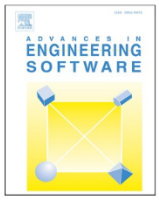
Advances in Engineering Software 159 (2021) 103033

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft



Fast evaluation of finite element weak forms using python tensor contraction packages

Robert Cimrman

New Technologies Research Centre, University of West Bohemia, Univerzitní 8, Plzeň 306 14, Czech Republic

ARTICLE INFO

Keywords:
Finite element method
Tensor contractions
Weak form transpiler
Numerical performance study

ABSTRACT

In finite element calculations, the integral forms are usually evaluated using nested loops over elements, and over quadrature points. Many such forms (e.g. linear or multi-linear) can be expressed in a compact way, without the explicit loops, using a single tensor contraction expression by employing the Einstein summation convention. To automate this process and leverage existing high performance codes, we first introduce a notation allowing trivial differentiation of multi-linear finite element forms. Based on that we propose and describe a new transpiler from Einstein summation based expressions, augmented to allow defining multi-linear finite element weak forms, to regular tensor contraction expressions. The resulting expressions are compatible with a number of Python scientific computing packages, that implement, optimize and in some cases parallelize the general tensor contractions. We assess the performance of those packages, as well as the influence of operand memory layouts and tensor contraction paths optimizations on the elapsed time and memory requirements of the finite element form evaluations. We also compare the efficiency of the transpiled weak form implementations to the C-based functions available in the finite element package SfePy.

1. Introduction

The Einstein summation convention, abbreviated as *einsum* in the following text according to the eponymous function of NumPy [15], is a powerful and concise way of writing linear algebra or tensor contraction expressions and as such it has received attention from computer science related communities. Large einsum-related research is going on in tensor networks applicable in machine learning or quantum physics calculations, see e.g. [12,17,19,25,27].

In the finite element method (FEM), a mathematical model defined using partial differential equations (PDEs) is transformed to a weak (integral) formulation and then discretized, see e.g. [31]. In the discretization, the unknown continuous functions from infinite-dimensional function spaces are approximated by finite-dimensional vectors of coefficients — the degrees of freedom (DOFs) — that are used as linear combination coefficients of basis functions with small support over the individual finite elements that cover the solution domain. This leads to the necessity of evaluating the weak form integrals over each element using a numerical quadrature, usually using nested loops over elements, and over quadrature points. Fast evaluation of the weak form integrals is crucial for efficient calculations, especially when using higher order polynomials/curves for the basis and hence the approximated variables. Many such integral forms (e.g. all linear or multi-linear) can be written using a single tensor contraction, or einsum, expression.

In recent years machine learning techniques and related contractions of large tensor networks started to be involved in the solution of PDEs using various approaches such as collocation or energy minimization approaches, see e.g. [26]. In contrast with that we wish to use the tensor contraction expressions directly in the classical FEM to evaluate the weak forms of PDEs. However, unlike the huge tensor networks in machine learning or quantum calculations [12], the einsum expressions in the FE context consist usually of only a few (e.g. three or four) tensors and the contractions have low numerical intensity. This complicates use of modern parallel architectures such as GPUs and is subject to an ongoing research [29]. Nevertheless, einsums are successfully used, for example, in the pure Python finite element package scikit-fem [13], where vectorized FE assembling (cf. [24]) is employed for fast integral form evaluations.

In this paper we assess several Python scientific computing packages, that implement, optimize and in some cases parallelize the general tensor contractions using the einsum semantics. The tested packages are:

E-mail address: cimrman3@ntc.zcu.cz.

https://doi.org/10.1016/j.advengsoft.2021.103033
Received 18 May 2021; Received in revised form 18 June 2021; Accepted 5 July 2021
Available online 20 July 2021

- NumPy [15], with the basic implementation and some contraction optimizations using a version of opt_einsum;
- opt_einsum [28], with state-of-the-art contraction optimization strategies;
- Dask [11], which can use opt_einsum as a backend, can parallelize the calculation and allows out-of-core calculations with very large data.
- JAX [7], with JIT compilation[1] and possible parallel execution or automatic GPU transfer and einsum optimization for GPUS in spirit of [29].

