Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications

CARSTEN UPHOFF and MICHAEL BADER, Technical University of Munich, Germany

The numerical solution of partial differential equations is at the heart of many grand challenges in supercomputing. Solvers based on high-order discontinuous Galerkin (DG) discretisation have been shown to scale on large supercomputers with excellent performance and efficiency if the implementation exploits all levels of parallelism and is tailored to the specific architecture. However, every year new supercomputers emerge and the list of hardware-specific considerations grows simultaneously with the list of desired features in a DG code. Thus, we believe that a sustainable DG code needs an abstraction layer to implement the numerical scheme in a suitable language. We explore the possibility to abstract the numerical scheme as small tensor operations, describe them in a domain-specific language (DSL) resembling the Einstein notation, and to map them to small General Matrix-Matrix Multiplication routines. The compiler for our DSL implements classic optimisations that are used for large tensor contractions, and we present novel optimisation techniques such as equivalent sparsity patterns and optimal index permutations for temporary tensors. Our application examples, which include the earthquake simulation software SeisSol, show that the generated kernels achieve over 50% peak performance of a recent 48-core Skylake system while the DSL considerably simplifies the implementation.

CCS Concepts: • Computing methodologies → Massively parallel and high-performance simulations; • Software and its engineering → Source code generation; Domain specific languages; • Applied computing → Earth and atmospheric sciences;

Additional Key Words and Phrases: Tensor operations, ADER-DG, finite element method, high-performance computing

ACM Reference format: Carsten Uphoff and Michael Bader. 2020. Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications. ACM Trans. Math. Softw. 46, 4, Article 34 (October 2020), 40 pages. https://doi.org/10.1145/3406835

1 INTRODUCTION

Solving partial differential equations (PDEs) is one of the pillars of computational science and engineering, and solving PDEs accurately on a computer is at the heart of many grand challenges in high-performance computing. Numerical methods may require billions of degrees of freedom.
for computing a solution with sufficient accuracy, hence highly scalable and energy-efficient codes are required. The implementation of a highly efficient software package poses difficulties: While, on the one hand, the solver might need to support several PDEs, several finite element types, or multi-physics simulations, on the other hand, developers have to deal with the subtleties of modern hardware architectures—for example, vector instruction latency or level 1 cache bandwidth. An optimised implementation is likely to require expert knowledge, it might not reflect the underlying numerical scheme, and the extension of the solver is a time-consuming task. Moreover, evolution of hardware architectures may require the adaptation of major parts of the code base or even a radical change of data structures.

Domain-specific languages and code generation have become popular tools, because many of the steps on the path from problem formulation to numerical scheme to code are repetitive and can be automated. For example, Alnæs et al. [2014] introduce the Unified Form Language (UFL) for the finite element method. The language allows to directly enter the problem formulation, the weak form of a PDE, and closely resembles the mathematical notation of variational problems. The origins of UFL lie in the FEniCS project [Logg et al. 2012], it has been adopted by the Firedrake project [Rathgeber et al. 2016], and is also the base of the DUNE-pdelab project [Kempf et al. 2018].

In another code generation approach, which we call the “small GEMM” approach, one starts with a fixed numerical scheme. Small tensor contractions are identified in the scheme and manually mapped to small General Matrix-Matrix Multiplication (GEMM) routines. Specialised code generators for small GEMMs exist, e.g., LIBXSMM [Heinecke et al. 2016b] or GiMMiK [Wozniak et al. 2016]. These code generators tailor GEMM routines for small block sizes. Moreover, the generated routines process input data “as is,” because the overhead of packing data (as with large GEMMs [Van Zee and van de Geijn 2015]) is too high for small GEMMs. The small GEMM approach is, for example, employed by the codes Nek5000 [Hutchinson et al. 2016], SeisSol [Heinecke et al. 2014], PyFR [Vincent et al. 2016], and EDGE [Breuer et al. 2017]. All of these codes are able to handle billions of degrees of freedom on recent petascale systems with high efficiency.

The UFL approach and the small GEMM approach differ greatly in the level of abstraction. In the former approach, the user only needs to specify the finite element spaces and the weak form. The details of the numerical scheme and the implementation are left to a compiler. In the latter approach, the numerical scheme is formulated by the user, tensor contractions are identified and are mapped manually to small GEMMs (or to other specialised code generators [Breuer et al. 2017]). But the essential link between the two approaches are sequences of small tensor contractions. This becomes evident from the COFFEE compiler [Luporini et al. 2015] or the intermediate representation GEM [Homolya et al. 2018], which are designed essentially to optimise small tensor contractions within the Firedrake software.

Many techniques for optimising sequences of tensor contractions have their root in the computational chemistry community, in particular within the Tensor Contraction Engine [Baumgartner et al. 2005]. Here, tensors are very large, i.e., so large that they do not fit into a node’s local memory but need to be distributed or stored to disk. The order of evaluation in a sequence of tensor contractions plays a large role, because the number of floating point operations for a naïve implementation might be magnitudes larger than an optimised implementation [Lam et al. 1997] or intermediate products might take up too much memory [Baumgartner et al. 2005]. Moreover, several techniques have been developed to implement binary tensor contractions, including nested-loop code, Loop-over-GEMM, Transpose-Transpose-GEMM-Transpose, and GEMM-like Tensor-Tensor multiplication [Springer and Bientinesi 2018, and references therein], and a compiler for tensor transpositions has been developed [Springer et al. 2017].

In this article, we are interested in a level of abstraction that lies between those of UFL and the manual mapping to small GEMMs, and which is not tied to a framework, not tied to a particular
numerical scheme, and not even tied to PDE solvers. Plainly spoken, an abstraction in the spirit of the Tensor Contraction Engine but for small tensor contractions. Based on our experience with the code SeisSol, we have extracted a set of assumptions about a typical target application:

- All tensors fit inside low-level caches. There is no need to trade off memory usage and operation count [Baumgartner et al. 2005].
- Parallelisation within a tensor contraction is not required.
- Data copies or index permutations at runtime may be amortised for large GEMMs [Goto and van de Geijn 2008; Springer and Bientinesi 2018], as these scale with \( \mathcal{O}(N^2) \) compared to \( \mathcal{O}(N^3) \), but they need to be avoided for small GEMMs [Heinecke et al. 2016b].
- All tensor shapes and respective sparsity patterns are known \textit{a priori}. Furthermore, spending a lot of time optimising individual tensor operations can be justified, as those are called millions of times by a solver [Uphoff et al. 2017].
- Sparsity can be dealt with explicitly, as it is feasible to compute the sparsity pattern of a tensor operation during the invocation of a code generator. There is no need to estimate the sparsity, as proposed by Lam [1999].
- Some optimisations are special to small GEMMs or DG methods, such as padding tensors for optimal vectorisation or tailored software prefetching schemes [Heinecke et al. 2016a].

Existing open-source software packages focus on binary tensor contractions [Li et al. 2015; Matthews 2018; Shi et al. 2016; Solomonik et al. 2013; Springer and Bientinesi 2018], GPUs [Nelson et al. 2015], only support tensors up to order 2 (matrices) [Spampinato et al. 2018; Spampinato and Püschel 2014; Uphoff and Bader 2016], or focus on loop transformations [Kempf et al. 2018; Luporini et al. 2015; Stock et al. 2011], where the latter lack support for sparse matrices in element-local operators and are to our understanding not designed for use with code generators for small GEMMs.

We therefore present Yet Another Tensor Toolbox (YATeTo). In YATeTo, tensor operations are described in a domain-specific language that is motivated by the Einstein convention. The compiler automatically eliminates superfluous operations in sparse tensor contractions, minimises the number of floating point operations (\textit{strength reduction}), and chooses appropriate data layouts for temporary tensors. Tensor contractions are mapped to Loop-over-GEMM, which internally calls specialised code generators for small GEMMs such as LIBXSMM. Our main focus are high-order DG and spectral element methods, but the toolbox itself may be used in any application that fits the assumptions stated above.

2 RELATED WORK

Many components of YATeTo build on previous algorithms and ideas. In this section, we summarise related work and necessary background.

2.1 High-level Language and Representation

The need for high-level languages for tensor contractions was already clear to A. Einstein, who wrote in the foundations of general relativity:

"Dafür führen wir die Vorschrift ein: Tritt ein Index in einem Term eines Ausdrucks zweimal auf, so ist über ihn stets zu summieren, wenn nicht ausdrücklich das Gegenteil bemerkt ist." [Einstein 1916]

"We therefore introduce the following rule: when an index appears twice in a term of an expression, one shall always sum over it, unless the opposite is noted explicitly.” (authors’ translation)
In addition to the importance of Einstein’s convention for mathematical notation, it serves as the basis for elegant domain-specific languages [Åhlander 2002; Nelson et al. 2015; Solomonik et al. 2013]. For example, in the Cyclops tensor framework one may write [Solomonik et al. 2013]:

\[ W["MnIj"] = V["MnEf"] \ast T["EfiJ"] ; \]

\[ Z["Abij"] = W["MnIj"] \ast T["AbMn"] ; \]

In the above example, summation over indices \( E \) and \( f \) is implied in the first statement and summation over \( M \) and \( n \) is implied in the second statement.

### 2.2 Tensor Terminology

We briefly define terms used throughout this article: A tensor is a multi-dimensional array on a number field. The tensor \( A \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d} \) is said to have dimension or order \( d \). Order \( d \) stipulates that \( A \) needs \( d \) indices, e.g., \( A_{ijkl} \) for \( d = 4 \). The size of tensor \( A \) is \( n_1 \times \cdots \times n_d \). We use shape synonymously with size when size is represented as \( d \)-tuple \((n_1, n_2, \ldots, n_d)\).

A fibre of tensor \( A \) is a vector obtained by fixing all indices but one [Kolda and Bader 2009]. E.g., the third fibre of the fourth-order tensor \( A \) is the vector \( A_{ij} \):

\[ \text{The size of an index is identified with the size of the corresponding fibre. That is, index } k \text{ in } A_{ijkl} \text{ has size } n_3. \]

### 2.3 Strength Reduction

Often there are many mathematically equivalent ways to implement tensor operations. Baumgartner et al. [2005] discuss the following motivating example, where each index has size \( N \):

\[ S_{abij} = \sum_{c,d,e,f,k,l} A_{acik} B_{befl} C_{dfjk} D_{cdel} = \sum_{c,k} \left( \sum_{d,f} \left( \sum_{e,l} B_{befl} D_{cdel} \right) C_{dfjk} \right) A_{acik}. \tag{1} \]

The first implementation corresponds to the naive implementation with 10 nested loops, which requires \( 4N^{10} \) operations. The second implementation saves intermediate results and only requires \( 6N^6 \) operations. Hence, the second implementation is four orders of magnitudes cheaper than the first implementation.

Lam et al. [1997] breaks down Equation (1) into two types of formulae. The first type is a binary multiplication formula that computes the outer product of two tensors. The second type is a unary summation formula that sums a tensor over a single index. E.g., the term \( \sum_{e,l} B_{befl} D_{cdel} \) may be broken down as follows: The innermost formula is the multiplication formula \( X_{beflcd} := B_{befl} D_{cdel} \). Note that the set of indices on the left-hand side is given by the union of the two sets of indices on the right-hand side. The multiplication formula is followed by the two summation formulae \( Y_{befcd} := \sum_l X_{beflcd} \) and \( Z_{bfcd} := \sum_e Y_{befcd} \).

The summation and multiplication formulae define a class of loop computations. Finding the sequence of formulae with minimum operation count is called strength reduction. Lam et al. [1997] show that strength reduction is NP-complete. However, it is feasible to exhaustively search all valid formulae if the number of terms is small enough.

### 2.4 Loop-over-GEMM (LoG)

Di Napoli et al. [2014] give a summary of tensor contraction classes and possible mappings to BLAS. In a binary contraction, the contraction is classified in the number of free indices of each input tensor. (That is, those indices that are not contracted.) The most interesting case is when
Fig. 1. This listing shows the tensor contraction (3) mapped to BLAS, assuming that \(a, b, c, d, e, f \in \{0, \ldots, 7\}\) and that a column-major layout is used. The GEMM operation is \(C := \alpha AB^T + \beta C\), where \(A\) is \(M \times K\), \(B\) is \(N \times K\), and \(C\) is \(M \times N\). LDA, LDB, and LDC give the number of doubles GEMM has to skip to reach the next column of \(A, B,\) and \(C\), respectively.

both tensors have at least one free index, as in this case a mapping to GEMM is always possible, if data are allowed to be copied or non-stride-one-GEMMs are considered.

Consider the following contraction:

\[
C_{abcd} := C_{abcd} + \sum_{e,f} A_{abef} B_{cdef}. \tag{2}
\]

Following Shi et al. [2016], we have the following options in a Loop-over-GEMM implementation: We may batch modes yielding a non-free index \(i\), denoted with \([i]\); we may flatten modes yielding a single free index, denoted with \((ij)\); and we may transpose modes, denoted with \(A^T_{ij}\). Note that transposes may only be applied to exactly two free modes.

A possible Loop-over-GEMM implementation of Equation (2) is

\[
C_{(ab)(cd)} := C_{(ab)(cd)} + \sum_{e,f} A_{(ab)[e][f]} B_{(cd)[e][f]}^T. \tag{3}
\]

That is, we loop over index \(e\) and call GEMM for subtensors of \(A\) and \(B\) inside the loop. An implementation using the standard BLAS interface is listed in Figure 1.

3 TENSORS IN DISCONTINUOUS GALERKIN METHODS

The discontinuous Galerkin (DG) method has become a popular method for solving hyperbolic PDEs. The DG method has several advantages compared to other classic methods, such as Finite Difference methods, Finite Volume methods, and continuous Finite Element methods: It simultaneously allows to handle complex geometries, achieves high-order convergence rates, and is explicit in time [Hesthaven and Warburton 2008]. Additionally, the high number of element-local operations and the small stencil make it a promising candidate to exploit the capabilities of modern supercomputing hardware. Before discussing the details of our tensor toolbox, we want to motivate that tensor contractions are ubiquitous in DG methods. Note that Einstein’s convention is used throughout this section.

As an example, we consider the following linear system of PDEs:

\[
\frac{\partial u_p}{\partial t} + A_{pq} \frac{\partial u_q}{\partial x} + B_{pq} \frac{\partial u_q}{\partial y} + C_{pq} \frac{\partial u_q}{\partial z} = 0. \tag{4}
\]

The vector \(u\) contains \(P\) space- and time-dependent functions, and we refer to it as vector of quantities or simply quantities. \(A, B,\) and \(C\) are (potentially) space-dependent matrices of size \(P \times P\).
Multiplying Equation (4) with a test function $\phi_k$, integrating over a domain $\Omega$, and integrating by parts yields the corresponding weak form

$$
\int_{\Omega} \phi_k \frac{\partial u_p}{\partial t} \, dV + \int_{\partial \Omega} \phi_k \left( n_x A_{pq} + n_y B_{pq} + n_z C_{pq} \right) u_q \, dS
- \int_{\Omega} \left( \frac{\partial \phi_k}{\partial x} A_{pq} + \frac{\partial \phi_k}{\partial y} B_{pq} + \frac{\partial \phi_k}{\partial z} C_{pq} \right) u_q \, dV = 0, \tag{5}
$$

where $(n_x, n_y, n_z)$ is the outward-pointing normal vector. From here a sequence of steps follows to obtain a semi-discrete form, such as partitioning the domain $\Omega$ in finite elements $\Omega_e$ (e.g., tetrahedra or hexahedra), introducing a numerical flux to weakly couple the finite elements, transforming the physical elements to a reference element, and introducing a polynomial approximation of the quantities on the reference element [Atkins and Shu 1998; Dumbser and Käser 2006; Hesthaven and Warburton 2008]. After these steps, the semi-discrete form is a system of ODEs, which may, for example, be solved with a Runge-Kutta scheme or the ADER approach [Dumbser and Käser 2006]. We skip most steps for simplicity and set $\Omega = \Omega_e$.

