain into the user workflow of DUNE, focussing on the generation of innermost loops in the finite element integration kernels.

Furthermore, this thesis has provided a new class of SIMD vectorization strategies that work on batches of subkernels within an integration kernel. We have studied several variants whose applicability relies on features of the mathematical model, such as the occurence of specific terms in the PDE or the number of quadrature points used for integration. Embedding of the vectorization strategy selection algorithm into the code generator allows to inspect the PDE problem to find suitable variants. From these, an autotuning approach selects the best variant by running a set of micro benchmarks at code generation time. The definition of performance optimization search spaces and their exploration at code generation time allows a separation of concern between performance engineers and computational scientists.

We have shown performance measurements of our approach for the diffusion-reaction equation, the Stokes equations and Maxwell's equations. For a full application of the DG operator in the matrix-free setting, a machine utilization of 40% could be reached on the Intel Haswell architecture. For volume integrals, the utilization even reaches 60%, while skeleton integrals run at rougly 35% of the machine's theoretical peak. On the Intel Skylake architecture with the AVX-512 instruction set, the differences between integral types are even more pronounced: Volume integrals run at 50% utilization and almost show the expected speed-up compared to AVX2 for sufficiently high polynomial degrees. Skeleton integrals however cannot achieve such speed-ups and limit the performance of the overall algorithm.

6.2 Outlook

The vectorization strategy search space introduced in section 4 could be improved by considering additional strategies. We have already seen the necessity of fusion strategies with an arbitrary number of input tensors in the case of Maxwell's equations. Furthermore, inclusion of cross-element vectorization strategies would be very interesting: This would allow rigorous comparison of our strategies with those developed by e.g. [76] and [109], which is currently not possible because no code base supports both types of vectorization strategies. Integrating cross-element vectorization into our framework would also open up space for interesting hybrid strategies, where e.g. two volume integrals are batched together such that the lower and upper half of a SIMD register calculate the contributions for one cell.

With our code generator's vectorization capabilities being implemented independently of the SIMD width, it will be very interesting to follow future SIMD developments. If the trend of increasing SIMD width continues, the above mentioned extension of the search space will be necessary. If future CPU generations focus on improving cross-lane manipulation of SIMD registers such as horizontal addition, our strategies would benefit significantly.

We have restricted the performance measurements in this thesis to structured grids. However, work on doing the same on unstructured grids has already been started. This involves two major challenges: The multilinear (or even higher order)

geometry mappings need to be evaluated using sum factorization techniques in order to preserve the overall algorithmic complexity, much like we mentioned for the semistructured geometries in section 4.4.4. On the other hand, the embedding of the reference element of the facet into the cell reference element requires special care in the code generator for fully unstructured grids.

Our code generator and its HPC-enabled IR have also seen use in a different area of HPC: In order to apply SIMD vectorization to low order CG finite element methods, virtually refined volume integral kernels are generated. Within these kernels, explicit SIMD vectorization is employed through `loopy` transformations.

So far, the autotuning procedure has been restricted to the selection of vectorization strategies from the search space defined in section 4. However, many other search spaces are imaginable. We already mentioned some transformation opportunities for sum factorization kernels such as loop reordering and loop tiling in section 4.4.3. The code generation approach is a very good fit to introduce such search spaces.

Finally, we mention that although the introduction of code generation into the DUNE framework was motivated by the need for HPC implementations, the implemented toolchain has more benefits: Robust code generation for complex PDE models greatly reduces development time for both experienced programmers and beginners. This adds an exciting new facet to DUNE's rich feature set.

Hardware Configurations

We used two compute nodes for the performance measurements in section 5. We now give details about the hardware and its configuration. The frequencies used to calculate the theoretical peak performance are those that we measured during the execution of SIMD-heavy workloads. These frequencies match the minimum of the CPU base frequency and the *all core turbo frequency* given in [31].