The used tensor contraction packages require rectangular (non-ragged) arrays, which limits a single einsum call to a set of finite elements of the same shape and approximation order. Applying the einsum semantics in the context of p- or hp-adaptivity [2] or XFEM [6,21] would require an additional sorting/grouping of the same element kinds together. Such situation is not considered in this paper.

The calculation speed and memory requirements of the various strategies are evaluated using several integral forms common in solving multi-physical problems and compared with the C-based implementation available in the finite element package SfePy [10]. SfePy is used by our team for solving relatively small ($\sim 10^6$ DOFs), but still physically relevant, multiphysical and multiscale problems on a single computer and so far targets only CPU computations.

We present mostly the single-thread performance of the above packages because that is directly influenced by the contraction path optimality in terms of the number of floating point operations (flops) and use of BLAS [5], nevertheless results with enables multi-threading are presented in Appendix A.

While we compare the performance with a particular FE solver (SfePy), the obtained results do not depend on that and are relevant to general audience for the following reasons. The local element tensors and resulting vectors/matrices are tied to the mathematical problem and thus are essentially the same in every FE code. We analyze effects of the memory layout of the tensors on the time and memory requirements of the tensor contractions. The tensor contractions in einsum function syntax are generated by a simple "weak form to einsum" transpiler that we introduce in Section 5. This description/implementation of multi-linear forms does not abstract out the details (basis functions evaluated in quadrature points etc. are numpy arrays) and can be reused. The calculations are pure-Python from user's perspective, and while our approach is smaller in scope than e.g. libCEED, it might offer a simple yet efficient way of implementing the local matrix action operator [3].

The paper is structured as follows: first, in Section 2 several notions that are used throughout the text are explained. In Section 3 we discuss factors that influence the speed of tensor contractions from the perspective of local FE evaluations. Then in Section 4 we recall some basic FE concepts necessary for the subsequent text and establish the relation to einsum expressions, and in Section 5 we introduce the transpiler from generalized einsum-like expressions, suitable for describing multi-linear finite element weak forms, to regular einsum/tensor contraction expressions. Finally, we present the results of many numerical simulations in Section 6 allowing us to assess the performance of the several einsum implementations mentioned above and conclude the paper in Section 7.

2. Explanation of basic concepts

The presented topic combines aspects of computer science, numerical approximation of PDEs and software engineering. This section purports to define the common ground necessary for readers of different backgrounds to understand the following parts.

A *transpiler* (or source-to-source compiler) translates input in a programming/domain specific language to another language that works at approximately the same level of abstraction, unlike a traditional compiler that translates from a higher level programming language to a lower level programming language. In Section 5 we use this term to denote our translator from generalized einsum-like notation to regular einsum function notation.

By *einsum expressions* we mean both the mathematical way of expressing implicit summation by repeated indices (the Einstein summation convention), such as

$$A_{ij} = B_{ik}C_{kj} \equiv \sum_{k=1}^{n} B_{ik}C_{kj} \ ,$$

where A, B and C are second order tensors (matrices), B has n columns and C has n rows, and, more specifically, the syntax and arguments used to call einsum functions of Python packages such as NumPy [15]. The above matrix multiplication in this syntax is

`A = einsum('ik,kj->ij', B, C).`

This operation is called *tensor contraction* and the index k is the contraction index or axis. Many common linear algebra operations can be expressed using the einsum syntax, for example (A is a matrix, u, v vectors with compatible dimensions):

trace of A	···	`einsum('ii', A)`
diagonal of A	···	`einsum('ii->i', A)`
transposition of A	···	`einsum('ij->ji', A)`
outer product of u, v	···	`einsum('i,j->ij', u, v)`
bilinear form A applied to u, v	···	`einsum('i,ij,j->', u, A, v)`

All multi-linear (linear, bilinear, etc.) forms that are defined using such multiplicative expressions can be expressed in this syntax. More examples specific to FE evaluations are given in Section 5.3.