The quantities are approximated with $u_p = Q_{lp} \phi_l(x, y, z)$, $p = 1, \ldots, P$. (The sum over $l$ is implied.) The functions $\phi_l(x, y, z)$, $l = 1, \ldots, L$, are polynomial basis functions. The time-dependent $B \times P$ matrix $Q$ stores the basis expansion coefficients. By inserting the approximation of the quantities, we may write the second line of Equation (5) in the following way:

$$
Q_{1l} A_{pq} \int_{\Omega} \frac{\partial \phi_k}{\partial x} \phi_l \, dV + Q_{1l} B_{pq} \int_{\Omega} \frac{\partial \phi_k}{\partial y} \phi_l \, dV + Q_{1l} C_{pq} \int_{\Omega} \frac{\partial \phi_k}{\partial z} \phi_l \, dV.
\tag{6}
$$

The integrals in Equation (6) may be pre-computed and stored as (so-called) stiffness matrices $K_x$, $K_y$, and $K_z$ of size $B \times B$. The implementation in terms of GEMM is then given by the sequence of matrix chain products $K_x Q A^T + K_y Q B^T + K_z Q C^T$.

When one uses the unit cube $\Omega = [0, 1]^3$ as reference element, then it is possible to use a spectral basis, where the basis functions are given by $\phi_a(x, y, z) = \phi_{(a, b, c)}(x, y, z) = \psi_a(x)\psi_b(y)\psi_c(z)$. Note that we interpret $a$ as multi-index with $a = (a, b, c) \in \{0, \ldots, N\}^3$. Integer $N$ is the maximum polynomial degree. The basis expansion coefficients are stored in the 4-dimensional tensor $Q$, i.e., $u_p = Q_{abcq} \psi_a(x)\psi_b(y)\psi_c(z)$. We replace $\phi_k$ with $\phi_k$ in the second line of Equation (5), where $k = (k, l, m)$, we insert the spectral approximation $u_q = Q_{abcq} \psi_a \psi_b \psi_c$, and obtain

$$
\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \left( \frac{\partial \psi_k}{\partial x} \psi_l \psi_m A_{pq} + \psi_k \frac{\partial \psi_l}{\partial y} \psi_m B_{pq} + \psi_k \psi_l \frac{\partial \psi_m}{\partial z} C_{pq} \right) Q_{abcq} \psi_a \psi_b \psi_c \, dx \, dy \, dz
= K_{ka} M_{lb} M_{mc} A_{pq} Q_{abcq} + M_{ka} K_{lb} M_{mc} B_{pq} Q_{abcq} + M_{ka} M_{lb} K_{mc} C_{pq} Q_{abcq},
$$

where $K_{ij} := \int_{0}^{1} \frac{\partial \psi_i}{\partial x} \psi_j \, dx$ and $M_{ij} := \int_{0}^{1} \psi_i \psi_j \, dx$. We note here that the index order $abcq$ is chosen arbitrarily. In fact, the index permutations $qabc$, $aqbc$, and $abqc$ are equally viable. E.g., if $P$ is large and $N + 1$ is smaller than the SIMD vector width, then it might be beneficial to choose order $qabc$ such that index $q$ is contiguous in memory in column-major order. Otherwise, if $N + 1$ is large and $P$ is small, then either of $^Tm$ should be contiguous in memory.

Another need for small tensor contractions arises when one wants to solve multiple problems concurrently. The numerical scheme for the latter is obtained by adding a problem dimension $s$ to the degrees of freedom $Q_{lp}$. The 3D tensor $Q_{slp}$ then stores multiple degrees of freedom [Breuer

\footnote{Note that these matrices need to be only computed for the reference element in an actual implementation.}
Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications

The matrix chain product $K^TQA$ from Equation (6) then becomes the tensor contraction sequence $K_{kl}Q_{slq}A_{pq}$ and may be implemented using the Loop-over-GEMM approach, e.g., $(Q_{slq}A_{pq})_{sl[p]}(K_{kl})^T$.

As a final example, PDEs might have non-linear flux functions, too. E.g., a general scalar conservation law in one spatial dimension is given by $\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0$. In this case, the flux function is typically interpolated at a finite set of points, e.g., by employing a quadrature rule with points and weights $(\xi_i, w_i)$, $i = 1, \ldots, M$. The running example of this section becomes:

$$\int_0^1 \frac{\partial \phi_k}{\partial x}(x) f(u(x, t)) \, dx \approx \sum_{i=1}^M w_i \frac{\partial \phi_k}{\partial x}(\xi_i) f(u(\xi_i, t)) = w_i D_{ki} F_i,$$

where $D_{ki} := \frac{\partial \phi_k}{\partial x}(\xi_i)$ and $F_i := f(u(\xi_i, t))$. Einstein's convention does—strictly speaking—not cover the term $w_i D_{ki} F_i$, because index $i$ appears thrice. However, indices appearing more than twice are supported by our language (cf. Section 4.2). We note that evaluating $u(x, t)$ at quadrature points $\xi_i$ involves another tensor contraction, and that the flux function $f$ cannot be cast as tensor operation in general. Therefore, the evaluation of $f$ needs to be handled outside of the tensor toolbox.

4 YET ANOTHER TENSOR TOOLBOX

The input to our tensor toolbox is a domain-specific language (DSL), which naturally resembles the Einstein notation. An abstract syntax tree (AST) is derived from an expression. Subsequently, the tree is shaped using a sequence of visitors [Gamma et al. 1995]. Afterwards, we transform the AST to a control flow graph (CFG) to apply standard compiler techniques [Seidl et al. 2012]. Note that the CFG is very simple, as the DSL does not feature loops and branches. Finally, a code generator is called that generates either C++11 code or may invoke existing code generators or BLAS libraries. An overview is shown in Figure 2. YATeTo is open-source and available on www.github.com/SeisSol/yateto.

In the following, we introduce our DSL and elaborate algorithms and design choices of each of the three stages.

4.1 High-level Description of Tensor Operation

The DSL we created is embedded into Python 3. An embedded DSL has the advantage that we do not need our own parser and lexer. Furthermore, a user may use any feature of Python to form expressions—for example, including loops, branches, classes, and lambda expressions.

The basic data types are the classes Tensor and Scalar. A tensor definition includes a name and a shape tuple, e.g., as in the following:

- $A = \text{Tensor}(\text{"A"}, (N, N))$
- $B = \text{Tensor}(\text{"B"}, (N, N, N))$
- $w = \text{Tensor}(\text{"w"}, (N,))$
- $C = \text{Tensor}(\text{"C"}, (N, N))$

$A$ and $C$ are matrices of size $N \times N$, with $N = 8$, $B$ is a third-order tensor of size $N \times N \times N$, and $w$ is a vector of size $N$. From a Tensor, an object of type IndexedTensor may be derived by supplying an index string. The convention is that the length of the index string must be equal to

$\text{(N,)}$ is required to define a single-element tuple in Python.

Note that the trailing comma in $(N,)$ is required to define a single-element tuple in Python.
Fig. 2. Overview of YATeTo, showing the path from the high-level language to the generated code. The first stage (left box) operates on an abstract syntax tree, in the second stage (middle box) the representation is changed to a simple control flow graph without branches, and in the last stage (right box) different back-ends are called that output C++ or assembly.

the order of the tensor and that only lower- and upper-case letters may be used. Objects of type IndexedTensor may appear in expressions, e.g.,

\[ \text{kernel} = C[ij] \leftarrow 2.0 \cdot C[ij] + A[ij] \cdot B[ijk] \cdot w[k] \]

The above statement represents the operation:

\[ C_{ij} := 2 \cdot C_{ij} + \sum_{l=0}^{N-1} \sum_{k=0}^{N-1} A_{lj} B_{ikl} w_k \] or equivalently \( C := 2 \cdot C + (B \times_2 w) A, \)

where \( \times_n \) is the n-mode product. The variable kernel contains an AST built by overloading the binary operators \( *, +, \) and \( \leftarrow \) (assignment). Hence, standard Python rules apply for handling operator precedence and parentheses. For the presentation hereafter, we define a syntactically correct statement as following:

**Definition 4.1.** A statement in YATeTo is

\[ X[\alpha] \leftarrow y. \]

Large letters denote tensors (\( X \)), Greek letters denote indices (\( \alpha \)), and small letters denote arithmetic expressions (\( y \)). Indices are strings over an alphabet \( \mathcal{A} = \{a, \ldots, z, A, \ldots, Z\} \) without repeated symbols, that is, indices are elements of the set

\[ I := \{\beta \in \mathcal{A}^*: \beta_i \neq \beta_j \text{ whenever } i \neq j\}, \]

where \( \mathcal{A}^* \) is the set of all strings over the alphabet \( \mathcal{A} \).

---

Note that this limits the number of distinct indices that may appear in an expression to 52. We think that this is not an issue in practice, hence, we favour simplicity instead of generality.
Table 1. Summary of Operations on Indices

| Input | Operation | Returns | Example |
|-------|-----------|---------|---------|
| \( \alpha = \alpha_1 \ldots \alpha_n \in \mathcal{I} \) | set(\( \alpha \)) | \( \bigcup_{i=1}^{n} [\alpha_i] \) | set(\( kj \)) = \{i, j, k\} |
| \( \mathcal{B} = \{\beta_1, \ldots, \beta_m\} \subseteq \mathcal{A} \) | \( str(\mathcal{B}) \) | \( \beta_1 \ldots \beta_m \) | \( str(\{i, j, k\}) = ijk \) |
| \( \alpha \in \mathcal{I} \) | | | |
| \( \beta_1 \leq \cdots \leq \beta_m \) | \( set(\mathcal{B}) \) | | |
| \( \alpha \in \mathcal{I}, \mathcal{B} \subseteq \mathcal{A} \) | \( \alpha \setminus \mathcal{B} \) | Projection of \( \alpha \) on \( \mathcal{A} \setminus \mathcal{B} \) | \( ijkl \setminus \{j, k\} = i1 \) |
| \( \alpha \in \mathcal{I}, \beta \in \mathcal{I} \) | \( \alpha \sqcup \beta \) | \( \alpha(\beta \setminus set(\alpha)) \) | \( kij \sqcup jkli = kj1l \) |
| \( \alpha \in \mathcal{I}, b \in set(\alpha) \) | \( \alpha.\text{position}(b) \) | \( i \) such that \( \alpha_i = b \) | \( (ijk).\text{position}(k) = 3 \) |

The set \( \mathcal{I} \) is the set of strings over alphabet \( \mathcal{A} \) without repeated symbols. The lexicographical order is denoted with \( \leq \).

An arithmetic expression is either

1. an indexed tensor \( Y[\beta] \),
2. a sum of \( p \geq 2 \) arithmetic expressions \( x_1 + \cdots + x_p \),
3. or a product of \( q \geq 2 \) arithmetic expressions \( x_1 \ast \cdots \ast x_q \).

### 4.2 Semantics

Einstein’s convention is sometimes ambiguous. For example, shall one sum over \( i \) on the right-hand side of the expression \( A_i = B_i C_i \)? Or how shall one interpret an expression with an index appearing more than twice (e.g., \( A = B_i C_i D_i E_i \))? And what is the interpretation of \( A_i = B_{ji} \) or \( A_{ij} = B_{ji} + C_j \)?

In practice, additional conventions or annotations are required. The interested reader is referred to the paper of Åhlander [2002], who discusses possible conventions for a formal language.

In this section, we present a small set of rules that encode the conventions adopted in this work and allow the unambiguous interpretation of statements.

#### 4.2.1 Preliminaries

Three key components are required to set up the semantic rules: First, we define the iteration space for an index string \( \alpha \) as

\[
S(\alpha) = \times_{k=1}^{n} \{0, 1, \ldots, \text{size}(\alpha_k) - 1\},
\]

where \( \times \) is the \( n \)-ary Cartesian product operator. The size function \( \text{size} : \mathcal{A} \rightarrow \mathbb{N} \) is chosen according to the tensor shape. That is, if the \( d \)-dimensional tensor \( X \) has shape \( (a_1, \ldots, a_d) \in \mathbb{N}^d \), then writing \( X[\alpha] \) implies \( \forall i : \text{size}(\alpha_i) = a_i \). A statement is invalid when an index symbol appears in two or more indexed tensors and the tensor shapes imply different sizes for the same index symbol.

Second, we define projection functions to “access” the entries of a tensor. A projection function \( \pi_{\beta|\alpha} \) maps an element \( a = (a_1, \ldots, a_m) \), \( m = |\alpha| \), of the iteration space of \( \alpha \) onto an element \( b = (b_1, \ldots, b_n) \), \( n = |\beta| \), of the iteration space of \( \beta \), i.e.,

\[
\pi_{\beta|\alpha} : S(\alpha) \rightarrow S(\beta).
\]

It is defined as

\[
b = \pi_{\beta|\alpha}(a) = (a_{\alpha.\text{position}(\beta_1)}, \ldots, a_{\alpha.\text{position}(\beta_n)}).
\]

The position operation returns the position of a symbol in a string (cf. Table 1). The projection function is undefined if \( set(\beta) \not\subseteq set(\alpha) \). We require the inverse image of a projection
function, too:

\[ \pi_{\beta|\alpha}^{-1}(b) = \{ a \in S(\alpha) : \pi_{\beta|\alpha}(a) = b \}. \]

We give an example to clarify the purpose of projection functions.

**Example 4.2.** Consider the statement \( C[y] \equiv A[\alpha] \ast B[\beta] \) with \( y = ijk, \alpha = i1k, \) and \( \beta = j1. \) The shape of tensor A is \((2, 2, 2)\) and the shape of tensor B is \((2, 2)\). A straightforward implementation of the tensor contraction would require four loops over indices \( i, j, k, \) and \( l. \) In the loop body, we map indices \( \omega := \{ ijk1 \} \) to the tensors A and B with projection functions \( \pi_{\alpha|\omega}(w) = (w_1, w_2, w_3) \) and \( \pi_{\beta|\omega}(w) = (w_2, w_4), \) respectively. The inverse image of the projection function \( \pi_{\gamma|\omega} \) is used to determine all iterations that affect a particular entry \( C_c \):

\[ \pi_{\gamma|\omega}^{-1}(c) = \{ w \in S(\omega) : (w_1, w_2, w_3) = (c_1, c_2, c_3) \} = \{ c_1 \} \times \{ c_2 \} \times \{ c_3 \} \times \{0, 1\}. \]

Altogether, we write

\[ \forall c \in S(\gamma) : C_c := \sum_{w \in \pi_{\gamma|\omega}^{-1}(c)} A_{\pi_{\alpha|\omega}(w)} B_{\pi_{\beta|\omega}(w)}. \]

Above equation is equivalent to "summation over index l," as required by Einstein’s convention.