A.1 Intel Haswell

- Dual-socket main board
- Intel Xeon processor E5-2698v3
  - 2x 16 cores
  - Turbo mode switched off
  - Base frequency: 2.3 GHz, 1.9 GHz on AVX2-heavy loads
  - Observed frequency for AVX2-heavy loads: 2.3 GHz
  - Theoretical full node double precision peak performance: 1.17 TFLOPs/s
  - TDP: 270 W
- 128 GB DDR4 RAM, 2133 MHz

A.2 Intel Skylake

- Dual-socket main board
- 2x Intel Xeon Gold 6148
  - 2x 20 cores

  - Turbo mode switched off
  - Base frequency: 2.4 GHz, 2.0 GHz on AVX2-heavy loads, 1.6 GHz on AVX-512-heavy loads
  - Observed frequency for AVX2-heavy loads: 2.4 GHz
  - Observed frequency for AVX-512-heavy loads: 2.2 GHz
  - Theoretical full node double precision peak performance: 2.82 TFLOPs/s
  - TDP: 300 W
- 384 GB DDR4 RAM, 2666 MHz

Getting the Software

This thesis documents the design and implementation of several software projects. In this appendix, we provide details about how the source code of these software projects can be obtained. We distinguish two intentions the reader might have in doing so: Firstly, we provide an exact description of the DUNE software stack at the time of this writing. We provide this in order to allow the reader to reproduce any results of this thesis. Secondly, we explain how the reader can obtain the latest version of the software contributions of this thesis. In both cases, the software is distributed as DUNE modules and the reader should familiarize themselves with the DUNE build system which is documented on the DUNE website [111].

B.1 Obtaining dune-codegen

The *dune-codegen* module [62] is the main project that provides the form compiler that translates UFL into C++ code that can be used with DUNE. dune-codegen depends on the following other DUNE modules:

- *dune-pdelab* and its dependencies which include all the core functionality of DUNE. PDELab provides all the components of the simulation workflow which are not covered by the code generator. PDELab can be obtained from the DUNE GitLab server [96].
- *dune-opcounter* provides the C++ floating point type that automatically counts FLOPs. It also provides an interface compatible, operation counting version of the vector class library from section 3.2.3. It is also available from the DUNE GitLab server [66].
- *dune-testtools* [65] provides the build system integration for system tests. The systematic covery of functionality through these tests is essential for the

software quality of dune-codegen. dune-testtools is available from the DUNE GitLab as well [64].

- *dune-alugrid* [6] is used for testing discretizations on simplicial meshes.

Beyond these DUNE dependencies, a Python interpreter (preferrably version 3.6 or higher) with the Python packages `virtualenv` (or `venv`) and `pip` installed is required.

In contrast to other DUNE modules, dune-codegen makes use of `git` submodules to bundle upstream dependencies such as necessary Python projects. Therefore, dune-codegen needs to be cloned recursively:

```
git clone --recursive \
    https://gitlab.dune-project.org/extensions/dune-codegen.git
```

Currently, this clones the master branch of dune-codegen, but starting with the upcoming 2.7 release of DUNE, release branches that stay compatible with the release branches of other DUNE modules will become available. As dune-codegen currently still relies on patching its Python dependencies in some places, this additional step is necessary:

```
cd dune-codegen
./patches/apply_patches.sh
```

The stack of DUNE modules is best built with the `dunecontrol` tool provided by dune-common. In order to enable the execution of Python code under control of CMake, the following two options should be given to the CMake process:

```
-DDUNE_PYTHON_VIRTUALENV_SETUP=1
-DDUNE_PYTHON_ALLOW_GET_PIP=1
```

This will allow the creation of a virtual environment that is shared between all DUNE modules that contain Python code. If the user-provided Python interpreter is from a virtual environment itself, the environment needs to be activated before running CMake. The build directories of these DUNE modules contains an `activate` script which enables DUNE's virtual environment, e.g. like this:

```
source activate
which python
deactivate
```

However, the intended user workflow does not require users to explicitly activate the virtual environment. It can be useful however to do so in some cases, e.g. to get a list of configuration options for the code generation process:

```
source activate
show_options
```

Users should typically write their own DUNE module and list dune-codegen as a dependency of the module. Within such a user module, the code generator is controlled through the CMake functions described in section 3.4.2.