3. What influences tensor contraction speed

Evaluating tensor expressions brings various trade-offs between the numerical intensity/number of flops, the memory consumption and access patterns, intermediate/temporary array sizes etc. In this Section we consider several factors that influence the evaluation speed of the weak form einsum expressions and address some of the trade-offs. The effects of memory layouts are presented in Section 3.1. Tensor contraction paths and related optimizations are introduced in Section 3.2.

3.1. Memory layout of operands

The einsum operands are initially, irrespective of the package used for performing the calculations, always stored in NumPy's n-dimensional array objects (`ndarray` class). A NumPy array is a continuous one-dimensional segment of memory that can be accessed using an indexing scheme that maps n integer indices into the location of a fixed-size item (e.g. a 64 bit floating point number) in the block. The ranges in which the indices can vary is specified by the shape of the array $(d_0, \cdots, d_{n-1})$ [15].

Among many schemes of arranging items of an n-dimensional array in a one-dimensional block, NumPy uses a flexible *strided indexing scheme* where an offset of an item $(i_0, \cdots, i_{n-1})$ from the beginning of the memory block is given by

$$i_{\text{offset}} = \sum_{k=1}^{n} s_k i_k \ ,$$

where the integers s_k specify the array strides. The most common strides are given by the *row-major order* (as in C), where the leftmost index varies the fastest, i.e. $s_k = \prod_{j=k+1}^{n-1} d_j$, or the *column major order* (as in Fortran or Matlab), where the rightmost index varies the fastest, i.e. $s_k =$

[1] JIT compilation = Just in time compilation.

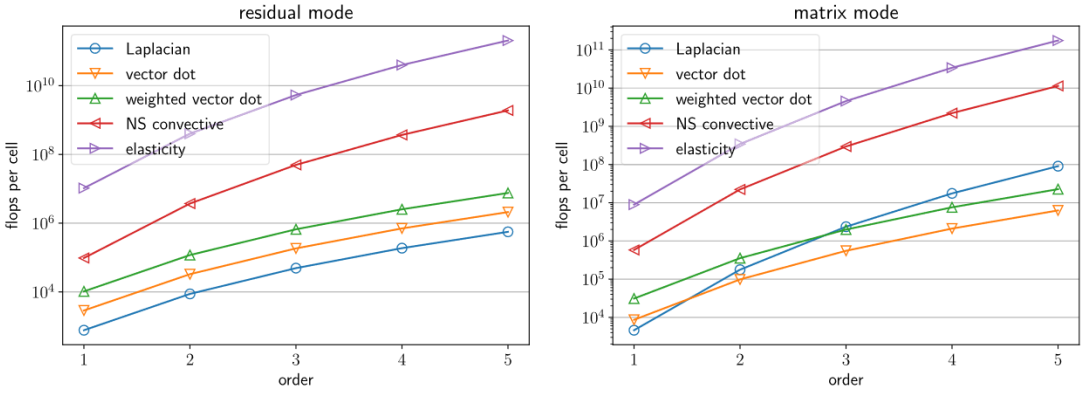


Fig. 1. Dependence of the naive flops counts per FE cell on the FE approximation order for several weak forms considered in Section 6. Left: residual mode, i.e. evaluation of the weak form action on a vector. Right: matrix mode, i.e. evaluation of the local element matrices.

$\prod_{j=0}^{k-1} d_j$.

The term memory layout can refer both to the layout in terms of the strides (row-major order, column major order,...) and to the order of the array dimensions. In this work we assume all operands to be in a contiguous block of memory in the row-major order, and by layout we denote the various permutations of array dimensions.[2]

The operand layout significantly influences the speed of the tensor contractions, as will be shown in Section 6.4. For illustration, let us consider the multiplication of two matrices: for all i, k, $C_{ik} = \sum_j A_{ij} B_{jk}$. Having the innermost loop index j vary along a contiguous block of memory is usually much faster than when, for fixed i,k, the jth and $(j+1)$th items of A and B are not next to each other in memory for all j, as data access patterns with high locality help avoiding memory cache misses.