Third, we need to determine the summation indices. We obtain the set of all indices appearing in an arithmetic expression with

\[ U[X[\alpha]] = set(\alpha), \]

\[ U[x_1 + \cdots + x_p] = U[x_1] \cup \ldots \cup U[x_p], \]

\[ U[x_1 \ast \cdots \ast x_q] = U[x_1] \cup \ldots \cup U[x_q]. \]

The set of potential summation indices contains all indices that appear more than once, i.e.,

\[ S[x_1 \ast \cdots \ast x_q] = \bigcup_{1 \leq i, j \leq q, i \neq j} U[x_i] \cap U[x_j]. \]

4.2.2 Semantic Rules. Definition 4.1 describes the structure of a statement; here, we define its meaning.

**Definition 4.3.** The meaning of an assignment statement \( X[\alpha] \equiv y \) is

\[ \forall a \in S(\alpha) : X_a := [y|\alpha](a). \]

The “square bracket function” on the right-hand side depends on the type of the arithmetic expression \( y. \) For an indexed tensor, the evaluation function is

\[ [Y[\beta]|\alpha](a) = Y_{\pi_{\beta|\omega}(a)}. \]

The evaluation function for the “+”-operation of arity \( p \geq 2 \) is

\[ [x_1 + \cdots + x_p|\alpha](a) = [x_1|\alpha](a) + \cdots + [x_p|\alpha](a). \]

For the “*”-operation of arity \( q \geq 2, \) we define

\[ [x_1 \ast \cdots \ast x_q|\alpha](a) = \sum_{s \in \pi_{\alpha|\omega}(a)} [x_1|\alpha \sqcup \sigma](a) \cdots [x_q|\alpha \sqcup \sigma](a), \]

where \( \sigma := str(S[x_1 \ast \cdots \ast x_q]). \) (The \( \sqcup \) operation concatenates two strings and erases duplicate characters; cf. Table 1.)
4.2.3 Examples and Limitations. We may resolve the examples from the section’s beginning. No sum is implied in the statement $A[i] \leftarrow B[i] \ast C[i]$, because index $i$ appears on the left-hand side. The summation is placed at the outermost position when indices appear more than twice. E.g., $A[] \leftarrow B[i] \ast C[i] \ast D[i] \ast E[i]$ is interpreted as $A := \sum_i B_i C_i D_i E_i$. The statement $A[i] \leftarrow B[ji]$ is illegal, because the projection function $\pi_{i,j|i}$ is undefined. However, transposition is allowed, e.g., $A[i,j] \leftarrow B[j]$. Statements such as $A[i,j] \leftarrow B[j] + C[j]$ or $D[i,j] \leftarrow E[k] \ast F[k,j]$ require special attention. In the first case, the vector $C$ is implicitly casted into a matrix by replicating the vector. Similarly, an implicit broadcast from a vector to a matrix is required in the second case. While the semantic rules allow such statements, we believe that implicit broadcasts are rather counter-intuitive. Therefore, statements that require implicit broadcasts are forbidden and raise an error.

Last, we discuss a more complex example:

$$
A[i,j] \leftarrow A[ji] + B[i1k] \ast C[j1] \ast (D[km1] \ast E[mk1] + F[1k]).
$$

The set of potential summation indices of arithmetic expression $y_1$ is $\{i, k, l\}$. The sum is only taken over indices $k$ and $l$, because index $i$ appears on the left-hand side. Potential summation indices of $y_2$ are $\{i, k, m\}$, but the sum is only taken over index $m$ because indices $k$ and $i$ are bound beforehand.

4.3 Equivalent Sparsity Patterns

The notion of equivalent sparsity patterns (EQSPPs) and an algorithm to compute them was introduced in Uphoff and Bader [2016] for matrix chain products (MCP). It is defined as the minimal sparsity patterns of the involved matrices that leaves the result of an MCP unchanged. Note that we assume no cancellation [Cohen 1998]. That is, when an inner product of two vectors is computed (e.g., as part of a matrix matrix multiplication) and the two vectors have overlapping non-zeros, then the inner product is assumed to be always non-zero. The no-cancellation assumption allows us to work with Boolean tensors.

The concept of EQSPPs can be illustrated with the following example (where the entries may be either scalars or dense block matrices):

$$
\begin{pmatrix}
K_1 & 0 \\
Q_{11} & Q_{12} & A_1 \\
Q_{21} & Q_{22} & 0
\end{pmatrix} = K_1 Q_{11} A_1.
$$

Here, the multiplication with the first matrix removes $Q_{21}$ and $Q_{22}$ from the right-hand side and the multiplication with the third matrix removes $Q_{12}$ and $Q_{22}$. Thus, the equivalent sparsity pattern of the second matrix is only non-zero for the top-left block ($Q_{11}$).

In Uphoff and Bader [2016] an algorithm to compute EQSPPs for MCPs is presented, which is based on a graph-based representation. The extension of the MCP graph to general tensor operations is not straightforward, thus, we derive an algorithm that is not based on a graph in the remainder of this section.

4.3.1 Computation of EQSPPs for Tensor Operations. We discuss the computation of EQSPPs for statements of the form

$$
U[\alpha] \leftarrow T^1[\beta_1] \ast \ldots \ast T^q[\beta_q].
$$

The tensors $T^1, \ldots, T^q$ may be dense or sparse and we call the Boolean tensor $\Theta^i$ the sparsity pattern of $T^i$. We define equivalent sparsity patterns as following:

**Definition 4.4.** The Boolean tensors $\Theta^1, \ldots, \Theta^q$ are called EQSPPs w.r.t. Equation (15) if
(1) \( \hat{\Theta}^k \) is a Boolean tensor with the same shape as \( T^k \), that is,
\[
\forall b \in S(\beta_k) : \hat{\Theta}^k_b \in \{0, 1\}.
\]

(2) \( \forall a \in S(\alpha) : U_a = \hat{U}_a \), where \( \hat{U}[\alpha] \equiv \hat{T}^1[\beta_1] \ast \ldots \ast \hat{T}^q[\beta_q] \) and
\[
\forall k = 1, \ldots, q : \forall b \in S(\beta_k) : \hat{T}^k_b = \begin{cases} T^k_b & \text{if } \hat{\Theta}^k_b = 1 \\ 0 & \text{if } \hat{\Theta}^k_b = 0. \end{cases}
\]

(3) The number of non-zeros of \( \hat{\Theta}^k, k = 1, \ldots, q \), is minimal, that is, we cannot set a non-zero to zero without implying \( \exists a \in S(\alpha) : U_a \neq \hat{U}_a \) (assuming no cancellation).

Before going into the details, we sketch the underlying idea. The goal of EQSPPs is to get rid of all tensor entries that do not affect the result of a tensor operation. To find these irrelevant entries, we list all possible terms of a tensor operation. E.g., for Equation (14), we have terms \( K_1Q_{11}A_1, K_1Q_{12}A_2, K_2Q_{21}A_1, \) and \( K_2Q_{22}A_2 \). The guaranteed zeros \( K_2 = 0 \) and \( A_2 = 0 \) allow us to delete all but the first term from the list. To obtain the EQSPPs, we simply check which entries remain in the depleted list of terms.

We encode the “list of all terms” in an outer product tensor. For Equation (14) the outer product tensor is given by \( Z[1, j] \leftarrow K[1] * Q[1, j] * A[j] \). (Recall that no sum is implied here, as all summation indices appear on the left-hand side.)

We first show that finding EQSPPs for Equation (15) (which involves sums) may be recast into finding EQSPPs for the outer product tensor (which does not involve sums):

**Lemma 4.5.** The EQSPPs \( \tilde{\Theta}^1, \ldots, \tilde{\Theta}^q \) w.r.t. Equation (15) are identical to the EQSPPs \( \tilde{\Theta}^1, \ldots, \tilde{\Theta}^q \) w.r.t.
\[
Z[\omega] \leftarrow T^1[\beta_1] \ast \ldots \ast T^q[\beta_q],
\]
where \( \omega = \alpha \cup \sigma \) and \( \sigma = \text{str}(S[T^1[\beta_1] \ast \ldots \ast T^q[\beta_q]]) \).

**Proof.** We simply check that \( \tilde{\Theta}^1, \ldots, \tilde{\Theta}^q \) fulfil the three conditions in Definition 4.4 w.r.t. Equation (15):

1. Fulfilled trivially.
2. The entries of \( Z \) are given by
\[
\forall w \in S(\omega) : Z_w = T_{\pi_{1,1}(w)} \ast \ldots \ast T_{\pi_{q,1}(w)},
\]
\[
U \text{ is identical to } \hat{U} \text{ because}
\]
\[
\forall a \in S(\alpha) : U_a = \sum_{s \in \pi_{1,1}(a)} Z_s = \sum_{s \in \pi_{1,1}(a)} \hat{Z}_s = \hat{U}_a.
\]

3. Assume there exist \( \tilde{\Theta}^1, \ldots, \tilde{\Theta}^q \) with less non-zeros than \( \tilde{\Theta}^1, \ldots, \tilde{\Theta}^q \). Then there exists \( w \in S(\omega) \) such that \( Z_w = \tilde{Z}_w \neq \hat{Z}_w \), because otherwise \( \tilde{\Theta}^1, \ldots, \tilde{\Theta}^q \) would not be minimal for \( Z \). Entry \( \tilde{Z}_w \) is not cancelled in sums, therefore an index \( a \in S(\alpha) \) exists where \( \hat{U}_g \neq U_g \).

The outer product tensor \( Z \) contains the “list of all terms.” We want to recover the EQSPPs by checking the non-zero entries in \( Z \). Our main result is that one does not need to explicitly compute \( Z \) (which is potentially huge and might be infeasible to compute). Instead, one only needs Boolean tensor contractions.
Proposition 4.6. The EQSPPs w.r.t. Equation (15) are given by

\[ \forall b \in S(\beta_k) : \hat{\Theta}_b^k = \sum_{s \in \pi_{\beta_k}^{-1}(b)} \Theta_{\pi_{\beta_k}^{-1}(z)}^1 \cdots \Theta_{\pi_{\beta_k}^{-1}(z)}^q, \]  

where \( \omega = \alpha \sqcup \sigma \) and \( \sigma = \text{str}(S[T^1[\beta_1] \ast \cdots \ast T^q[\beta_q]]) \) and \( \Theta^l \) is the sparsity pattern of \( T^l, l = 1, \ldots, q \). The operations + and \cdot are identified with the operations \( \lor \) and \( \land \), respectively.

Proof. We show that the EQSPPs \( \hat{\Theta}_1, \ldots, \hat{\Theta}_q \) satisfy Definition 4.4 w.r.t. to the outer product \( Z \) (these are equivalent to the EQSPPs w.r.t. Equation (15) due to Lemma 4.5).

Condition 1 is satisfied, because addition and multiplication are taken as logical operations. Hence, \( \hat{\Theta}_k \) is a Boolean tensor.

For condition 2, we need to show that \( Z = \hat{Z} \). It is sufficient to show that the sparsity pattern of \( Z \), say \( \zeta \), is identical to the sparsity pattern of \( \hat{Z} \), say \( \hat{\zeta} \). The Boolean tensor \( \zeta \) is given by

\[ \forall s \in S(\omega) : \zeta_s = \Theta_{\pi_{\beta_k}^{-1}(z)}^1 \cdots \Theta_{\pi_{\beta_k}^{-1}(z)}^q, \]

and the Boolean tensor \( \hat{\zeta} \) is given equivalently by placing a hat on every theta. Using Equation (16) and above equation, we note that

\[ \forall b \in S(\beta_k) : \hat{\Theta}_b^k = \sum_{s \in \pi_{\beta_k}^{-1}(b)} \zeta_s. \]

From above equation follows

\[ \forall z \in S(\omega) : \hat{\zeta}_z = \Theta_{\pi_{\beta_k}^{-1}(z)}^1 \cdots \Theta_{\pi_{\beta_k}^{-1}(z)}^q = \left( \sum_{s \in \pi_{\beta_k}^{-1}(\pi_{\beta_k}^{-1}(z))} \zeta_s \right) \cdots \left( \sum_{s \in \pi_{\beta_k}^{-1}(\pi_{\beta_k}^{-1}(z))} \zeta_s \right) = \zeta_z, \]

due to \( \pi_{\beta_k}^{-1}(\pi_{\beta_k}^{-1}(z)) = \{z\} \) and the absorption law.

For condition 3, assume there exist \( \overline{\Theta}_1, \ldots, \overline{\Theta}_q \) with less non-zeros than \( \hat{\Theta}_1, \ldots, \hat{\Theta}_q \). Hence, for some \( m \in \{1, \ldots, q\} \) there exists an entry \( b \in S(\beta_m) \) such that \( \overline{\Theta}_b^m = 0 \) and \( \hat{\Theta}_b^m = 1 \). It holds that \( \sum_{s \in \pi_{\beta_m}^{-1}(b)} \zeta_s = 1 \), because otherwise \( \hat{\Theta}_b^m \neq 1 \). Thus, there exists an index \( c \in \pi_{\beta_m}^{-1}(b) \) with \( \zeta_c = 1 \). As \( \pi_{\beta_m}^{-1}(c) = b \), we have

\[ \zeta_c = \overline{\Theta}_{\pi_{\beta_m}^{-1}(c)}^1 \cdots \overline{\Theta}_{\pi_{\beta_m}^{-1}(c)}^{m-1} \cdot \overline{\Theta}_b^m \cdot \overline{\Theta}_{\pi_{\beta_m+1}^{-1}(c)}^1 \cdots \overline{\Theta}_{\pi_{\beta_q}^{-1}(c)}^q = 0 \neq \zeta_c = 1. \]

Therefore, \( \overline{\Theta}_1, \ldots, \overline{\Theta}_q \) violate condition 2. \( \Box \)

4.3.2 Implementation and Discussion. We have applied our EQSPP algorithm to an application example from Uphoff and Bader [2016], where we extended the degrees of freedom by an additional dimension (cf. Section 5.1). The original SPPs and EQSPPs can be seen in Figure 3. We observe that the non-zero entries of \( K \) and \( A \) induce additional zeros in \( \hat{Z} \) that may be disregarded when computing the tensor operation. For example, in the evaluation of \( I_{slq}K_{lk} \), we may restrict the loop ranges of \( q \) and \( l \).

The implementation of EQSPP-computation with Equation (16) is straightforward, as we require the same kind of tensor operations that we support within YATeTo, the only difference being that the tensors are Boolean. E.g., we may apply strength reduction (cf. Section 4.4) to reduce the amount of computations. Still, EQSPPs are more expensive to compute than to simply evaluate the original tensor operation. Nevertheless, the cost of computing EQSPPs is negligible in comparison to the possibly millions of times a kernel is called within a DG scheme.
Fig. 3. We show the kernel $Q_{skp} = I_{slq}K_{lk}A_{qp}$ (depicted with n-mode product, i.e., $Q = I \times_2 K^T \times_3 A^T$), which is similar to an application example in Uphoff and Bader [2016] but with an additional dimension added to the degrees of freedom. The degrees of freedom $(I, Q)$ are given as full tensor (Figure 3(a)), but we detect that large blocks in $I$ do not influence the final result, as they are multiplied with zeros in the tensor contractions (Figure 3(b)).

4.4 Strength Reduction

We already mentioned in Section 2.3 that finding the sequence of tensor operations with minimal operation count is an NP-hard problem. We implement the well-pruned enumeration procedure of Lam et al. [1997], which performs an exhaustive search but should be feasible for the number of terms appearing in practice. The same enumeration procedure may also be used when sparse tensors are involved, but in this case the sparsity patterns of intermediate results are required, or at least an estimate of the sparsity [Lam et al. 1999].

In YATetTo, we assume that tensors are small enough, such that it is feasible to explicitly compute all intermediate products during strength reduction. The number of operations is then determined in the following way [Lam et al. 1999]: For a multiplication formula $V[\ldots] = X[\ldots] \times Y[\ldots]$ the number of operations is equal to the number of non-zeros in $V$. For a summation formula $W[\ldots] = \sum_i Z[\ldots]$ the number of operations is equal to the number of non-zeros in $Z$ minus the number of non-zeros in $W$.

An example of an intermediate AST after strength reduction is shown in Figure 4(b).

4.5 Memory Layouts

Following Figure 2, the next step is to determine the memory layouts of the tensors. The memory layout influences the possibility to fuse indices in a Loop-over-GEMM implementation. Thus, the layout influences the cost function in Section 4.6 and needs to be fixed at this point already. In the following, we give an overview over the two classes of memory layouts that are currently supported.
Fig. 4. Overview of major stages during AST transformation of an Einsum node (left). After determining the equivalent sparsity patterns, the Einsum node is transformed during strength reduction, yielding an operation-minimal tree (middle). This tree is binary and consists of Product and IndexSum nodes only. Finally, a mapping to Loop-over-GEMM is found that minimises the cost function described in Section 4.6 (right).