B.2 Reproducing this thesis

The stack of DUNE modules that was used for the results in this thesis is archived in [60]. The reference provides the exact state of all DUNE modules that were in use as of the writing of this thesis. Once checked out, the software can be built using the same instructions as above.

Bibliography

[1] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. *Small Tensor Operations on Advanced Architectures for High-order Applications*. Tech. rep. Technical Report UT-EECS-17-749, 04-2017 2017. — [75]

[2] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. "Performance, design, and autotuning of batched GEMM for GPUs". In: *International Conference on High Performance Computing*. Springer. 2016, pp. 21–38. — [99]

[3] R. Agelek, M. Anderson, W. Bangerth, and W. L. Barth. "On orienting edges of unstructured two-and three-dimensional meshes". In: *ACM Transactions on Mathematical Software (TOMS)* 44.1 (2017), p. 5. — [116]

[4] M. Ainsworth, G. Andriamaro, and O. Davydov. "Bernstein–Bézier finite elements of arbitrary order and optimal assembly procedures". In: *SIAM Journal on Scientific Computing* 33.6 (2011), pp. 3087–3109. — [22]

[5] B. Aktemur, Y. Kameyama, O. Kiselyov, and C.-c. Shan. "Shonan challenge for generative programming: short position paper". In: *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*. ACM. 2013, pp. 147–154. — [47]

[6] M. Alkämper, A. Dedner, R. Klöfkorn, and M. Nolte. "The dune-alugrid module". In: *arXiv preprint arXiv:1407.6954* (2014). — [10, 136]

[7] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. "Unified form language: A domain-specific language for weak formulations of partial differential equations". In: *ACM Transactions on Mathematical Software (TOMS)* 40.2 (2014), p. 9. — [17, 28, 29]

[8] M. S. Alnæs, A. Logg, and K.-A. Mardal. "UFC: a finite element code generation interface". In: *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012, pp. 283–302. — [17]

[9] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. "Opentuner: An extensible framework for program autotuning". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM. 2014, pp. 303–316. — [99]

[10] P. Antolin, A. Buffa, F. Calabrò, M. Martinelli, and G. Sangalli. "Efficient matrix computation for tensor-product isogeometric analysis: The use of sum factorization". In: *Computer Methods in Applied Mechanics and Engineering* 285 (2015), pp. 817–828. DOI: 10.1016/j.cma.2014.12.013. — [22]

[11] D. N. Arnold. "An interior penalty finite element method with discontinuous elements". In: *SIAM journal on numerical analysis* 19.4 (1982), pp. 742–760. — [8]

[12] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. *PETSc users manual*. Tech. rep. Technical Report ANL-95/11-Revision 2.1. 5, Argonne National Laboratory, 2004. — [17]

[13] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz-Reichert, and C. Wieners. "UG – A flexible software toolbox for solving partial differential equations". In: *Computing and Visualization in Science* 1.1 (July 1997), pp. 27–40. DOI: 10.1007/s007910050003. — [10]

[14] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. "A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE". In: *Computing* 82.2–3 (2008), pp. 121–138. — [10]

[15] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. "A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework". In: *Computing* 82.2–3 (2008), pp. 103–119. — [10]

[16] P. Bastian, M. Blatt, and R. Scheichl. "Algebraic multigrid for discontinuous Galerkin discretizations of heterogeneous elliptic problems". In: *Numerical Linear Algebra with Applications* 19.2 (2012), pp. 367–388. — [24]

[17] P. Bastian and C. Engwer. "An unfitted finite element method using discontinuous Galerkin". In: *International journal for numerical methods in engineering* 79.12 (2009), pp. 1557–1576. — [14]

[18] P. Bastian, C. Engwer, J. Fahlke, M. Geveler, D. Göddeke, O. Iliev, O. Ippisch, R. Milk, J. Mohring, S. Müthing, et al. "Hardware-based efficiency advances in the EXA-DUNE project". In: *Software for Exascale Computing-SPPEXA 2013-2015*. Springer, 2016, pp. 3–23. — [19, 21, 75]