3.2. Tensor contraction path optimization

For einsum expressions with more than two operands, a number of possibilities exist of how, or in what order, to perform the tensor contractions. The naive flops cost, corresponding to a for loop for each index in the expression, is proportional to the product of all operand shapes, and quickly grows prohibitive. Even though in the context of this work the multi-linear weak form expressions have less than 10 operands (unlike the machine learning networks), the naive calculation time and memory requirements are not feasible for all but the simplest forms, see Fig. 1. It is much faster to contract a subset of operands at a time, which massively reduces the flops and intermediate array sizes and also allows BLAS use, see [28]. An ordered list of operands to contract at a time is called a contraction path.

The considered packages support various methods of contraction path optimizations. The resulting path is a sequence of index tuples (usually pairs), denoting the positions of operands to contract in a list of operands, see below. While the contraction path optimization algorithms themselves introduce an intrinsic overhead, the resulting contraction paths strongly influence the elapsed time and memory consumption, especially as the problem size increases and the optimization overhead becomes amortized.

For illustration let us compare the naive and optimized contraction paths of a chained dot product $A_{ij} B_{jk} C_{kl}$, i.e. the einsum expression `einsum('ij,jk,kl->il', A, B, C)`, where A is 2×2, B is 2×5, C is 5×2. In this case, opt_einsum [28] contraction path optimization results in the following:

```
Complete contraction: ij,jk,kl->il
Naive scaling: 4
Optimized scaling: 3
Naive FLOP count: 1.200e+2
Optimized FLOP count: 5.600e+1
Theoretical speedup: 2.143e+0
Largest intermediate: 4.000e+0 elements
-------------------------------------------------------------------------
scaling    BLAS        current              remaining
-------------------------------------------------------------------------
   3       GEMM        kl,jk->lj              ij,lj->il
   3       GEMM        lj,ij->il                il->il
```

That is, instead of the naive four nested loops, it is faster to first contract the B and C: `D = einsum('kl,jk->jl', B, C)`, three nested loops, and then to contract the result with A: `einsum('jl,ij->il', D, A)`, again three nested loops. The flops count of the naive algorithm is 120, while the optimized flops count is reported to be 56, providing a theoretical speed-up of about two. Furthermore, the two-step contraction allows using the fast function GEMM of a BLAS implementation. More examples of tensor contraction paths from the FE form evaluation context are shown in Section 5.3.

4. FE Evaluation and einsum

In this Section we recall some very basic FEM concepts that will be needed in subsequent text, details can be found e.g. in [31]. In particular, the concepts are reviewed using vector functions (scalar functions correspond to a single component) and two example weak forms: the vector dot product/mass matrix weak form and the convective term weak form from Navier-Stokes equations. We propose a unified notation that allows straightforward translation to einsum expressions as well as a trivial differentiation of those expressions.

4.1. Notation

Let us denote $\underline{u}, \underline{v}$ two vector functions from suitable function spaces (e.g. $H^1(\Omega)$ in the case of elasticity, $H^{\text{div}}(\Omega)$ in the case of fluid flow problems). For simplicity, we assume both functions to be from the same

[2] Both terms are related: an array of shape (m,n) in the row-major order has exactly the same arrangement of items in memory as an array of shape (n,m) in the column-major order.

space of functions $V_0(\Omega)$: a space of functions that are zero on the boundary $\partial\Omega$. In the FEM and related methods, the continuous functions are approximated by functions defined using a finite set of degrees of freedom (DOFs) and a finite element basis. A component u_i, $i = 1, \cdots, D$ can be expressed as the linear combination

$$u_i\left(\underline{x}\right) \approx u_i^h\left(\underline{x}\right) = \mathsf{u}_i^k \phi_k\left(\underline{x}\right) , \tag{1}$$

where u_i^k, $k = 1, \cdots, N$ are the discrete DOFs and $\phi_k\left(\underline{x}\right)$ are the basis functions — for example piece-wise polynomials with a small support in the case of FEM. The discrete functions $\underline{u}^h$ $\underline{v}^h$ are substituted in the weak form of a PDE in question and a discrete problem is obtained, in terms of weak form integrals. The integrals need to be evaluated over the solution domain Ω, resp. its approximation Ω^h, that is covered by non-overlapping cells — the elements. The total integral is then the sum of the element contributions.