4.5.1 Dense Layout. Let \( A \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d} \). A simple memory layout of a dense tensor is the "column-major" storage:

\[
A_{i_1 \ldots i_d} := A \left[ \sum_{k=1}^{d} i_k s_k \right],
\]

where the so-called stride is given by \( s_k = \prod_{l=0}^{k-1} n_l \) with \( n_0 := 1 \). That is, the tensor is stored linearly in a 1D array, such that the first index varies fastest and the last index varies slowest in memory. The stride \( s_k \) gives us the number of floating numbers we have to skip when we increase index \( i_k \) by one.

From Section 4.3, we expect that we also have tensors with large zero blocks, where it would be wasteful to store the full tensor. Hence, our next evolution is the bounding-box column-major storage:

\[
A_{i_1 \ldots i_d} := A \left[ \sum_{k=1}^{d} (i_k - b_k) t_k \right],
\]

where \( t_k = \prod_{l=0}^{k-1} (B_l - b_l) \) and \( 0 \leq b_k < B_k \leq n_k \). This memory layout models the case where \( A \) is non-zero within the index set \([b_1, B_1) \times \cdots \times [b_d, B_d)\) and zero otherwise.

Finally, it might be beneficial to ensure aligned memory access. That is, the number of bytes in the first fibre should be divisible by the architecture’s SIMD vector length. In some cases, one has to add artificial zeros to the first fibre, hence, we allow to enlarge the bounding interval \([b_1, B_1)\) as follows [Uphoff and Bader 2016]:

\[
[b_1', B_1') := [b_1 - b_1 \bmod v, B_1 + (v - B_1 \bmod v) \bmod v),
\]

where \( v \) is the number of floating point values that fit into a SIMD vector (e.g., using AVX-512 one sets \( v = 8 \) for double precision or \( v = 16 \) for single precision).

We conclude the presentation of dense layouts with a possible pitfall of using bounding boxes and alignment. It might be beneficial to fuse multiple indices when mapping a tensor contraction to a Loop-over-GEMM (see Section 2.4). Fusing indices \( i_a \ldots i_b \) is always possible when [Shi et al. 2016]

\[
\forall i \in [a, b) : t_{i+1} = n_i t_i.
\]
Other cases would require more involved compatibility conditions. For example, in the GEMM $C_{ijkl} = A_{ijm}B_{mkl}$, where artificial zeros are added to fibre $i$ in $A$ and fibre $k$ in $B$, then one may only fuse $(ij)$ and $(kl)$ whenever $C$ has the same number of artificial zeros in fibres $i$ and $k$. Otherwise, one needs temporary storage and an additional data copy. To avoid possible data copies and complications arising in subsequent optimisation steps, we only allow fused indices when Equation (17) is fulfilled. Conversely, aligned layouts or bounding box layouts have to be considered carefully, as they may prohibit fusing indices, which leads to smaller (and possibly less efficient) GEMMs.

4.5.2 Compressed Sparse Column (CSC) Layout. Sparse matrices may appear in discontinuous Galerkin methods, e.g., the stiffness matrices are sparse when an orthogonal set of basis functions is used. We have a limited support for the CSC format: CSC matrices may appear in GEMM or Loop-over-GEMM calls but only in a sparse $\times$ dense or a dense $\times$ sparse GEMM.

4.5.3 Other (Sparse) Formats. It might be surprising that we describe Equivalent Sparsity Patterns in great detail but currently do not offer a genuine sparse tensor memory layout. First, we want to point out that if an EQSPP induces a large zero block, then we are able to exploit this block using the bounding box memory layout or we may let a LoG operate only on a subtensor. Second, to the authors’ knowledge, an efficient code generator or efficient routines for general small sparse tensors do not currently exist.

4.6 Optimal Index Permutations

The strength reduction step converts each Einsum node (Figure 4(a)) to a binary tree consisting only of IndexSum and Product nodes (Figure 4(b)). In a first step, we identify contractions in the tree, which are (multiple) IndexSum nodes followed by a single Product node. A sub-tree that was identified to be a contraction is replaced by a new node of type Contraction. For example, we might end up with the (sub-)tree shown in Figure 5.

Each Contraction node shall now be mapped to an implementation as Loop-over-GEMM. In general, the mapping to GEMM is not unique, but several implementations are possible (e.g., Shi et al. [2016], Table II). Springer and Bientinesi [2018, Listing 7] discuss an algorithm to list all possible Loop-over-GEMM implementations. Their algorithm assumes that the order of all indices are known. In our case, we have temporary tensors in which an order of indices is not prescribed. (E.g., $T^{(1)}$, $T^{(2)}$, and $T^{(3)}$ in Figure 5.) The index order has to be chosen carefully, as it decides possible mappings to GEMM and there are even cases where non-unit stride GEMMs are required [Shi et al. 2016, Table II]. Moreover, the index order of a temporary result influences possible mappings to GEMM in nodes that involve the temporary result.
4.6.1 Optimality. We aim to automatically select the best index order in an optimisation pass. To do so, we need to predict the overall execution time. The accurate prediction of execution time is challenging, though, even for a single BLAS kernel [Peise and Bientinesi 2012] and in particular if cache effects need to be considered [Peise et al. 2015]. We therefore assume that it is infeasible or impractical to come up with an accurate execution time model for a complete AST, in particular in an optimisation setting where we need to evaluate the execution time model quite often.

We therefore consider cost models as a proxy for execution time. The class of cost functions we are interested in can be informally stated as following:

\[
\text{cost} = \text{cost of root node} + \text{cost of children.}
\]

The above allows to model the cost of a node in a “standard setting.” E.g., if we consider two alternative index permutations where one permutation leads to a call to a large GEMM and the other permutation leads to a call to a small GEMM, then we would prefer the large GEMM, as these are typically more efficient. Clearly, these kinds of cost models exclude “non-standard settings,” i.e., those where one choice is favourable to another due to uncommon circumstances (such as an L1i cache overflow).

4.6.2 Optimal Index Permutations. We now present an algorithm to find the best index order based on a cost function structured as in Section 4.6.1.

Our approach to select the index orders of temporary tensors is to solve the following discrete optimisation problem: Let \( V(r) \) be the set of vertices of an AST with root \( r \). We denote the set of children of a vertex \( v \) with \( C(v) \), and the set of descendants of a vertex \( v \) with \( D(v) = V(v) \setminus \{v\} \). For each vertex \( v \), we have a set of permissible index permutations \( P_v \). We introduce variables \( x_v \), for all \( v \in V(r) \), where \( x_v \in P_v \) i.e., variable \( x_v \) contains the index permutation for vertex \( v \). The Cartesian product of all permissible index permutations is called the configuration space. The optimisation problem is to find a configuration that minimises a cost function over all possible configurations, that is,

\[
c^* = \min_{x_r \in P_r, x \in S} W(x_r, x), \quad \text{where } S = \bigtimes_{v \in D(r)} P_v. \tag{18}
\]

The function \( W \) is the cost model, informally given in Section 4.6.1. Our formal definition of \( W \) for a tree with root \( r \) is

\[
W(x_r, x) = w_r \left( x_r, (x_c)_{c \in C(r)} \right) + \sum_{c \in C(r)} W(x_c, (x_d)_{d \in D(c)}). \tag{19}
\]

Note that we use a tuple builder notation to form the input tuple \( x \). For example, if node \( r \) has children \( C(r) = \{c_1, c_2\} \), where the children are ordered by \( c_1 < c_2 \), then \( (x_c)_{c \in C(r)} = (c_1, c_2) \). The functions \( w_r \) (“cost of root node”) measure the cost of the operation at vertex \( r \). We give a detailed definition of \( w_r \) in Section 4.6.3.

We split the minimisation in two stages. In the first stage, we minimise over the root variable and its children; in the second stage, we minimise over all other variables, i.e., the variables associated with the vertices in the set \( G(r) = \bigcup_{c \in C(r)} D(c) \). One can then show that

\[
c^* = \min_{x_r, (x_c)_{c \in C(r)}} \min_{(x_d)_{d \in G(r)}} W(x_r, (x_d)_{d \in D(r)})
\]

\[
= \min_{x_r, (x_c)_{c \in C(r)}} \left( w_r \left( x_r, (x_c)_{c \in C(r)} \right) + \sum_{c \in C(r)} f_c(x_c) \right),
\]
where we introduced

\[ f_c(y) = \min_{(x_d)_{d \in D(c)}} W(y, (x_d)_{d \in D(c)}). \] (20)

The sub-problem introduced in Equation (20) can be interpreted as finding the optimal configuration for a sub-tree, assuming that the index permutation of the root node is fixed. In fact, \( c^* = \min_{x_r \in \mathcal{P}_r} f_r(x_r) \). Such problems are said to have an optimal substructure, because the optimal solution can be constructed out of optimal solutions to sub-problems [Cormen et al. 2009].

Dynamic programming is a commonly used approach for problems with optimal substructure. We consider a bottom-up dynamic programming algorithm that works in the following way: The AST is traversed in post-order. For each vertex \( v \), we solve Equation (20) for every permissible index permutation \( x_v \in \mathcal{P}_v \). The minimum cost as well as a minimising configuration is memoised in a dictionary. If vertex \( v \) is a leaf node, we simply evaluate all index permutations in \( \mathcal{P}_v \). For internal nodes, we evaluate all configurations in \( \mathcal{P}_v \times (\times_{w \in \mathcal{C}(v)} \mathcal{P}_w) \). To evaluate the cost, we evaluate the function \( w_v \) and for the sub-problems, we look up the memoised costs. (The latter are available due to the post-order traversal.) The runtime of this algorithm can be bound with \( \mathcal{O}(N(n!)^{1+c}) \), where \( N \) is the number of vertices in an AST, \( n \) is the maximum number of indices in a vertex (and hence \( n! \) is the maximum size of each permissible index permutation set), and \( c \) is the maximum number of children.

4.6.3 Cost Function. The missing pieces in the last section are the cost functions \( w_r \). Here, we choose a simple heuristic. Our basic assumption is that our ASTs will consist mostly of Loop-over-GEMM (LoG) nodes and that those will dominate the runtime. Further assumptions are listed in the following:

- Non-unit stride GEMMs are inferior to unit stride GEMMs.
- Transposes of \( A \) (or \( B \)) in the GEMM \( AB \) should be avoided when using column-major (row-major) layout. Transposes of \( B \) (or \( A \)) should be avoided due to missing support in code generation back-ends.
- Large GEMMs are more efficient than small GEMMs, i.e., one should fuse as many indices as possible.

From these assumptions, we define the cost of a LoG as the 4-tuple \((s, l, r, -f)\), where \( s \) is the number of required slices with non-unit stride, \( l \) is the number of left-transposes (assuming column-major layout), \( r \) is the number of right-transposes, and \( f \) is the number of fused indices. Cost comparison is based on lexicographic comparison of \((s, l + r, -f)\), where the lower number of left-transposes is deciding when two 3-tuples are equal.

We use the algorithm of Springer and Bientinesi [2018] to list all feasible LoG implementations. That is, the cost function for a LoG node with root \( x_r \) and children \( x_{c_1} \) and \( x_{c_2} \) is

\[ w_r(x_r, (x_{c_1}, x_{c_2})) = \text{MinLoG}(x_r, x_{c_1}, x_{c_2}), \]

where MinLoG selects a LoG implementation with minimum cost according to our cost model, or infinite cost if no implementation exists.

4.6.4 Discussion. The cost function we employ is clearly limited, as it is a heuristic and makes sense for LoG nodes only. However, choosing another cost function does not change the dynamic programming scheme as long as the cost function is structurally equivalent to Equation (19). For example, one could swap our LoG-cost by a runtime model based on micro-benchmarks, using the methods developed by Peise and Bientinesi [2012].

The algorithm needs to visit every node only once, hence the algorithm is feasible also for large ASTs. Problematic are high-dimensional tensors (say, large \( n \)), as the number of permutations is \( n! \).
and we need to check all of them. But, as stated in the introduction, we assume that our tensors fit into low-level caches, which constrains $n$. E.g., $n = 6$ is the maximum dimension we may choose such that a $4^n$ tensor (double precision) still fits into a 32 KiB L1 cache. In other words, YATeTo is not designed for high-dimensional tensors.

An example of an AST after mapping to LoG can be seen in Figure 4(c).

### 4.7 Prefetching

The library LIBXSMM, a generator for small matrix-matrix multiplications, allows software prefetching instructions to be included in the generated assembler code [Heinecke et al. 2016b]. This may improve performance, particularly on Intel Knights Landing architecture [Heinecke et al. 2016a; Uphoff et al. 2017]. A possible prefetching strategy is to insert `vprefetch1` instructions for a matrix $B$ after writes to the result matrix $C$, where the memory offset calculated for the matrix $C$ are also used for the matrix $B$. The rationale is that $B$ has the same or a similar shape as $C$.

The `FindPrefetchCapabilities` visitor determines the number of bytes that may be prefetched for every node of the kernel. For a LoG-node, this is equal to the number of bytes in the result tensor. (Other nodes have no prefetch capability, but this may be added if appropriate code generators are available.) Tensors that shall be prefetched during the execution of a kernel may be attached to the kernel. `AssignPrefetch` then greedily assigns each to-be-prefetched tensor $P$ to one of the nodes, such that the number of bytes of $P$ matches the prefetch capability of the node.

### 4.8 Control Flow Graph

Following Figure 2, we convert the AST to a CFG representation [Seidl et al. 2012]. The aim here is first to obtain a standardised sequential representation for the later code generation step, and second to manage temporary buffers or to avoid them at all.

The data structure of the CFG are program points (vertices), which save the current state of the kernel, and actions (edges), which transform the kernel’s state. An action may be either an assignment or a plus-equals operation. The left-hand side is always a tensor. The right-hand side may be either a tensor or one of the operations modelled in the AST (such as LoG or Product). The right-hand side may be augmented by multiplication with a scalar.

The individual optimisation steps use standard techniques from compiler optimisation [Seidl et al. 2012], thus, we only briefly summarise them in the following:

1. **MergeScalarMultiplications**: The actions $A = f(\ldots); B = \alpha A$ are replaced by $B = \alpha f(\ldots)$.
2. **LivenessAnalysis**: Determine live variables [Seidl et al. 2012].
3. **SubstituteForward**: If possible, replace $A$ by $\text{tmp}$ after action $A = \text{tmp}$.
4. **SubstituteBackward**: If possible, replace left-hand side $\text{tmp}$ by $A$ if there follows an action $A = \text{tmp}$.
5. **RemoveEmptyStatements**: Remove statements such as $A = A$.
6. **MergeActions**: Try to merge actions, e.g., merge $A = f(\ldots)$ and $B += A$ to $B += f(\ldots)$, when there is no intermediate action that depends on $A$.
7. **DetermineLocalInitialization**: Greedy map of temporary buffers to temporary variables. If buffers are used for multiple variables, set the buffer size to the maximum size required by its assigned variables.

To illustrate the individual steps, we present matrix multiplication as an example:

\[ C[ij] \leftarrow C[ij] + 0.5 \times A[ik] \times B[kj] \]
A simple, but correct, traversal of the corresponding AST yields the sequential CFG shown in Figure 6(a). This inefficient CFG is transformed to a single call to GEMM (with $\alpha = 0.5$ and $\beta = 1$), as shown in Figure 6(e).

4.9 Code Generation

The final step is to generate machine code. In principle, the aim is to re-use existing code generators, such as LIBXSMM [Heinecke et al. 2016b] or PSpaMM [Brei 2018; Wauligmann and Brei 2019], math libraries such as Eigen [Guennebaud et al. 2010] or Blaze [Iglberger et al. 2012], or flavours of BLAS. Besides, YATeTo generates C++-glue-code, and it is able to generate generic fallback code whenever an efficient implementation is not available. The fallback code consists of nested loops and is architecture-independent, but we expect the performance to be rather poor and compiler-dependent.

Operations are divided into four types: copy, scale, add, index sum, product, and log. The first type is a combination of the BLAS routines COPY and AXPY, and may be implemented in terms of these two functions. The product and index sum types correspond to the multiplication formulae and summation formulae presented in Section 2.3. Having a generic implementation for these two types ensures that we cover a large class of tensor operations [Lam et al. 1997]. The log type implements Loop-over-GEMMs.