[19] P. Bastian, F. Heimann, and S. Marnach. "Generic implementation of finite element methods in the distributed and unified numerics environment (DUNE)". In: *Kybernetika* 46.2 (2010), pp. 294–315. — [11]

[20] C. E. Baumann and J. T. Oden. "A discontinuous hp finite element method for convection—diffusion problems". In: *Computer Methods in Applied Mechanics and Engineering* 175.3-4 (1999), pp. 311–341. — [8]

[21] D. Braess. *Finite Elemente: Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer-Verlag, 2013. — [5]

[22] P. E. Buis and W. R. Dyksen. "Efficient vector and parallel manipulation of tensor products". In: *ACM Transactions on Mathematical Software (TOMS)* 22.1 (1996), pp. 18–23. — [21, 93]

[23] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, and C. Jégo. "MIPP: a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard". In: *the 2018 4th Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*. ACM Press. 2018. — [53]

[24] J. Chessa, P. Smolinski, and T. Belytschko. "The extended finite element method (XFEM) for solidification problems". In: *International Journal for Numerical Methods in Engineering* 53.8 (2002), pp. 1959–1977. — [32]

[25] B. Cockburn and C.-W. Shu. "The local discontinuous Galerkin method for time-dependent convection-diffusion systems". In: *SIAM Journal on Numerical Analysis* 35.6 (1998), pp. 2440–2463. — [9]

[26] B. Cockburn and C.-W. Shu. "The Runge–Kutta discontinuous Galerkin method for conservation laws V: multidimensional systems". In: *Journal of Computational Physics* 141.2 (1998), pp. 199–224. — [9]

[27] P. Cordes. *Get sum of values stored in ___m256d with SSE/AVX*. https://stackoverflow.com/questions/49941645/get-sum-of-values-stored-in-m256d-with-sse-avx/49943540. Accessed: 2019-04-25. — [86, 88]

[28] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. "Generative programming for embedded software: An industrial experience report". In: *International Conference on Generative Programming and Component Engineering*. Springer. 2002, pp. 156–172. — [16]

[29] L. Dagum and R. Menon. "OpenMP: An industry-standard API for shared-memory programming". In: *Computing in Science & Engineering* 1 (1998), pp. 46–55. — [19]

[30] T. A. Davis. "Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method". In: *ACM Transactions on Mathematical Software (TOMS)* 30.2 (2004), pp. 196–199. — [52, 67]

[31] J. De Gelas and I. Cutress. *Sizing Up Servers: Intel's Skylake-SP Xeon versus AMD's EPYC 7000 - The Server CPU Battle of the Decade?* https://www.anandtech.com/show/11544/intel-skylake-ep-vs-amd-epyc-7000-cpu-battle-of-the-decade/8. Accessed: 2019-06-26. — [133]

[32] A. Dedner and M. Nolte. "The Dune Python Module". In: *arXiv preprint arXiv:1807.05252* (2018). — [10, 26, 68]

[33] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. "Liszt: a domain specific language for building portable mesh-based PDE solvers". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2011, p. 9. — [18]

[34] D. A. Di Pietro and A. Ern. *Mathematical aspects of discontinuous Galerkin methods*. Vol. 69. Springer Science & Business Media, 2011. — [5]

[35] L. T. Diosady and S. M. Murman. "Tensor-product preconditioners for higher-order space–time discontinuous Galerkin methods". In: *Journal of Computational Physics* 330.Supplement C (2017), pp. 296–318. DOI: https://doi.org/10.1016/j.jcp.2016.11.022. — [24]

[36] R. Dolbeau. “Theoretical peak FLOPS per instruction set: a tutorial”. In: *The Journal of Supercomputing* (2017), pp. 1–37. — [107]

[37] H. C. Edwards, C. R. Trott, and D. Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3202–3216. — [19]

[38] A. Ern, A. F. Stephansen, and P. Zunino. “A discontinuous Galerkin method with weighted averages for advection–diffusion equations with locally small and anisotropic diffusivity”. In: *IMA Journal of Numerical Analysis* 29.2 (2009), pp. 235–256. — [9]