The actual elements T_c, that can have various shapes, sizes and orientations in space, are often mapped to a reference (unit) element T, where the integration using a numerical quadrature takes place. Let us denote $\underline{x}$ the coordinates in space and $\underline{\xi}$ the coordinates in the reference element, then the mapping is defined as $F^c : \underline{\xi} \to \underline{x};\ \underline{x} \in T_c;\ x_i\left(\underline{\xi}\right) = x_i^k \phi_k\left(\underline{\xi}\right)$, where $\underline{x}^k$, $k = 1, \cdots, N$ are the coordinates of T_c vertices, $\phi_k\left(\underline{\xi}\right)$ are the basis functions restricted to the reference element[3] and the Jacobian matrix of F^c is $J_{ij}^c\left(\underline{\xi}\right) = \left\{\frac{\partial x_i}{\partial \xi_j}\right\}$. Applying F^c to $\underline{\xi} \in T_c$, we have also

$$u_i\left(\underline{\xi}\right) \approx u_i^h\left(\underline{\xi}\right) = \mathsf{u}_i^k \phi_k\left(\underline{\xi}\right) . \tag{2}$$

This allows evaluating weak form integrals on the reference element.

Thus the basis functions have well defined restrictions $\phi_k\left(\underline{\xi}\right)$ to the elements — the forms can be evaluated independently in the individual elements, in a (parallel) loop. An example of such a loop is in Algorithm 1. This particular loop evaluates a weak form integral in each cell using a numerical quadrature for a given DOF vector u. For linear forms, it is equivalent to the matrix-vector action of the form, i.e. $\mathsf{M}\mathsf{u}$. The same loop structure can be seen also in Algorithm 2, which evaluates the weak form derivative w.r.t. u, i.e. the matrix M for linear forms. Both functions return arrays of values for the given set of cells, the actual assembling to a global vector/sparse matrix is not shown.

Require: F ▷ A weak form evaluation function.
Require: $cells, qps$ ▷ Lists of FE mesh cells and reference cell quadrature points.
Require: v, u ▷ FE variables
Require: d ▷ material parameters, reference mappings (Jacobian J) and other data

function EVAL_RESIDUAL($cells$, qps, F, v, u, d)
  $n_c, n_{dr} \leftarrow get\_shape(cells, v, u, d)$ ▷ get result array shape
  $r \leftarrow zeros((n_c, n_{dr}))$ ▷ allocate result array
  for i_c in $cells$ **do** ▷ loop over cells
    for $\underline{\xi}$ in qps **do** ▷ loop over cell quadrature points
      $r[i_c, :] \leftarrow r[i_c, :] + d.J(\xi) \cdot F(i_c, \underline{\xi}, v, u, d)$ ▷ evaluate and integrate F
    end for
  end for
  return r
end function

Algorithm 1. Basic finite element loop algorithm for evaluating local element contributions to a global residual vector, i.e. the action of the form on a DOF vector u. The quadrature weights are incorporated in the element reference mapping Jacobian $J\left(\underline{\xi}\right)$.

function EVAL_MATRIX($cells$, qps, F, v, u, d)
  $n_c, n_{dr}, n_{dc} \leftarrow get\_shape(cells, v, u, d)$ ▷ get result array shape
  $M \leftarrow zeros((n_c, n_{dr}, n_{dc}))$ ▷ allocate result array
  for i_c in $cells$ **do** ▷ loop over cells
    for $\underline{\xi}$ in qps **do** ▷ loop over cell quadrature points
      $M[i_c, :, :] \leftarrow M[i_c, :, :] + d.J(\xi) \cdot \partial F(i_c, \underline{\xi}, v, u, d)/\partial u$ ▷ evaluate and integrate $\frac{\partial F}{\partial u}$
    end for
  end for
  return M
end function