Our concept requires that it should be simple to include additional code generators, depending on the type of operation and on the architecture. We use the factory method pattern [Gamma et al. 1995] to distinguish between code generators. For each type, a factory method may return a different code generator, depending on the operation’s description (e.g., dense or sparse matrices) and the architecture. For the LoG type, YATeTo offers a generic C++-loop-based implementation, which internally calls another factory method for GEMM. But our structure would also allow to call specialised LoG generators, such as the ones developed by Breuer et al. [2017] for contractions involving 3D tensors and matrices.

4.10 Application Interface

A class is generated for every kernel. The class has an argument-free execute function and pointers to the input and output tensors of the kernel are public members, where the name of a member...
variable is given by the name of the tensor. So to invoke a kernel, one has to create a kernel object, set the pointers to the input and output tensors, and call the execute function. A kernel class also stores the following information as static members: The minimal number of flops,⁴ also sometimes called non-zero flops, and the number of flops the implementation requires, also sometimes called hardware flops. The number of non-zero flops is computed by means of strength reduction with the sparse cost function in Section 4.4, and the hardware flops are returned by the code generators. These flop counters are useful to evaluate the performance of the kernels.

In addition to the kernel classes, a unit test is generated for every kernel. The unit test compares the optimised kernel to the naive implementation of the same kernel. If the Frobenius norm of the difference of both implementations matches up to a tolerance, the unit test passes. (We allow a tolerance due to finite arithmetic imprecision caused by different summation orders.)

Moreover, a class is generated for every tensor, where information about the memory layout is stored. The most useful static member, which is declared \texttt{constexpr}, is the number of required floating point numbers. This information may be used to allocate stack or heap memory for a tensor. For convenience, each tensor class contains a function that returns a view object for a memory region. Using a view object $V$, the user may access a d-dimensional tensor with \texttt{operator()}, i.e., $V(i_1, \ldots, i_d)$. Tensors whose entries are known at compile time (e.g., stiffness matrices) may be stored within the generated code as static array, already formatted according to the selected memory layout.

To integrate YATeTo into the build process of an application, users need to have Python 3 and NumPy installed. The generated code follows C++11 syntax and depends on a small header-only library, i.e., the generated code itself requires only a recent C++-compiler. The kernels may also be called from C or Fortran by manually wrapping the kernel calls in a C function. We plan to generate a C and Fortran interface automatically in the future.

5 APPLICATION TO AN ADER-DG METHOD

We integrated YATeTo in two codes to evaluate practicability of our design, as well as to evaluate the performance and optimisation opportunities. Both codes employ a discontinuous Galerkin method with ADER time-stepping based on the discrete Cauchy-Kovalewski procedure [Käser et al. 2007]. Furthermore, both codes solve a linear PDE in two or three dimensions of the form of Equation (4). The differences lie in the PDE (elastic wave equation, acoustic wave equation) and the underlying finite elements (tetrahedra, rectangular cuboids).

In the following, we do not derive the numerical schemes but only indicate changes w.r.t. literature references to stay within the scope of this article. We validated both codes with a plane wave problem (e.g., Käser et al. [2007]) and checked that the code converges (for an appropriately chosen time-step) and that the observed convergence order matches the theoretical order of the scheme. Moreover, we tried to keep the amount of indices to a minimum by mostly omitting dependencies on space and time. We use Einstein’s convention for subscript indices, but not for superscript indices.

5.1 SeisSol

The earthquake simulation code SeisSol (www.github.com/SeisSol/SeisSol) solves the seismic wave equation with support for elastic [Dumbser and Käser 2006], viscoelastic [Käser et al. 2007], and viscoplastic rheological models [Wollherr et al. 2018]. The code operates on unstructured tetrahedral meshes and the quantities are approximated with Dubiner’s basis functions. (The elastic

⁴Strictly speaking, the term “minimal” is incorrect due to Strassen’s algorithm or other bounds on the number of operations. That is, minimal is understood as minimal w.r.t. the cost function defined in Section 4.4.
model presented here has nine quantities.) In essence, one may write the computational kernels as matrix chain products of small matrices, where small means that the matrices’ fibres have at most a size of $\mathcal{B} = (N+3)$. Integer $N$ is the maximum polynomial degree of the basis functions and $N+1$ is the theoretical convergence order. SeisSol already makes use of the code generator tailored for matrix chain products, which simplifies the implementation of the various rheological models [Uphoff and Bader 2016].

For elastic rheological models, the code achieves about 50% of peak performance on compute clusters based on Intel’s Haswell or Knight’s Landing architecture [Heinecke et al. 2016a; Uphoff et al. 2017]. Moreover, the authors of Breuer et al. [2017] investigate the possibility to fuse multiple simulations in a single run, using the same ADER-DG method as SeisSol and an elastic rheological model. They report a speedup of 2.1× for a fourth-order scheme when fusing eight simulations in one run (compared to eight individual simulations). Their main optimisation ingredient is a specialised code generator for tensor contractions involving 3D tensors and sparse matrices.

5.1.1 ADER-DG. The numerical scheme of SeisSol is described in detail in the literature [Dumbser and Käser 2006; Käser et al. 2007; Uphoff et al. 2017]. Here, we focus on the tensor operations and state the ADER-DG scheme only in the compact form of Equations (21) to (24) for an elastic rheological model. The equations already include multiple simulations [Breuer et al. 2016a]. Multiple simulations are easily modelled using Einstein’s convention, as one simply has to add another index to the degrees of freedom, which we have denoted with a blue $s$.

The first step in the numerical scheme is to compute the time-derivatives of the degrees of freedom $Q$, using a Cauchy-Kovalewski procedure:

$$\mathcal{D}_{s\text{kp}}^{(\delta+1)} := \tilde{K}_{kl}^s \mathcal{D}_{slq}^\delta A_{pq}^s + \tilde{K}_{kl}^\eta \mathcal{D}_{slq}^\delta B_{pq}^s + \tilde{K}_{kl}^\varepsilon \mathcal{D}_{slq}^\delta C_{pq}^s, \quad \text{where} \quad \mathcal{D}_{s\text{kp}}^0 := Q_{s\text{kp}}^{\mu\tau}. \quad (21)$$

The indices $\mu$ and $\tau$ denote the space-time element and index $\delta$ denotes the order of the time-derivative. The matrices $\tilde{K}$ are sparse $\mathcal{B} \times \mathcal{B}$ matrices, and $A^s, B^s, C^s$ are sparse $9 \times 9$ matrices, which are element-dependent linear combinations of the Jacobians $A, B, C$ from Equation (4). For every simulation, one needs to store a $\mathcal{B} \times \mathcal{B}$ matrix for the degrees of freedom $Q$ and the derivatives $\mathcal{D}$, yielding $\mathcal{S} \times \mathcal{B} \times 9$ tensors, where $\mathcal{S}$ is the number of simulations.

The evolution in time is predicted with a Taylor expansion without considering an element’s boundaries. In the DG-scheme, the time-integral over a time-step with size $\Delta t$ is required, which is computed as

$$I_{s\text{kp}} := \sum_{\delta=0}^{N} \Delta t^{\delta+1} (\delta+1)! \mathcal{D}_{s\text{kp}}^\delta. \quad (22)$$

The time-integral is inserted into the DG-scheme, yielding the following update scheme for the degrees of freedom:

$$Q_{s\text{kp}}^{(\tau+1)} := Q_{s\text{kp}}^{\mu\tau} + \tilde{K}_{kl}^s I_{slq}^\delta A_{pq}^s + \tilde{K}_{kl}^\eta I_{slq}^\delta B_{pq}^s + \tilde{K}_{kl}^\varepsilon I_{slq}^\delta C_{pq}^s + \sum_{\sigma=4}^4 \tilde{R}_{km}^\sigma I_{lm}^\delta R_{pq}^\sigma (A^\sigma_{pq}^s) + \sum_{\sigma=4}^4 \tilde{R}_{km}^\sigma f_{mn}^{\text{Rot}(\sigma,\mu)} L_{in}^\delta \hat{I}_{pq}^\text{Face}(\sigma,\mu) (A^\sigma_{pq}^s). \quad \quad (23)$$

The matrices $\tilde{K}$ are sparse $\mathcal{B} \times \mathcal{B}$ matrices, $\tilde{R}, \hat{R}$, and $R$ are sparse $\mathcal{B} \times \mathcal{B}$ matrices, where $\hat{\mathcal{B}} = (N+2)$. And $f$ are sparse $\mathcal{B} \times \hat{\mathcal{B}}$ matrices. $A^+$ and $A^-$ contain the solution to the 1D Riemann problem, which is solved along the four sides of a tetrahedron, and these are dense $9 \times 9$ matrices.

ACM Transactions on Mathematical Software, Vol. 46, No. 4, Article 34. Publication date: October 2020.
Fig. 7. Exemplary code that models vanishing coefficients in the Cauchy-Kovalewski procedure from Equation (21). First a partial AST is built and stored in `derivativeSum`. Then the first two visitors in the transformation process shown in Figure 2 are used to obtain the sparsity pattern of derivative $D_{i}$. Finally, a tensor using the derived sparsity pattern is defined and a complete AST is added to the code generator.

5.1.2 Equivalent Sparsity Patterns. The matrices $\hat{K}$ contain large zero blocks, especially only the first $\tilde{B}$ columns are non-zero. The zero blocks are consequence of Dubiner’s basis functions. YATeTo automatically determines that the EQSPP of $I$ is a $S \times \tilde{B} \times 9$ subtensor. Hence, the contraction $I_{(s)lq}(\hat{K})^T_{kl}$ is mapped to a GEMM of size $(M, N, K) = (S, B, \tilde{B})$ instead of a GEMM of size $(S, B, B)$.

The sparsity pattern of $\tilde{K}$ is equivalent to the sparsity pattern of $\hat{K}$ and it has a staircase structure. The effect is that derivative $D^h$ has only $(N-d+3)/3$ non-zero coefficients per quantity [Breuer et al. 2014b]. We may set the sparsity patterns of the derivatives $D^h$ accordingly to exploit the vanishing coefficient—or we use existing visitors of YATeTo to simultaneously define the derivative kernels and determine the sparsity patterns of the derivatives automatically, as shown in Figure 7.

5.1.3 Strength Reduction. By solving the matrix chain multiplication order problem, the number of required flops in Equation (24) can be reduced from $O(N^6)$ to $O(N^5)$ [Uphoff et al. 2017]. The optimal matrix chain order $\hat{R}_{ka}(\hat{I}_{ab}(R_{lb}I_{lq}))A^T_{pq}$ is reproduced by the implementation of the strength reduction algorithm. Interestingly, for 8–32 fused simulations, the order of evaluation $(\hat{R}_{ka}A_{pq})(\hat{I}_{ab}(R_{lb}I_{lq}))$ is optimal, found automatically by strength reduction.

5.1.4 Optimal Index Permutations. We take Equation (24) as example, but note that the same discussion can be applied to the other kernels. From YATeTo, we obtain the following mappings to LoG for either a single simulation or for multiple simulations (Greek letters denote temporary tensors):

\[ K^\alpha_{kl} = \int \frac{\partial \phi_k}{\partial \alpha} \phi_l dV / \int \phi_k \phi_k dV \] (no sum implied and $\alpha \in \{\xi, \eta, \zeta\}$). Basis function $\phi_l$ of degree $n$ is orthogonal to polynomials of degree $n - 1$ or smaller (by construction of the Dubiner basis). Thus, $\deg(\phi_k) \leq \deg(\phi_l) \Rightarrow K^\alpha_{kl} = 0$.

\[ \tilde{K}^\alpha = M^{-1}K^\alpha \] and $\hat{K} = M^{-1}K^\alpha$, where $M$ is the diagonal mass matrix. $K^\alpha$ is the weak discrete derivative operator and $(K^\alpha)^T$ is the strong discrete derivative operator.
We observe that the optimal index permutation algorithm from Section 4.6 fulfils the goals of the cost function: Non-unit stride GEMMs are not present, transposes are kept to a minimum, and indices are fused if possible.

In SeisSol, the implementation is in fact a bit different: The $A^-$ and $R$ matrices (as well as the star matrices and the $A^+$ matrices) are stored transposed, such that a transpose-free scheme is obtained. For multiple simulations, we can also obtain a transpose-free scheme in the following manner: The $A^-$ matrices (as well as the star matrices and the $A^+$ matrices) are stored transposed as for a single simulation, but for the other matrices, we face the opposite situation: The $R$ matrices are stored in normal order, and we have to store $\hat{R}$ and $f$ transposed as well as $\hat{K}$ and $\tilde{K}$ for the other kernels.

5.1.5 Findings. YATeTo is able to reproduce the major optimisations from SeisSol: Zero-blocks are exploited and the optimal matrix chain multiplication order is found. For multiple simulations, zero blocks are also exploited automatically, strength reduction revealed the optimal—and different—evaluation order, and a transpose-free scheme is found by inspection of mappings to GEMM produced by YATeTo.

5.2 LinA

The code LinA was developed for education purposes and contains a basic implementation of the ADER-DG method. LinA solves the linearised equations of acoustics in two dimensions [LeVeque 2002], which involve the three quantities pressure and particle velocities in horizontal and vertical direction. A uniform mesh with rectangular elements is used.

Our goal here is to evaluate the application of YATeTo to a DG-spectral-element-like method (cf. Kopriva [2009], Chapter 5.4, for an introduction to DG-SEM). As briefly introduced in Section 3, quantities are represented using a tensor product basis, i.e., $q_p = Q_{lmpn} \psi_l(x) \psi_m(y) \psi_n(z)$. An advantage of DG-SEM is that integrals over the reference elements may be split in a tensor product of 1D integrals. E.g., the computation of the gradient in the weak form can be done with $O(N^{d+1})$ instead of $O(N^{2d})$ computations per element, where $d$ is the number of spatial dimensions. On the downside, one needs to store $(N + 1)^d$ degrees of freedom per quantity instead of $\binom{N+d}{d}$ per quantity to obtain the same theoretical order of convergence.

We implemented the ADER-DG-SEM method in LinA using YATeTo (www.github.com/TUM-I5/LinA) and extended the code to three dimensions, which implies an additional quantity or four quantities in total. For the 1D basis functions $\psi_i$, we use a nodal basis with Legendre-Gauss-Lobatto points [Hesthaven and Warburton 2008]. In the following, we are going to abuse notation and use the same symbols for tensors as in SeisSol, even though the tensors are of different shape and have different sparsity patterns.

We remark that the restriction of LinA to rectangular domains limits its applicability to real-world problems (heterogeneous material parameters are supported, though). However, ADER-DG for acoustics may be extended to curved geometries, which again involves many small tensor
5.2.1 Numerical Scheme. The first step in LinA, as in SeisSol, is to predict the element-local evolution in time:

\[
D^\delta_{xyp} := \tilde{K}_x D^\delta_{lyzq} A^\delta_{pq} + \tilde{K}_y D^\delta_{xmqz} B^\delta_{pq} + \tilde{K}_z D^\delta_{xyzn} C^\delta_{pq}, \quad \text{where} \quad D^0_{xyp} := Q^\xi_{xyp}.
\]

\[
I_{xyp} := \frac{\Delta t^{\delta+1}}{(\delta + 1)!} D^\delta_{xyp}.
\]

Indices \( \xi, \nu, \zeta \) denote the location of an element on the grid and \( \tau \) is the time index. The matrices \( \tilde{K} \) are \((N + 1) \times (N + 1)\) matrices and \( A^*, B^* \), and \( C^* \) are sparse \( 4 \times 4 \) matrices. Both \( Q \) and \( D \) are \((N + 1) \times (N + 1) \times (N + 1) \times (N + 1) \times 4 \) tensors.