[39] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté. “Boost. simd: generic programming for portable simdization”. In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM. 2014, pp. 1–8. — [53]

[40] A. Fog. “Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs”. In: *Copenhagen University College of Engineering* 93 (2011), p. 110. — [55, 86]

[41] A. Fog. “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers”. In: *Copenhagen University College of Engineering* (2012), pp. 02–29. — [55]

[42] A. Fog. “VCL C++ vector class library v 1.30”. In: (). URL: http://www.agner.org/optimize/vectorclass.pdf. — [53, 86]

[43] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995. — [40, 64]

[44] M. Ganapathi, C. N. Fischer, and J. L. Hennessy. “Retargetable compiler code generation”. In: *ACM Computing Surveys (CSUR)* 14.4 (1982), pp. 573–592. — [15]

[45] R. Garg and L. Hendren. “A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems”. In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2014, pp. 672–680. — [99]

[46] C. Geuzaine and J.-F. 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 (2009), pp. 1309–1331. — [38]

[47] M. Giles. “An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation”. In: *the 5th International Conference on Automatic Differentiation*. 2008. — [59]

[48] M. Godbolt. *Godbolt Compiler Explorer*. http://www.godbolt.org. — [86]

[49] Google. *benchmark*. https://github.com/google/benchmark. 2019. — [100]

[50] Google. *dimsum*. https://github.com/google/dimsum. 2019. — [53]

[51] A. Griewank et al. "On automatic differentiation". In: *Mathematical Programming: recent developments and applications* 6.6 (1989), pp. 83–107. — [44]

[52] F. Hecht. "New development in FreeFem++". In: *Journal of numerical mathematics* 20.3-4 (2012), pp. 251–266. — [18]

[53] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. "LIBXSMM: accelerating small matrix multiplications by runtime code generation". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2016, p. 84. — [74]

[54] M. Homolya, R. C. Kirby, and D. A. Ham. "Exposing and exploiting structure: optimal code generation for high-order finite element methods". In: *CoRR* abs/1711.02473 (2017). arXiv: 1711.02473. URL: http://arxiv.org/abs/1711.02473. — [21]

[55] M. Homolya, L. Mitchell, F. Luporini, and D. A. Ham. "TSFC: a structure-preserving form compiler". In: *SIAM Journal on Scientific Computing* 40.3 (2018), pp. C401–C428. — [17, 61]

[56] P. Houston and N. Sime. "Automatic symbolic computation for discontinuous Galerkin finite element methods". In: *SIAM Journal on Scientific Computing* 40.3 (2018), pp. C327–C357. — [124]

[57] P. Kanapickas. *libsimdpp*. https://github.com/p12tic/libsimdpp. 2019. — [53]

[58] P. Karpiński and J. McDonald. "A high-performance portable abstract interface for explicit SIMD vectorization". In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM. 2017, pp. 21–28. — [53]

[59] R. Karrenberg. "Whole-function vectorization". In: *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer, 2015, pp. 85–125. — [74]

[60] D. Kempf. *Code Generation for High Performance PDE Solvers on Modern Architectures - Software Stack*. June 2019. DOI: 10.5281/zenodo.3258324. — [137]

[61] D. Kempf and P. Bastian. "An HPC perspective on generative programming". In: *Proceedings of the Software Engineering for Science workshop*. 2019. — [3, 16, 18]

[62] D. Kempf, R. Heß, and M. Koch. *dune-codegen*. https://gitlab.dune-project.org/extensions/dune-codegen. 2019. — [68, 135]

[63] D. Kempf, R. Heß, S. Müthing, and P. Bastian. "Automatic Code Generation for High-Performance Discontinuous Galerkin Methods on Modern Architectures". In: *arXiv preprint arXiv:1812.08075* (2018). — [3, 22, 75, 105]

[64] D. Kempf and T. Koch. *dune-testtools*. https://gitlab.dune-project.org/quality/dune-testtools. 2019. — [70, 136]

[65] D. Kempf and T. Koch. "System testing in scientific numerical software frameworks using the example of DUNE". In: *Archive of Numerical Software* 5.1 (2017), pp. 151–168. — [3, 10, 70, 135]