Algorithm 2. Basic finite element loop algorithm for evaluating local element contributions to a global matrix, i.e. the derivative of the form w.r.t. a DOF vector u. The quadrature weights are incorporated in the element reference mapping Jacobian $J\left(\underline{\xi}\right)$.

[3] For the sake of notation simplicity, we assume isoparametric elements here, i.e. the same basis functions for the variables and the reference element mapping. We also use the same basis for each vector component.

In the following we will be interested in both modes of evaluation. The global matrices are required e.g. when using a direct linear system solver, while the matrix action/residual form can be used in connection with iterative solvers that do not need the global sparse matrix explicitly, only its application to a vector.

In compiled languages, such as Fortran or C++, the integrals over a group of elements can be calculated in a loop. In Python (and other interpreted languages), the loops are very slow — that is why NumPy and other packages allowing fast vectorized calculations were introduced. Considering this, our aim is to evaluate a FE expression by a single call vectorized over many elements, unlike the aforementioned listings. Below we expand on the already introduced notation to allow formulating the evaluation loops using einsum functions in a straightforward and efficient way. The notation facilitates using scalar $H^1(\Omega)$ basis functions with vector variables, which saves both time and memory during calculations.

Traditionally the DOFs of all components will be stored in a vector — a 1D array: (2) in matrix notation, for $D = 3$, has the following form:

$$\begin{aligned} \underline{u}^h\left(\underline{\xi}\right) &= \begin{bmatrix} \phi_1 & \cdots & \phi_N & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \phi_1 & \cdots & \phi_N & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \phi_1 & \cdots & \phi_N \end{bmatrix} \begin{bmatrix} \mathsf{u}_1^1 \\ \vdots \\ \mathsf{u}_1^N \\ \mathsf{u}_2^1 \\ \vdots \\ \mathsf{u}_2^N \\ \mathsf{u}_3^1 \\ \vdots \\ \mathsf{u}_3^N \end{bmatrix} \\ &\equiv \begin{bmatrix} \boldsymbol{\phi} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\phi} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{\phi} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \end{bmatrix} \equiv \boldsymbol{\Phi}\left(\underline{\xi}\right)\mathbf{u} \equiv \sum_{i=1}^{DN} (\boldsymbol{\Phi})_i (\mathbf{u})_i \,, \end{aligned} \tag{3}$$

where $\boldsymbol{\phi}$ (argument $\underline{\xi}$ omitted) is the vector of scalar basis functions evaluated in $\underline{\xi}$ and $\boldsymbol{\Phi}\left(\underline{\xi}\right)$ is the $D \times DN$ matrix, whose columns can be interpreted as vector basis functions $(\boldsymbol{\Phi})_i$ corresponding to items of the DOF vector $\mathbf{u}$ of size DN. Because all components of $\underline{u}$ in (3) are approximated using the same scalar basis functions, it can be rearranged as follows:

$$\underline{u}^h\left(\underline{\xi}\right) = \begin{bmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \mathbf{u}_3^T \end{bmatrix} \boldsymbol{\phi}^T\left(\underline{\xi}\right) \equiv \mathbf{U}^T \boldsymbol{\phi}^T\left(\underline{\xi}\right) \,, \tag{4}$$

where $\mathbf{U}$ is the $N \times D$ matrix with columns formed by component DOF vectors $\mathbf{u}_i$. In our weak form to einsum transpiler in Section 5, we use the compact expression (4) as well as a tensor (three-dimensional array of shape $D \times D \times N$) form of the basis functions matrix $\boldsymbol{\Phi}\left(\underline{\xi}\right)$ of (3)