The weak derivatives are evaluated afterwards:

\[
Q^\xi_{xyp} := Q^\xi_{xyp} + \tilde{K}_x I_{lyzq} A^\xi_{pq} + \tilde{K}_y I_{xmqz} B^\xi_{pq} + \tilde{K}_z I_{xyzn} C^\xi_{pq}.
\]

In contrast to SeisSol, the neighbour exchange is not based on volume data. Instead, it is sufficient to consider the boundary nodes. We copy the boundary nodes to contiguous storage for each side, such that we need to store six 3D tensors instead of a single 4D tensor. In YATeTo, we represent the “copy” operation by the following matrix-vector products:

\[
X^\sigma_{xyp} := F^\sigma_{lyzq}, \quad Y^\sigma_{xyp} := F^\sigma_{xmqz}, \quad Z^\sigma_{xyp} := F^\sigma_{xyzn},
\]

where \( F^\sigma = \delta_{i0} \) for \( \sigma \in \{\text{left, bottom, back}\} \), and \( F^\sigma = \delta_{iN} \) for \( \sigma \in \{\text{right, top, front}\} \). The 2D boundary nodes are then used to apply the numerical fluxes:

\[
\hat{Q}^\xi_{xyp}^{(\tau + 1)} := Q^\xi_{xyp} + \sum_{\sigma \in \left\{\text{left, top, back}\right\}} \hat{F}_x^\sigma \left( X^\sigma_{yq} (A^\sigma_{pq}) + X^\text{Neigh}(\sigma, \xi, \nu, \zeta) (A^{-\sigma}_{pq}) \right)
\]

\[
+ \sum_{\sigma \in \left\{\text{bottom, top}\right\}} \hat{F}_y^\sigma \left( Y^\sigma_{xzq} (A^\sigma_{pq}) + Y^\text{Neigh}(\sigma, \xi, \nu, \zeta) (A^{-\sigma}_{pq}) \right)
\]

\[
+ \sum_{\sigma \in \left\{\text{bottom, back}\right\}} \hat{F}_z^\sigma \left( Z^\sigma_{xyq} (A^\sigma_{pq}) + Z^\text{Neigh}(\sigma, \xi, \nu, \zeta) (A^{-\sigma}_{pq}) \right).
\]

5.2.2 Implementation. Mapping the numerical scheme to our DSL is simple, as one may almost copy the above formulation in Einstein notation. E.g., the last line in Equation (29) can be written as following:

\[
\text{Q('xyzp')} <= \text{Q('xyzp')} + \text{Fhat[sigma]['z']} \* \backslash
\text{(Z['xyz'] \* Aplus['pq'] + ZNeigh['xyz'] \* Aminus['pq'])}
\]

Furthermore, by inspecting the generated implementation, we decided to store the matrices \( A^*, B^*, C^*, A^-, A^- \) transposed and also a transposed copy of \( \tilde{K} \) and \( \tilde{K} \), which leads to a transpose-free scheme. The eventual C++-code for the last line in Equation (29) can be seen in Figure 8. There, a generic nested loop-code implements the outer product \( Q_{xyzp} := \text{tmp}_{xyzp} \hat{F}_z + Q_{xyzp} \). In principle, one might also map this operation to BLAS, e.g., in terms of the level 2 routine GER, which implements a rank-1 update of a matrix. That is, one could implement the outer product as a Loop-over-GER, i.e., \( Q_{(xyz)}[p] := \text{tmp}_{(xyz)[p]} \hat{F}_z + Q_{(xyz)}[p] \). In YATeTo such an implementation corresponds to adding another back-end. However, we leave the implementation and performance evaluation of Loop-over-GER for future work.
Fig. 8. Sample C++-code generated in LinA. Indices in tensor contractions are fused such that only a call to GEMM is required. The GEMMs (prefix `pspamm`) are separate functions that contain inline assembler code and are generated specifically for the kernel. The nested loops implement an outer product and were generated using the generic fallback code generator.

5.3 Towards Further Applications: Integration with Higher Abstraction Layers

The application examples in this chapter show that the implementation of a numerical scheme with YATeTo’s DSL is straightforward for the most part. The time-consuming task of writing down the numerical scheme in the first place is left to the user. To simplify this task, one could use the Unified Form Language as a front-end language for the end-user to generate a representation of tensor operations compatible with YATeTo.

Such a modular structure is adopted in the Firedrake project [Homolya et al. 2017, 2018; Luporini et al. 2015]. Here, several compiler stages are used where the tensor algebra language GEM [Homolya et al. 2018] is used as intermediate representation. Therefore, one might realise the combination of UFL with YATeTo by supplying a suitable Python wrapper from the GEM language to YATeTo’s language.

6 IMPLEMENTATION ASPECTS AND PERFORMANCE RESULTS

In this section, we thoroughly evaluate the performance of the applications presented in Section 5. From the results in this section, we ultimately want to answer the following questions: First, is our tensor toolbox able to reproduce SeisSol’s performance for the special case of matrix chain products? Second, can we achieve similar performance improvements as Breuer et al. [2017] by using multiple simulations, but using our generic approach? Third, do we still get high performance when changing the PDE, the finite element type, and the basis functions?
6.1 Hardware and Software Environment

We run all our performance tests on two recent Intel architectures:

**KNL** Intel Xeon Phi 7250 on Stampede 2, "Knights Landing," single socket configuration with 68 cores, 1.4 GHz nominal frequency, 1.6 GHz single-core turbo frequency, theoretical peak performance of 3.0 TFLOPS using double precision or 6.1 TFLOPS using single precision (at 1.4 GHz), 1 MiB private L2 cache per tile (shared between two cores).

**SKX** Intel Xeon Platinum 8174 on SuperMUC-NG, "Skylake," dual socket configuration with 24 cores per socket, 3.1 GHz nominal frequency, 2.7 GHz AVX-512 all-core turbo frequency, theoretical peak performance of 4.1 TFLOPS using double precision or 8.3 TFLOPS using single precision, 1 MiB private L2 cache per core, 1.375 MiB shared L3 cache per core (non-inclusive), STREAM triad bandwidth of 204 GB/s.

In each experiment, threads are pinned to cores using the `KMP_AFFINITY` or `OMP_PLACES` environment variable. On SKX, we use 48 threads and pin each thread to exactly one core. For simultaneous multithreading (SMT), we use a compact pinning with 2 threads per core. On KNL, we basically employ the same strategy with a maximum of 2 threads per core, but we leave the first tile free for the operating system. Thus, we use 66 threads without SMT and 132 threads with SMT. Furthermore, on KNL, we use the cache memory mode for SeisSol and the flat memory mode for LinA, where memory is bound to MCDRAM using `numactl`.

Our software is compiled with the Intel C++-compiler (version 18). We generate our GEMM kernels with LIBXSMM (version 1.10, [Heinecke et al. 2016b]) and PSpaMM (commit f59f98d, [Wauligmann and Brei 2019]), which both offer highly tuned GEMMs for KNL and SKX. LIBXSMM is used in static mode (instead of the just-in-time-compilation mode), i.e., code is generated at compile-time. We remark that the generated file `subroutine.cpp`, which contains kernels from LIBXSMM and PSpaMM, is compiled with option `-mno-red-zone`. Otherwise, the compiled code might be wrong due to a combination of (1) the compiler does not inspect inline assembly, (2) the generated code might modify the stack (`pushq`, `popq`), (3) Linux calling conventions.

If not stated otherwise, we report the minimum runtime over at least five runs.

6.2 Sparse Matrices

PSpaMM originated from a student research project for dense \( \times \) sparse matrix multiplications [Brei 2018]. The starting point of the project is a clone of the almost optimal register-blocked LIBXSMM kernels for KNL. In principle, sparse matrix multiplication is then implemented by removing every instruction that either is a multiplication with zero or becomes unnecessary (such as loads of never-used data). Control structures such as the row indices array and the column pointer array are completely unrolled.

An issue with unrolling control structures is that code is also data, which needs to be fetched from memory or cache. An experiment with random sparse matrices by Brei [2018], using a predecessor of PSpaMM, suggests that the generated dense \( \times \) sparse matrix multiplication is always faster than its dense \( \times \) dense counterpart—as long as the number of non-zeros stays below a threshold such that code stays in the L1i cache. The threshold can be estimated by dividing the L1i cache

\footnote{We could not find the specification of the frequency scaling of the 8174 model [Intel Corporation 2020]. The system’s peak performance is claimed to be 26.3 PFLOPS on 6,336 nodes (https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG), which would imply a peak AVX-512 frequency of 2.7 GHz. A test measuring the time of around 54B `vfmadd231pd` instructions (using alternating `zmm` registers) indicates that a peak frequency of 2.79 GHz is possible. In this article, we adopt the vendor’s claim of 2.7 GHz, but we encourage the reader to scale the results with a peak frequency that may be more appropriate.}
The figure shows performance experiments with SeisSol’s kernels. We compare architectures (KNL, SKX), precision (SP=single, DP=double), orders (O2–O7), and the number of simultaneous simulations (1–32). The letter R denotes the reference version [Uphoff et al. 2017]. Dark bars show non-zero GFLOPS and light bars denote hardware GFLOPS. The term non-zero here refers to the minimal number of floating point operations, i.e., if sparsity is exploited perfectly, and the term hardware refers to the actual number of floating point operations. A dark shade of blue indicates that two threads per physical core are used, a light shade of blue indicates that one thread per physical core is used. Numbers to right of bars show speedups, if one compares S simultaneous simulations to S individual simulations. E.g., 8 simulations on SKX / DP for order 6 are 1.4× more efficient than the reference implementation, one thread per physical core is used, and the performance is close to 1,400 GFLOPS. The benchmark results are tabulated in Appendix A.

size by the size of a fused multiply-add instruction, as a fused multiply-add instruction is generated for every non-zero in the sparse matrix.

The sparse × dense case is more involved [Breuer et al. 2014a], the fundamental issue being that the standard outer product formulation for matrix multiplications forces to vectorise over sparse vectors. In SeisSol, sparse × dense kernels did not improve time-to-solution on KNL [Heinecke et al. 2016a].

As a consequence, we employ the following rule: We use a sparse memory layout when a sparse matrix is multiplied from the right, and we use a dense memory layout when a sparse matrix is multiplied from the left.

6.3 SeisSol: Performance

In this section, we present and discuss the performance results for SeisSol shown in Figure 9. We note that SeisSol’s performance reproducer is used [Uphoff et al. 2017]. The latter calls the same kernels as SeisSol, but the reproducer operates on random initial data and the cell neighbouring relations are random.
Table 2. SeisSol: Number of Floating Point Operations (Average) and Data Traffic (Estimated) per Element

| Order | Num. sims. | HW-FLOP | Data traffic | AI     |
|-------|------------|---------|--------------|--------|
| 2     | 1          | 28,566  | 7.78 kB–8.64 kB | 3.3–3.7 |
|       | 8          | 56,368  | 21.9 kB–28.8 kB  | 2–2.6  |
| 4     | 1          | 171,234 | 15.8 kB–20.2 kB  | 8.5–10.8|
|       | 8          | 503,491 | 86.4 kB–121 kB   | 4.2–5.8|
| 7     | 1          | 1,653,552 | 48.1 kB–66.2 kB | 25–34.4|
|       | 8          | 5,297,274 | 344 kB–490 kB  | 10.8–15.4|

As SeisSol uses unstructured tetrahedral meshes, the data traffic is difficult to predict. Therefore, data traffic and arithmetic intensity (AI) is stated for the best case and the worst case. The numbers are exemplarily given for orders 2, 4, and 7 on SKX / DP. Note that the data traffic is composed as follows: 576 B are due to loading \( A^*, B^*, C^* \), and \( Q \) is loaded twice and stored twice. \( I \) is loaded once and stored once. \( I^{\text{Neigh}} \) is loaded 1–4 times on average.

We investigate single precision arithmetic as well as double precision arithmetic. (As YATeTo supports single and double precision, we take an agnostic viewpoint on necessary precision and simply generate and test both precisions.) Performance is measured in terms of non-zero flops and hardware flops. The important measure here is non-zero flops, as it is independent of the choice of sparse and dense memory layouts and as such represents time-to-solution. However, the non-zero flops measure assumes perfect usage of sparsity patterns and we see it as too optimistic w.r.t. hardware. Hence, we include hardware flops as measure of exploitation of the hardware’s capabilities.

In Figure 9, we observe that the results for single simulations (row “1”) closely match the performance of the reference (row "R") over all orders, architectures, and precisions. We conclude that the special case of matrix chain multiplications is well handled within YATeTo. We also see that there is a large gap between non-zero flops and hardware flops. The latter is due to handling sparse matrices as dense, which minimises time-to-solution according to an auto-tuning procedure [Heinecke et al. 2016a]. For multiple simulations (rows "8," "16," and "32"), most sparse matrices are multiplied from the right such that the favourable dense \( \times \) sparse case is obtained. In our experiments, we observe that multiple simulations (compared to the same amount of individual simulations) yield a speedup of 1.1×–3.9× for DP and 1.1×–6.5× for SP. Furthermore, we observe an increase in non-zero peak performance from 1.1%–16.9% to 6.7%–27.5%. Specifically, a non-zero peak efficiency of 19.6% for KNL O4 DP is obtained, which is within a few percent of the non-zero peak efficiency obtained in Breuer et al. [2017].

A simple roofline model agrees with the observations. In Table 2, we exemplarily show the number of floating point operations per element and the estimated memory transfer per element. We see that the arithmetic intensity for a single simulation is higher than the arithmetic intensity of multiple simulations, because exploiting sparsity reduces the ratio of floating point operations

---

脚注说明：

8. 我们注意到 Uphoff et al. [2017] 和 Breuer et al. [2017] 的方案不完全相同，原因在于它们对边界的处理方式不同。然而，整体的方案是相同的。

9. 在计算内存转移时，我们假设每个独立的张量（\( K, \hat{K}, \hat{R}, \hat{R}, R \)，和 \( f \)）处于缓存中。此外，我们假设 \( I^{\text{Neigh}} \) 总是被缓存为最好的估计（每个元素访问 \( I^{\text{Neigh}} \) 一次），而我们假设 \( I^{\text{Neigh}} \) 永远不被缓存为最坏的估计（每个元素访问 \( I^{\text{Neigh}} \) 四次）。
Fig. 10. Comparison of simulation and reference solution. Shown is the particle velocity in x-direction of the 9th receiver of the LOH.1 benchmark (see http://www.sismowine.org/model/WP2_LOH1.pdf). A total of 16 simulations ran simultaneously in single precision, where the 0th simulation has the same seismic moment as the reference. Other simulations \((u_1 - u_{15})\) have a reduced moment (see text). The relative seismogram misfit for the 0th simulation is shown in \(E_{u_0}\).

to transferred bytes here. Note that the number of (hardware) floating point operations increases by a factor of 2.0–3.2 instead of a factor of 8.

There seems to be no consistent pattern whether SMT improves performance or whether it does not. The relative performance increase due to enabling SMT (or disabling SMT) is at most 22.8% on KNL and at most 5.3% on SKX, so the impact of SMT is rather minor. The exact reasons for the observed behaviour are unclear to us.

6.4 SeisSol: Validation

In Section 6.3 a performance reproducer is used to evaluate SeisSol’s performance. Here, we evaluate the correctness of the implementation using the well-established Layer Over Halfspace 1 (LOH.1) benchmark [Day et al. 2003]. The model description as well as the reference solution are taken from the SeISmic MOdeling Web INterfacE (www.sismowine.org).

We ran a total of 8 or 16 simulations simultaneously on a mesh with 1.1M elements and Order 6. The 0th simulation has a seismic moment of \(M_0 = 10^{18}\) Nm, as required by the benchmark. Other simulations have a seismic moment of \(M_0/(1 + s)\) where \(s\) is the simulation number, i.e., \(M_0/2, M_0/3, M_0/4,\) and so on. Due to the PDEs being linear, we expect that the amplitude of particle velocities is reduced by the same factor. Figure 10 shows that the 0th simulation matches the reference well, with a low relative seismogram misfit. The other simulations have their amplitude reduced, as expected. We note that the relative seismogram misfit is identical up to three digits, independent of the number of simulations or floating point precision.