[66] D. Kempf, S. Müthing, and M. Böhling. *dune-opcounter*. https://gitlab.dune-project.org/dominic/dune-opcounter. 2019. — [106, 135]

[67] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam. "Designing Vector-friendly Compact BLAS and LAPACK Kernels". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017, 55:1–55:12. DOI: 10.1145/3126908.3126941. — [74]

[68] R. C. Kirby. "Algorithm 839: FIAT, a new paradigm for computing finite element basis functions". In: *ACM Transactions on Mathematical Software (TOMS)* 30.4 (2004), pp. 502–516. — [17, 30]

[69] R. C. Kirby and K. T. Thinh. "Fast simplicial quadrature-based finite element operators using Bernstein polynomials". In: *Numerische Mathematik* 121.2 (2012), pp. 261–279. DOI: 10.1007/s00211-011-0431-y. — [22]

[70] R. C. Kirby and A. Logg. "A compiler for variational forms". In: *ACM Transactions on Mathematical Software (TOMS)* 32.3 (2006), pp. 417–444. — [17]

[71] A. Klöckner. "Loo.Py: Transformation-based Code Generation for GPUs and CPUs". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 2014, 82:82–82:87. DOI: 10.1145/2627373.2627387. — [46, 51]

[72] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174. — [47]

[73] A. Klöckner, L. C. Wilcox, and T. Warburton. "Array Program Transformation with Loo.Py by Example: High-order Finite Elements". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY 2016. Santa Barbara, CA, USA: ACM, 2016, pp. 9–16. DOI: 10.1145/2935323.2935325. — [46]

[74] M. Kretz. *P0214: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: https://wg21.link/p0214. — [55]

[75] M. Kretz and V. Lindenstruth. "Vc: A C++ library for explicit vectorization". In: *Software: Practice and Experience* 42.11 (2012), pp. 1409–1430. DOI: 10.1002/spe.1149. — [53]

[76] M. Kronbichler and K. Kormann. "A generic interface for parallel cell-based finite element operator application". In: *Computers & Fluids* 63 (2012), pp. 135–147. DOI: 10.1016/j.compfluid.2012.04.012. — [21, 75, 130]

[77] R. Leißa, I. Haffner, and S. Hack. "Sierra: A SIMD Extension for C++". In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. Orlando, Florida, USA: ACM, 2014, pp. 17–24. DOI: 10.1145/2568058.2568062. — [53]

[78] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, et al. "ExaStencils: Advanced stencil-code engineering". In: *European Conference on Parallel Processing*. Springer. 2014, pp. 553–564. — [18]

[79] C. F. Loan. "The ubiquitous Kronecker product". In: *Journal of Computational and Applied Mathematics* 123.1 (2000). Numerical Analysis 2000. Vol. III: Linear Algebra, pp. 85–100. DOI: https://doi.org/10.1016/S0377-0427(00)00393-9. — [21, 76]

[80] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. DOI: 10.1007/978-3-642-23099-8. — [2, 17]

[81] A. Logg and G. N. Wells. "DOLFIN: Automated finite element computing". In: *ACM Transactions on Mathematical Software (TOMS)* 37.2 (2010), p. 20. — [17]

[82] D. B. Loveman. "Program improvement by source-to-source transformation". In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 121–145. — [16]

[83] F. Luporini, A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. Kelly. "Cross-loop optimization of arithmetic intensity for finite element local assembly". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2015), p. 57. — [17]

[84] A. C. I. Malossi, Y. Ineichen, C. Bekas, and A. Curioni. "Fast exponential computation on simd architectures". In: *Proc. of HIPEAC-WAPCO, Amsterdam NL* (2015). — [56]

[85] K. Martin and B. Hoffman. *Mastering CMake: a cross-platform build system*. Kitware, 2010. — [69]

[86] A. T. T. McRae, G.-T. Bercea, L. Mitchell, D. A. Ham, and C. J. Cotter. "Automated Generation and Symbolic Manipulation of Tensor Product Finite Elements". In: *SIAM Journal on Scientific Computing* 38.5 (2016), S25–S47. DOI: 10.1137/15M1021167. eprint: http://dx.doi.org/10.1137/15M1021167. — [32]