$$\begin{aligned} \boldsymbol{\Phi}^{(3)}\left(\underline{\xi}\right) &= \left[\begin{bmatrix} \phi_1 & \cdots & \phi_N \\ 0 & \cdots & 0 \\ 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} 0 & \cdots & 0 \\ \phi_1 & \cdots & \phi_N \\ 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} 0 & \cdots & 0 \\ 0 & \cdots & 0 \\ \phi_1 & \cdots & \phi_N \end{bmatrix} \right] \\ &\equiv \left\{ \delta_{ij} \phi_k\left(\underline{\xi}\right) \right\} \,, \end{aligned} \tag{5}$$

where $i,j = 1, \cdots, D$ and $k = 1, \cdots, N$. This allows distinguishing the indices of u_i^h components from the DOF components u_j^k, by rewriting (2) as

$$u_i^h\left(\underline{\xi}\right) = \mathsf{u}_j^k \delta_{ij} \phi_k\left(\underline{\xi}\right) \,. \tag{6}$$

The expression (6) will be used when differentiating a form w.r.t. the DOFs and enables using scalar $H^1(\Omega)$ basis functions with vector variables as mentioned earlier in this Section. In the relation (3) the result of differentiation w.r.t. the DOFs is obvious: it is the matrix $\boldsymbol{\Phi}$. But to avoid forming this rather sparse matrix explicitly, (6) can be used, accompanied with a subsequent reshape operation $(j,k) \to J$, $j = 1, \cdots, D$, $k = 1, \cdots, N$, $J = 1, \cdots, DN$.

4.2. Example weak forms

One of the most common forms is the one corresponding to the vector dot product/mass matrix, used e.g. for l_2 projections, in elastodynamics and elsewhere. Evaluating this form in an element T_C, in the context of vector variables, means calculating

$$m_c\left(\underline{v}, \underline{u}\right) = \int_{T_c} v_i\left(\underline{x}\right) u_i\left(\underline{x}\right) = \int_T v_i\left(\underline{\xi}\right) u_i\left(\underline{\xi}\right) \left| J^c\left(\underline{\xi}\right) \right| \,. \tag{7}$$

Substituting the FE approximation (2), applied to both v_i and u_i, into (7) yields

$$\begin{aligned} m_c^h\left(\underline{v}^h, \underline{u}^h\right) &= \int_T \mathsf{v}_i^k \phi_k\left(\underline{\xi}\right) \mathsf{u}_i^m \phi_m\left(\underline{\xi}\right) \left| J^c\left(\underline{\xi}\right) \right| \\ &= \mathsf{v}_i^k \left(\int_T \phi_k\left(\underline{\xi}\right) \phi_m\left(\underline{\xi}\right) \left| J^c\left(\underline{\xi}\right) \right| \right) \mathsf{u}_i^m \,. \end{aligned} \tag{8}$$

The expression (8) is suitable for calculating the dot product of two known vector functions $\underline{v}^h$, $\underline{u}^h$, i.e. a scalar value, but to evaluate either a weak form residual or matrix, either the traditional matrix relation (3) can be used

Table 1
The generalized einsum-like notation.

symbol	meaning	example
`0`	scalar	p
`i`	ith vector component	u_i
`i.j`	gradient: derivative of ith vector component w. r.t. jth coordinate component	$\frac{\partial u_i}{\partial x_j}$
`i:j`	symmetric gradient	$\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$
`s(i:j)->I`	vector storage of symmetric second order tensor, I is the vector component	Cauchy strain tensor $e_{ij}\left(\underline{u}\right)$

Table 2
Examples of multi-linear weak form definitions.