The measured performance on eight SKX nodes is shown in Table 3, where we evaluated single simulations and multiple simulations, SP and DP, as well as Global Time Stepping (GTS) and Local Time Stepping (LTS). SMT is enabled or disabled according to Figure 9, but only 47 cores for computation are used, as one core is left for a communication thread. We observe that we obtain a lower performance in comparison to the performance reproducer (see column % HW_proxy). We do not attempt a detailed comparison of the performance reproducer with the real application, however, it is interesting to note that for multiple simulations we “lose” less performance than for a single simulation (e.g., 84% vs. 94% for GTS-SP). The latter is a hint that the remaining serial part of SeisSol might be responsible for the lower performance, as the serial part is mostly independent of the number of simulations but the parallel part becomes more expensive for more simulations. LTS is less efficient due to smaller loop lengths, but pays off in time-to-solution. In total, the expected speedup of multiple simulations is reproduced (1.6× for GTS-DP, 2.0× for GTS-SP).
Table 3. Summary of Performance Results for the LOH.1 Benchmark Using 8 SKX Nodes and O6

| Precision | LTS | Mode | NZ-GFLOPS | HW-GFLOPS | % HW proxy | Time/Sim. [min] |
|-----------|-----|------|-----------|-----------|------------|----------------|
| DP        | No  | 1    | 524       | 1,618     | 83         | 42.1          |
| DP        | No  | 8    | 839       | 1,148     | 92         | 26.6          |
| DP        | Yes | 1    | 480       | 1,485     | 76         | 17.8          |
| DP        | Yes | 8    | 752       | 1,030     | 83         | 11.5          |
| SP        | No  | 1    | 840       | 3,217     | 84         | 26.2          |
| SP        | No  | 16   | 1,719     | 2,559     | 94         | 13.0          |
| SP        | Yes | 1    | 789       | 3,023     | 79         | 10.8          |
| SP        | Yes | 16   | 1,433     | 2,134     | 78         | 6.0           |

The performance is reported per node.

![Fig. 11. We show performance results of a DG-SEM method implemented in the code LinA. We compare architectures (KNL, SKX), precision (SP=single, DP=double), orders (8, 16, 24, 32), and number of spatial dimensions (2D, 3D). Dark bars show non-zero GFLOPS, and light bars denote hardware GFLOPS. The colour indicates the employed code generators as well as the usage of the CSC memory layout for some matrices.](image_url)

6.5 LinA

While LinA uses the same ADER-DG scheme as SeisSol, the implementation is very different due to the use of tensor basis functions and cuboid elements. The main difference is that the arithmetic intensity only grows linearly with polynomial degree, instead of cubic growth as in SeisSol. But due to the use of a nodal basis, most of the matrices in the scheme are dense or almost dense and sparsity mainly comes in due to the coefficient matrices of the PDE.

We investigate order 8 to 32 schemes in two and three spatial dimensions. The very high order schemes, particularly 24 and 32, are somewhat artificial, because these orders are not commonly employed in practice. However, in this experiment, we are not so interested in the scheme itself but rather in the benefits and limits of YATeTo for a code involving three- and four-dimensional tensors.

The general expectation is that performance increases with order, because the number of floating point operations per degree of freedom increases. The results shown in Figure 11 do not meet...
Table 4. LinA: Number of Floating Point Operations and Data Traffic (Estimated and Measured) per Element, Given for SKX / DP

| Dim. | Order (O) | HW-FLOP | Estimate (best) Traffic | Estimate (worst) Traffic | Measured Traffic |
|------|-----------|---------|-------------------------|-------------------------|-----------------|
| 8    |           |         | 721,920                 |                         | 131 kB          |
|      |           |         | 115 kB                  |                         | 5.5             |
|      |           |         | 6.3                     |                         |                 |
| 3D   | 16        |         | 20,475,904              |                         | 833 kB          |
|      |           |         | 721 kB                  |                         | 24.6            |
|      |           |         | 28.4                    |                         |                 |
|      |           |         | 19.5 MB                 |                         |                 |
|      |           |         | 1.7                     |                         |                 |
|      |           |         | 6.68 MB                 |                         |                 |
|      |           |         | 22.6                    |                         |                 |
|      | 24        |         | 150,819,840             |                         | 6.68 MB         |
|      |           |         | 2.21 MB                 |                         | 22.6            |
|      |           |         | 68.2                    |                         |                 |
|      |           |         | 88.9 MB                 |                         |                 |
|      |           |         | 1.7                     |                         |                 |
|      |           |         | 2.3                     |                         |                 |
|      | 32        |         | 626,835,456             |                         | 309 MB          |
|      |           |         | 4.98 MB                 |                         | 2.0             |
|      |           |         | 125.8                   |                         |                 |
|      |           |         | 267 MB                  |                         |                 |

The working set of the ADER kernel consists of four tensors, each requiring $4O^3$ floating point numbers in 3D. In the best case, these tensors stay in cache. In the worst case, the working set is constantly evicted. We may assume the best case for $O = 8$ and the worst case for $O = 32$ due to cache sizes. $O = 16$ and $O = 24$ lie somewhere between best case and worst case, as confirmed by measurements.

our initial expectations: While in two dimensions performance increases with order for the most part, and achieves over 50% peak for order 32 on SKX, the performance in three dimensions stagnates on KNL and decreases on SKX beginning from order 16.

The reason for the observed behaviour becomes clear when carefully counting floating point operations and memory transfers, as shown in Table 4. In our initial estimate (“best”), the arithmetic intensity indicates that the order 8 implementation is memory bound and that the implementations for orders 16, 24, and 32 are compute bound. Here, we assume that the element-independent tensors $\tilde{K}, \tilde{K}, F, \tilde{F}$ stay in cache and that we need to load and store the degrees of freedom and the boundary nodes only once.

The initial estimate is too optimistic w.r.t. to the ADER kernel (cf. Equations (25) and (26)). Inside ADER, we require storage for four tensors of size $O^d(1 + d)$, where $O = N + 1$ is the order of the scheme and $d$ is the number of spatial dimensions. We need one buffer for the current and the next derivative, one buffer contains the time integral, and another temporary buffer is used for intermediate products. For each derivative, we need to access all four buffers in a round-robin fashion. In 2D, a maximum storage size of 96 KiB is required for all four tensors, which easily fits into L2 caches. In 3D, we require 512 KiB, 1.7 MiB, and 4 MiB for orders 16, 24, and 32, and half the amount for single precision respectively. Clearly, the 4 MiB for $O = 32$ clearly exceed the 512 KiB per-core cache on KNL and the 2.1375 MiB per-core cache on SKX. Thus, the four tensors are constantly evicted from cache and need to be reloaded from memory in each derivative. We considered that effect in the worst-case estimate in Table 4. On KNL, we expect the worst case also for orders 16 and 24, because the four tensors require a cache size greater than or equal to the available cache size.

To support our argument and to clarify the data traffic for orders 16 and 24 on SKX, we measured the memory data volume using LIKWID [Treibig et al. 2010], version 5. We listed the derived memory traffic per element in Table 4. For order 8, the memory traffic closely matches the best-case estimate, and for order 32 the memory traffic closely matches the worst-case estimate. Computing the arithmetic intensity shows that order 32 is clearly memory bound albeit the large number of operations. For order 16, the best-case estimate closely matches the measured data. For order 24, the four tensors must at least partially stay in cache but the measured traffic exceeds the best-case estimate by a factor of 3.

Furthermore, we evaluated possible performance gains due to the choice of code generator and memory layout. For the evaluation of the flux function (i.e., multiplication with $A^*, B^*,$ and $C^*$), we have to contract the degrees of freedom with matrices smaller than $(d + 1) \times (d + 1)$. We performed micro-benchmarks on KNL that suggest that the LIBXSMM kernels are sub-optimal for such matrix
sizes, which we believe to be due to superfluous vaddpd instructions and due to the fixed \(8 \times N\) register blocking on KNL. As a consequence, we added PSpaMM to replace the sub-optimal kernels, which is implemented in YATeTo by specifying a list of code generators, where we may give higher preference to a code generator for specific matrix sizes. Using PSpaMM and LIBXSMM combined gives us a non-zero performance boost up to 202 GFLOPS in DP or up to 344 GFLOPS in SP (for KNL O16 in 2D). Using a CSC memory layout for the matrices \(A^*, B^*,\) and \(C^*\) gives a further performance boost up to 74 GFLOPS in DP or up to 194 GFLOPS in SP (for SKX O32 in 2D). In total, we obtain a performance increase of up to 31% for DP and 30% for SP, and a maximum non-zero peak performance of 1,769 GFLOPS for DP and 3,969 GFLOPS for SP.

### 6.6 Findings

In this section, we found that kernels generated with YATeTo are able to match the performance of Uphoff et al. [2017] and all major optimisations of SeisSol are found automatically. The extension to multiple simulations closely matches the performance of Breuer et al. [2017].

For LinA, we observe that the very high-order schemes in three spatial dimensions violate the initial design assumption that tensors are small enough. Here, the limiting factor is the per-core cache size, because the arithmetic intensity decreases when tensors are constantly evicted from cache. When tensors are small enough, though, high performance is achieved with automatically generated kernels. Moreover, YATeTo’s ability to mix and match different code generators increased performance by up to 31%.

### 7 SYNTHETIC BENCHMARKS

We sketch applications beyond ADER-DG schemes at the hand of synthetic benchmarks in this section. First, we apply YATeTo to a kernel from a quantum chemistry application implemented in the MADNESS framework [Stock et al. 2011]. Second, we evaluate the cost function from Section 4.6.3 based on the example of Baumgartner et al. [2005] given in Equation (1). Both benchmarks are included in YATeTo’s main repository in the examples folder.

#### 7.1 MADNESS

Small tensor contractions are found to be the core of MADNESS [Stock et al. 2011], which is a framework for solving problems in quantum chemistry and other applications [Harrison et al. 2016]. Kernels in MADNESS often take the following form [Stock et al. 2011]:

\[
R_{ijk} = S_{xyz}A^L_{xlm}A^R_{nym}Y^R_{mnl}Z^L_{znk}.
\]

(30)

Each L-R matrix pair is obtained by a low-rank decomposition of a \(p \times p\) matrix. In the following, we assume that the low-rank is equal among all L-R pairs and given by \(q\), such that indices \(i, j, k, x, y, z\) have size \(q\) and indices \(l, m, n\) have size \(q\). The authors of Stock et al. [2011] developed a code generator for such tensor contractions, which uses loop transformations and outputs loop-code and SSE3 intrinsics. In the following, we investigate the performance of the above kernel when implemented with YATeTo.

The above expression is a natural fit for YATeTo’s DSL. We note that the optimal order of evaluation, as stated by Stock et al. [2011], is found automatically. A few tweaks may be applied to the above formulation: First, the matrices \(A^L\) and \(A^R\) are stored already transposed to avoid a transpose at runtime. (Stock et al. [2011] store \(Z^L\) transposed, too.) Second, the rows of \(A^L\) and \(A^R\) may be padded with zeros, such that the number of rows are a multiple of the vector width (i.e., 8 in double precision and 16 in single precision).

Figure 12 shows our single-core performance results. We observe that in the extreme case of \(q = 1\), we obtain low performance of 4.2 GFLOPS–6.5 GFLOPS in DP and 6.2 GFLOPS–7.1 GFLOPS...
Fig. 12. Single-core performance of (30) for multiple tensor sizes \( p \) and \( q \) (see text). The plot shows non-zero flops, i.e., artificial flops due to zero-padding are not counted. The frequency is fixed to 2.3 GHz.

In SP. A similar performance was achieved by Stock et al. [2011] for \( q = 1 \), even though they used a CPU from the year 2008. A potential issue is that the generated GEMMs degenerate to either inner products, outer products, or scalar multiplications. Moving to larger values of \( q \), we see a steady increase in performance, up to 54.9 GFLOPS in DP and 102.3 GFLOPS in SP, which corresponds to 75% and 70% of the theoretical peak performance w.r.t. the fixed frequency of 2.3 GHz, respectively.

Comparing LIBXSMM to MKL, we see that using LIBXSMM is beneficial in most cases, especially for small tensor sizes. In many cases, zero-padding is either beneficial or delivers almost the same performance as its counterpart. But for cases \( p = 24 \) SP and \( p = 40 \) SP zero-padding mostly lowers performance. The reason for this behaviour stems likely from the multiplication with \( X^R \): Here, the introduction of zeros leads to a non-fused GEMM, which is fused in the non-padded variant.

In summary, we find that mapping operations to GEMM might be sub-optimal for very small tensor contractions. However, given large enough tensor contractions, we obtain high performance with low implementation effort, as the DSL naturally suits the problem description and the evaluation order with minimum cost is found automatically. Furthermore, the choice of memory layout (padded, non-padded) and GEMM back-end (LIBXSMM, MKL) might have a large impact on performance.

### 7.2 Optimal Index Permutations?

A direct implementation of Equation (1) is given by

\[
S_{abij} := A_{acik} B_{befl} C_{dfjk} D_{cdel}.
\]  

\[ (31) \]

YATeTo introduces two temporary tensors to map the above equation to binary tensor contractions. The index permutations of the temporary tensors are selected according to the cost function from Section 4.6.3. An interesting question is whether the simple cost model selects the best implementation.
Fig. 13. We tested all 576 index permutation variants of a benchmark problem on a single SKX core. The figure shows the empirical cumulative distribution function of all variants. The variant selected by our simple cost function is marked with a red dot. The frequency is fixed to 2.3 GHz.

To test the cost function, we manually map Equation (31) to binary tensor contractions as following:

\[ T_{c}^{(1)} := B_{bef} D_{cdel}, \]
\[ T_{c}^{(2)} := T_{c}^{(1)} C_{dfjk}, \]
\[ S_{abij} := T_{c}^{(2)} A_{acik}. \]

For both \( T^{(1)} \) and \( T^{(2)} \), one has to choose from \( 4! \) possible index permutations, so in total there are 576 implementations.

Figure 13 shows our single-core performance results for all implementations (indices have size 16). We observe that the direct implementation (31) achieves about 74% of the maximum observed performance. Moreover, the performance of the direct implementation beats 89% of alternative implementation variants.

We conclude that using the simple cost function results in decent performance in this example but it could be improved.

8 CONCLUSION AND OUTLOOK

We presented YATeTo, a tensor toolbox especially suited for small tensor operations. The toolbox includes standard techniques, such as strength reduction or the mapping of tensor operations to Loop-over-GEMM. To our best knowledge, we present novel optimisations for tensor operations such as equivalent sparsity patterns or optimal index permutations.

We included YATeTo in SeisSol, which allows to generate kernels for single forward simulations but also multiple forward simulations. From our point of view, the DSL simplifies the simultaneous implementation of single and multiple simulations considerably, but still allows to tune both implementation variants individually.

The performance of our new SeisSol version matches its previous performance for a single simulation. For multiple simulations, we obtain speedups of \( 1.1 \times \) to \( 6.5 \times \) and a non-zero peak efficiency of up to 27.5% on Skylake. In LinA, we saw that ability to mix and match GEMM kernels allows to increase performance by up to 31%, leading to a non-zero peak efficiency of up to 48%. Application of YATeTo to a literature example showed high performance over 50% peak for large enough tensors, but likely sub-optimal performance for very small tensors.

We integrated YATeTo in the tensor product–based code LinA with low implementation effort. High performance is achieved in LinA when the working set in the ADER kernel stays within the cache size limits. In future work, we like to apply LinA to more general ADER-DG schemes sup-
porting curved geometries and compare performance to existing implementations, e.g., Schoeder et al. [2018].

YATeTo is currently limited by the number of available code-generation back-ends. Support for transposition in small GEMMs is limited and one has to rely on general purpose compilers for the optimisation of tensor operations that cannot be expressed as tensor contractions (e.g., Loop-over-GER). The simple cost model used in the optimal index permutation algorithm gives decent but non-optimal results.