[87] J. M. Melenk, K. Gerdes, and C. Schwab. "Fully discrete hp-finite elements: Fast quadrature". In: *Computer Methods in Applied Mechanics and Engineering* 190.32-33 (2001), pp. 4339–4364. — [21]

[88] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (2017), e103. — [40, 46, 49]

[89] E. Mueller, P. Bastian, S. Müthing, and M. Piatkowski. "Matrix-free multigrid block-preconditioners for higher order Discontinuous Galerkin discretisations". In: *Submitted to Journal Of Computational Physics* (May 2018). arXiv: 1805.11930 [math.NA]. — [22, 24, 103]

[90] S. Müthing. "A flexible framework for multi physics and multi domain PDE simulations". In: (2015). — [12]

[91] S. Müthing, M. Piatkowski, and P. Bastian. "High-performance Implementation of Matrix-free High-order Discontinuous Galerkin Methods". In: *Accepted to Int. J. High Performance Computing Applications* (Jan. 2018). arXiv: 1711.10885 [math.NA]. — [3, 22, 24, 75]

[92] C. Nugteren and V. Codreanu. "CLTune: A generic auto-tuner for OpenCL kernels". In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE. 2015, pp. 195–202. — [99]

[93] K. B. Olgaard, A. Logg, and G. N. Wells. "Automated Code Generation for Discontinuous Galerkin Methods". In: *SIAM J. Sci. Comput.* 31.2 (2008), pp. 849–864. DOI: 10.1137/070710032. — [36]

[94] S. A. Orszag. "Spectral methods for problems in complex geometries". In: *Journal of Computational Physics* 37.1 (1980), pp. 70–92. DOI: 10.1016/0021-9991(80)90005-4. — [21]

[95] W. Pazner and P.-O. Persson. "Approximate tensor-product preconditioners for very high order discontinuous Galerkin methods". In: *Journal of Computational Physics* 354 (2018), pp. 344–369. DOI: https://doi.org/10.1016/j.jcp.2017.10.030. — [24]

[96] PDELab team. *dune-pdelab*. https://gitlab.dune-project.org/pdelab/dune-pdelab. 2019. — [135]

[97] PDELab team. *dune-pdelab-tutorials*. https://gitlab.dune-project.org/pdelab/dune-pdelab-tutorials. 2019. — [11, 124, 128]

[98] E. Phipps, M. D'Elia, H. C. Edwards, M. Hoemmen, J. Hu, and S. Rajamanickam. "Embedded ensemble propagation for improving performance, portability, and scalability of uncertainty quantification on emerging computational architectures". In: *SIAM Journal on Scientific Computing* 39.2 (2017), pp. C162–C193. — [75]

[99] C. Prud'Homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena. "Feel++: A computational framework for galerkin methods and advanced numerical methods". In: *ESAIM: Proceedings*. Vol. 38. EDP Sciences. 2012, pp. 429–455. — [18]

[100] QuantStack. *xsimd*. https://github.com/QuantStack/xsimd. 2019. — [53, 86]

[101] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. Kelly. "Firedrake: automating the finite element method by composing abstractions". In: *ACM Transactions on Mathematical Software (TOMS)* 43.3 (2016), p. 24. — [17]

[102] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. Kelly. "PyOP2: A high-level framework for performance-portable simulations on unstructured meshes". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 1116–1123. — [17]

[103] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007. — [19]

[104] G. Ren, P. Wu, and D. Padua. "Optimizing Data Permutations for SIMD Devices". In: *SIGPLAN Not.* 41.6 (2006), pp. 118–131. DOI: 10.1145/1133255.1133996. — [88]

[105] P. J. Roache. "Code verification by the method of manufactured solutions". In: *Journal of Fluids Engineering* 124.1 (2002), pp. 4–10. — [108]

[106] N. Shibata. "Efficient evaluation methods of elementary functions suitable for SIMD computation". In: *Computer Science-Research and Development* 25.1-2 (2010), pp. 25–32. — [56]

[107] M. Snir, W. D. Gropp, and P. Kogge. "Exascale research: preparing for the post-Moore era". In: (2011). — [18]

[108] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), p. 66. — [19]

[109] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly. "A study of vectorization for matrix-free finite element methods". In: *arXiv preprint arXiv:1903.08243* (2019). — [17, 75, 130]

[110] W. Taha. "Multi-stage programming: Its theory and applications". PhD thesis. Oregon Graduate Institute of Science and Technology, 1999. — [15]

[111] The DUNE developers. *The DUNE project website*. https://www.dune-project.org. Accessed: 2019-06-27. — [135]

[112] TOP500 Website. *TOP500 list November 2018*. https://www.top500.org/list/2018/11/. Accessed: 2019-06-1. — [20]

[113] U. Trottenberg, C. W. Oosterlee, and A. Schuller. *Multigrid*. Elsevier, 2000. — [23]

[114] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), p. 22. — [47]

[115] S. Verdoolaege. “isl: An integer set library for the polyhedral model”. In: *International Congress on Mathematical Software*. Springer. 2010, pp. 299–302. — [48]

[116] P. E. Vos, S. J. Sherwin, and R. M. Kirby. “From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low-and high-order discretisations”. In: *Journal of Computational Physics* 229.13 (2010), pp. 5161–5181. — [21]

[117] H. Wang, P. Wu, I. G. Tanase, M. J. Serrano, and J. E. Moreira. “Simple, Portable and Fast SIMD Intrinsic Programming: Generic Simd Library”. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP ’14. Orlando, Florida, USA: ACM, 2014, pp. 9–16. DOI: 10.1145/2568058.2568059. — [53]

[118] Z. J. Wang, K. Fidkowski, R. Abgrall, F. Bassi, D. Caraeni, A. Cary, H. Deconinck, R. Hartmann, K. Hillewaert, H. T. Huynh, et al. “High-order CFD methods: current status and perspective”. In: *International Journal for Numerical Methods in Fluids* 72.8 (2013), pp. 811–845. — [24]

[119] M. F. Wheeler. “An elliptic collocation-finite element method with interior penalties”. In: *SIAM Journal on Numerical Analysis* 15.1 (1978), pp. 152–161. — [8]

[120] S. W. Williams. *Auto-tuning performance on multicore computers*. University of California, Berkeley, 2008. — [99]

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor Prof. Dr. Peter Bastian for the unique opportunity I was given during the last years: I was able to pursue a self-chosen research project, although the outcome of this project was unclear in the beginning. I am very proud to have received this trust and hope that I am able to repay it with the contributions of this thesis.

Furthermore, I would like to acknowledge the Interdisciplinary Centre for Scientific Computing (IWR), HGS MathComp and the faculty of mathematics and computer science at Heidelberg University for providing the perfect institutional frame for the interdisciplinary work that I do.

This work was partly funded by the German Federal Ministry of Education and Research (BMBF) under grant 01IH16003C, which I am genuinely grateful for. The experimental part of this work was greatly supported by Olaf Ippisch from Technische Universität Clausthal and Christian Engwer from the University of Münster by providing me access to their Skylake and Knights Landing servers.

I would also like to thank Andreas Klöckner from the University of Illinois at Urbana-Champaign for his impressive work on the `loopy` project and the very fruitful discussions we had on its development. I am also very grateful to the entire DUNE community and its developers.

This thesis would not have been possible without the constant feedback of my colleague Steffen Müthing. Not only did he plant the original idea of integrating code generation into the DUNE framework into my head, he also gave valuable feedback at all stages of the development process. Also, his counselling on all matters from progamming languages to performance engineering was essential for the success of this project.

I also would like to show my appreciation to my other colleagues for creating a work environment that I enjoyed every single day. I want to specificially mention René Heß who was an indispensible discussion partner and gave a great boost to the project as soon as he joined its development.

Lastly, I would like to dedicate this thesis to my wonderful wife Caro and my daughter Paula.