In future work, one might integrate YATeTo with existing compilers for UFL to simplify the setup of tensor expressions resulting from the finite element method. An interesting direction is also to evaluate vectorisation over elements (also sometimes called patches) on curved hexahedral meshes. Furthermore, we think that a useful extension to YATeTo are tools that assist the user in the development process, such as automatic roof-line models.

APPENDIX

A SEISSOL: FULL PERFORMANCE RESULTS
Table 5. Full Results of the SeisSol Benchmark

| Ord. | Sim. | SMT | MDof/s | NZ_{max} | HW_{max} | HW_{mad} |
|------|------|-----|--------|----------|----------|----------|
| 02   | R    | Yes | 770    | 134      | 1229     | 8.4      |
| 01   | Yes  | 739  | 129    | 1172     | 36.8     |
| 02   | Yes  | 3637 | 525    | 611      | 8.5      |
| 02   | Yes  | 4034 | 697    | 808      | 12.1     |
| 03   | Yes  | 1502 | 324    | 1709     | 5.4      |
| 03   | Yes  | 1427 | 907    | 1615     | 22.6     |
| 03   | Yes  | 4882 | 1038   | 1224     | 12.3     |
| 03   | No   | 3857 | 820    | 962      | 10.4     |
| 04   | Yes  | 1902 | 536    | 2122     | 56.1     |
| 04   | Yes  | 1897 | 534    | 2108     | 83.2     |
| 04   | No   | 4061 | 1143   | 1403     | 33.5     |
| 04   | No   | 2961 | 833    | 1017     | 22.2     |
| 05   | Yes  | 1930 | 723    | 2549     | 55.8     |
| 05   | Yes  | 1949 | 730    | 2568     | 55.6     |
| 05   | Yes  | 3030 | 1145   | 1469     | 33.8     |
| 05   | No   | 2163 | 817    | 1043     | 89.7     |
| 06   | Yes  | 1494 | 752    | 2902     | 5.6      |
| 06   | Yes  | 1489 | 749    | 2889     | 1.4      |
| 06   | No   | 2221 | 1135   | 1534     | 66.7     |
| 06   | No   | 1603 | 818    | 1095     | 13.8     |
| 07   | Yes  | 1204 | 806    | 2888     | 10.4     |
| 07   | Yes  | 1184 | 792    | 2839     | 11.2     |
| 07   | Yes  | 1191 | 814    | 1142     | 10.0     |
| 07   | Yes  | 1285 | 877    | 1220     | 17.6     |

(a) KNL / SP

| Ord. | Sim. | SMT | MDof/s | NZ_{max} | HW_{max} | HW_{mad} |
|------|------|-----|--------|----------|----------|----------|
| 02   | R    | Yes | 510    | 89       | 815      | 3.8      |
| 01   | Yes  | 530  | 92     | 840      | 0.3      |
| 02   | Yes  | 2939 | 508    | 584      | 0.9      |
| 02   | Yes  | 3300 | 570    | 653      | 3.9      |
| 03   | Yes  | 1494 | 269    | 1421     | 3.5      |
| 03   | Yes  | 1344 | 290    | 1521     | 10.5     |
| 03   | Yes  | 3377 | 718    | 877      | 3.2      |
| 03   | Yes  | 3619 | 769    | 935      | 1.6      |
| 04   | Yes  | 1562 | 440    | 1741     | 6.0      |
| 04   | Yes  | 1542 | 454    | 1714     | 26.8     |
| 04   | Yes  | 3565 | 1003   | 1306     | 2.2      |
| 04   | No   | 3687 | 1037   | 1543     | 0.5      |
| 05   | Yes  | 1901 | 712    | 2511     | 21.0     |
| 05   | Yes  | 1877 | 703    | 2474     | 12.4     |
| 05   | Yes  | 3655 | 1382   | 1918     | 1.5      |
| 05   | Yes  | 3712 | 1402   | 1937     | 20.2     |
| 06   | Yes  | 1965 | 988    | 3816     | 21.7     |
| 06   | Yes  | 1962 | 997    | 3845     | 49.8     |
| 06   | Yes  | 1600 | 840    | 2729     | 44.5     |
| 06   | Yes  | 3544 | 1809   | 2660     | 58.4     |
| 07   | Yes  | 1827 | 1222   | 4383     | 36.0     |
| 07   | Yes  | 1802 | 1206   | 4321     | 42.8     |
| 07   | Yes  | 3275 | 2237   | 3493     | 68.0     |
| 07   | No   | 3206 | 2187   | 3389     | 8.5      |

(b) KNL / DP

| Ord. | Sim. | SMT | MDof/s | NZ_{max} | HW_{max} | HW_{mad} |
|------|------|-----|--------|----------|----------|----------|
| 02   | R    | Yes | 414    | 72       | 330      | 0.7      |
| 02   | Yes  | 427  | 74     | 339      | 0.2      |
| 02   | Yes  | 1402 | 243    | 274      | 0.2      |
| 02   | Yes  | 1609 | 278    | 318      | 0.9      |
| 03   | Yes  | 696  | 150    | 459      | 2.1      |
| 03   | Yes  | 696  | 150    | 457      | 1.2      |
| 03   | Yes  | 1659 | 353    | 415      | 0.6      |
| 03   | Yes  | 1788 | 380    | 462      | 0.6      |
| 04   | Yes  | 1043 | 294    | 994      | 10.3     |
| 04   | Yes  | 1043 | 294    | 992      | 3.9      |
| 04   | Yes  | 1779 | 499    | 619      | 0.8      |
| 04   | Yes  | 1827 | 514    | 669      | 0.2      |
| 05   | Yes  | 1212 | 454    | 1363     | 5.9      |
| 05   | Yes  | 1199 | 449    | 1347     | 17.3     |
| 05   | No   | 1807 | 684    | 887      | 2.9      |
| 05   | No   | 1835 | 694    | 963      | 0.8      |
| 06   | Yes  | 1259 | 633    | 1970     | 28.6     |
| 06   | Yes  | 1244 | 626    | 1946     | 42.5     |
| 06   | Yes  | 1784 | 914    | 1246     | 3.1      |
| 06   | No   | 1766 | 902    | 1352     | 4.0      |
| 07   | Yes  | 1048 | 702    | 2293     | 101.3    |
| 07   | Yes  | 1324 | 688    | 2241     | 35.7     |
| 07   | No   | 1707 | 1142   | 1495     | 27.0     |
| 07   | No   | 1614 | 1103   | 1722     | 23.9     |

(c) SKX / SP

NZ:=non-zero GFLOPS and HW:=hardware GFLOPS. HW_{mad} shows the standard deviation estimated with the consistent median absolute deviation. Million degrees of freedom per second (MDof/s) are stated for a complete timestep.
ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their detailed comments and their valuable feedback.

REFERENCES

K. Åhlander. 2002. Einstein summation for multidimensional arrays. Comput. Math. Applic. 44, 8 (2002), 1007–1017. DOI: https://doi.org/10.1016/S0898-1221(02)00210-9

Martin S. Alnaes, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified form language: A domain-specific language for weak formulations of partial differential equations. ACM Trans. Math. Softw. 40, 2 (Mar. 2014). DOI: https://doi.org/10.1145/2566630

Harold L. Atkins and Chi-Wang Shu. 1998. Quadrature-free implementation of discontinuous Galerkin method for hyperbolic equations. AIAA J. 36:5 (1998), 775–782.

G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, Xiaoyang Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, Chi chung Lam, Qingda Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. 2005. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proc. IEEE 93, 2 (Feb. 2005), 276–292. DOI: https://doi.org/10.1109/JPROC.2004.840311

Nathan W. Brei. 2018. Generating Small Sparse Matrix Multiplication Kernels for Knights Landing. Master’s thesis. Technical University of Munich, Garching, Germany.

Alexander Breuer, Alexander Heinecke, Michael Bader, and Christian Pelties. 2014a. Accelerating SeisSol by generating vectorized code for sparse matrix operators. In Parallel Computing: Accelerating Computational Science and Engineering (CSE). IOS Press, 347–356. DOI: https://doi.org/10.3233/978-1-61499-381-0-347

Alexander Breuer, Alexander Heinecke, and Yifeng Cui. 2017. EDGE: Extreme scale fused seismic simulations with the discontinuous Galerkin method. In High Performance Computing, ISC 2017. Springer International Publishing, Cham, 41–60.

Alexander Breuer, Alexander Heinecke, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, and Christian Pelties. 2014b. Sustained petascale performance of seismic simulations with SeisSol on SuperMUC. In Proceedings of the 29th International Conference on Supercomputing (ISC’14). Springer, 1–18.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34, 3 (2008), 12:1–12:25. DOI: https://doi.org/10.1145/1356052.1356053

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. Retrieved from http://eigen.tuxfamily.org.

Edoardo Di Napoli, Diego Fabregat-Traver, Gregorio Quintana-Ortí, and Paolo Bientinesi. 2014. Towards an efficient use of the BLAS library for multilinear tensor contractions. Appl. Math. Comput. 235 (2014), 454–468. DOI: https://doi.org/10.1016/j.amc.2014.02.051

Michael Dumbser and Martin Käser. 2006. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes—I. The three-dimensional isotropic case. Geophys. J. Int. 167 (2006), 319–336.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34, 3 (2008), 12:1–12:25. DOI: https://doi.org/10.1145/1356052.1356053

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. Retrieved from http://eigen.tuxfamily.org.

R. Harrison, G. Beylin, F. Bischoff, J. Calvin, G. Fann, J. Fosso-Tande, D. Galindo, J. Hammond, R. Hartman-Baker, J. Hill, J. Jia, J. Kottmann, M. Yvonne Ou, J. Pei, L. Ratcliff, M. Reuter, A. Richie-Halford, N. Romero, H. Sekino, W. Shelton, B. Sundahl, W. Thornton, E. Valeev, Á. Vázquez-Mayagoitia, N. Vence, T. Yanai, and Y. Yokoi. 2016. MADNESS: A multiresolution, adaptive numerical environment for scientific simulation. SIAM J. Sci. Comput. 38, 5 (2016), S123–S142. DOI: https://doi.org/10.1137/15M1026171

Alexander Heinecke, Alexander Breuer, Michael Bader, and Pradeep Dubey. 2016a. High order seismic simulations on the Intel Xeon Phi processor (Knights Landing). In Proceedings of the 31st International Conference on High Performance Computing. Springer, 343–362. DOI: https://doi.org/10.1007/978-3-319-41321-1_18

Alexander Heinecke, Alexander Breuer, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, Christian Pelties, Arndt Bode, William Barth, Xiang-Ke Liao, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, and Pradeep Dubey. 2014.

ACM Transactions on Mathematical Software, Vol. 46, No. 4, Article 34. Publication date: October 2020.
Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications 34:39

Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 3–14.

Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016b. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’16)*. IEEE Press, Piscataway, NJ, 841–84:11. Retrieved from [http://dl.acm.org/citation.cfm?id=3014904.3015017](http://dl.acm.org/citation.cfm?id=3014904.3015017).

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

Miklós Homolya, Robert C. Kirby, and David A. Ham. 2017. Exposing and exploiting structure: Optimal code generation for high-order finite element methods. Retrieved from: arxiv.cs.MS/1711.02473.

Miklós Homolya, Lawrence Mitchell, Fabio Luporini, and David A. Ham. 2018. TSFC: A structure-preserving form compiler. *SIAM J. Sci. Comput.*. 40, 3 (2018), C401–C428. DOI: [https://doi.org/10.1137/17M1130642](https://doi.org/10.1137/17M1130642)

Maxwell Hutchinson, Alexander Heinecke, Hans Pabst, Greg Henry, Matteo Parsani, and David Keyes. 2016. Efficiency of high order spectral element methods on petascale architectures. In *Proceedings of the 31st International Conference on High Performance Computing*. Springer, 449–466. DOI: [https://doi.org/10.1007/978-3-319-41321-1_23](https://doi.org/10.1007/978-3-319-41321-1_23)

Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. Expression templates revisited: A performance analysis of current methodologies. *SIAM J. Sci. Comput.*. 34, 2 (2012), C42–C69. DOI: [https://doi.org/10.1137/110830125](https://doi.org/10.1137/110830125)

Intel Corporation. 2020. *Intel Xeon Processor Scalable Family: Specification Update* (June 2020 ed.). Retrieved from [www.intel.de/content/www/de/deprocessors/xeon/scalable/xeon-scalable-spec-update.html](http://www.intel.de/content/www/de/deprocessors/xeon/scalable/xeon-scalable-spec-update.html).

Martin Käser, Michael Dumbser, Josep de la Puente, and Heiner Igel. 2007. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes—III. Viscoelastic attenuation. *Geophys. J. Int.*, 168 (2007), 224–242.

D. Kempf, R. Heß, S. Müthing, and P. Bastian. 2018. Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *arXiv e-prints* arXiv:1812.08075 (2018).

T. Kolda and B. Bader. 2009. Tensor decompositions and applications. *SIAM Rev.* 51, 3 (2009), 455–500. DOI: [https://doi.org/10.1137/0707111X](https://doi.org/10.1137/0707111X)

David A. Kopriva. 2009. *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers* (1st ed.). Springer.

Chi Chung Lam. 1999. Performance optimization of a class of loops implementing multi-dimensional integrals. Ph.D. Dissertation. Graduate School of the Ohio State University, UMI Company. Retrieved from: [http://rave.ohiolink.edu/etdc/view?acc_num=osu1488191667180786](http://rave.ohiolink.edu/etdc/view?acc_num=osu1488191667180786).

Chi-Chung Lam, P. Sadayappan, Cociorva Daniel, Mebarek Alouani, and John Wilkins. 1999. Performance optimization of a class of loops involving sums of products of sparse arrays. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*.

Chi-Chung Lam, P. Sadayappan, and Raphaël Wenger. 1997. Optimal reordering and mapping of a class of nested-loops for parallel execution. In *Languages and Compilers for Parallel Computing*. Springer Berlin, 315–329.

Randall J. LeVeque. 2002. *Finite Volume Methods for Hyperbolic Problems*. Vol. 31. Cambridge University Press.

J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’15)*. 1–12. DOI: [https://doi.org/10.1109/SC.2015.2807591](https://doi.org/10.1109/SC.2015.2807591).

Anders Logg, Kent-Andre Mardal, and Garth Wells. 2012. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Springer.

Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2015. Cross-loop optimization of arithmetic intensity for finite element local assembly. *ACM Trans. Archit. Code Optimiz.* 11, 4 (Jan. 2015). DOI: [https://doi.org/10.1145/2687415](https://doi.org/10.1145/2687415)

D. Matthews. 2018. High-performance tensor contraction without transposition. *SIAM J. Sci. Comput.*. 40, 1 (2018), C1–C24. DOI: [https://doi.org/10.1137/16M108968X](https://doi.org/10.1137/16M108968X)

T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris. 2015. Generating efficient tensor contractions for GPUs. In *Proceedings of the 44th International Conference on Parallel Processing*. 969–978. DOI: [https://doi.org/10.1109/ICPP.2015.106](https://doi.org/10.1109/ICPP.2015.106)

Elmar Peise and Paolo Bientinesi. 2012. Performance modeling for dense linear algebra. In *Proceedings of the SC Companion: High Performance Computing, Networking Storage and Analysis (SC’12)*. IEEE Computer Society, Washington, DC, 406–416. DOI: [https://doi.org/10.1109/SC.Companion.2012.60](https://doi.org/10.1109/SC.Companion.2012.60)

Elmar Peise, Diego Fabregat-Traver, and Paolo Bientinesi. 2015. On the performance prediction of BLAS-based tensor contractions. In *Proceedings of the Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems Workshop (Lecture Notes in Computer Science, Vol. 8966)*. Springer, 193–212. DOI: [https://doi.org/10.1007/978-3-319-17248-4_10](https://doi.org/10.1007/978-3-319-17248-4_10)