description	definition	weak form expression
vector dot product	`('i,i', v, u)`	$\int_T v_i u_i$
weighted vector dot product	`('ij,i,j', M, v, u)`	$\int_T v_i M_{ij} u_j$
weak Laplacian	`('0.i,0.i', v, u)`	$\int_T \frac{\partial v}{\partial x_i} \frac{\partial u}{\partial x_i}$
Navier-Stokes convection	`('i,i.j,j', v, u, u)`	$\int_T v_i \frac{\partial u_i}{\partial x_j} u_j$
Stokes coupling	`('i.i,0', v, p)`	$\int_T \frac{\partial v_i}{\partial x_i} p$
divergence operator	`('i.i', v)`	$\int_T \frac{\partial v_i}{\partial x_i}$
transposed Stokes coupling	`('i.i,0', u, q)`	$\int_T q \frac{\partial u_i}{\partial x_i}$
linear elasticity	`('IK,s(i:j)->I,s(k:l)->K', D, v, u)`	$\int_T D_{ijkl}\, e_{ij}\left(\underline{v}\right) e_{kl}\left(\underline{u}\right)$
Cauchy stress	`('IK,s(k:l)->K', D, u)`	$D_{ijkl}\, e_{kl}\left(\underline{u}\right)$

Table 3
Memory layout specification letters.

letter	axis
c	cells
q	quadrature points
v	variable component
g	gradient component
d	local DOF (basis, node)
0	all material axes

$$m_c^h\left(\underset{\sim}{v}^h,\underset{\sim}{u}^h\right) = \mathbf{v}^T\left(\int_T \mathbf{\Phi}^T\left(\underset{\sim}{\xi}\right)\mathbf{\Phi}\left(\underset{\sim}{\xi}\right)\left|J^c\left(\underset{\sim}{\xi}\right)\right|\right)\mathbf{u} = (\mathbf{v})_I \mathbf{M}_{IJ}(\mathbf{u})_J \,, \tag{9}$$

which requires working with $\mathbf{\Phi}^T\left(\underset{\sim}{\xi}\right)$, or the form (6) can be employed:

$$m_c^h\left(\underset{\sim}{v}^h,\underset{\sim}{u}^h\right) = \mathsf{v}_a^k\left(\int_T \delta_{ia}\phi_k\left(\underset{\sim}{\xi}\right)\delta_{ib}\phi_m\left(\underset{\sim}{\xi}\right)\left|J^c\left(\underset{\sim}{\xi}\right)\right|\right)\mathsf{u}_b^m = \mathsf{v}_a^k \mathsf{M}_{akbm}\mathsf{u}_b^m \,. \tag{10}$$

followed by a simple reshape operation $ak{\to}I$, $bm{\to}J$, which does not need to copy/move data, if the reshaping of M_{akbm} corresponds to its memory storage scheme (e.g. row-major contiguous).

The form (8) has a very simple structure, so it can be readily seen that the mass matrix corresponding to vector variables is formed by D blocks of the scalar mass matrix ($\int_T \phi_k\left(\underset{\sim}{\xi}\right)\phi_m\left(\underset{\sim}{\xi}\right)\left|J^c\left(\underset{\sim}{\xi}\right)\right|$), but the tensor form (10) enables a straightforward (tangent) matrix evaluation of more complex forms too, for example the convective term below.

The convective term weak form of the Navier-Stokes equations is non-linear, if the advection velocity coincides with the flow velocity, but its structure is that of a multi-linear term. In an element T_C it is defined as

$$c_c\left(\underset{\sim}{v},\underset{\sim}{u},\underset{\sim}{u}\right) = \int_{T_c} v_i\left(\underset{\sim}{x}\right)\frac{\partial u_i}{\partial x_j}\left(\underset{\sim}{x}\right)u_j\left(\underset{\sim}{x}\right) = \int_T v_i\left(\underset{\sim}{\xi}\right)\frac{\partial u_i}{\partial x_j}\left(\underset{\sim}{\xi}\right)u_j\left(\underset{\sim}{\xi}\right)\left|J^c\left(\underset{\sim}{\xi}\right)\right| \,, \tag{11}$$

where $\frac{\partial u_i}{\partial x_j}\left(\underset{\sim}{\xi}\right)$ can by expressed in terms of $\underset{\sim}{\xi}$ using $\frac{\partial u_i}{\partial x_j}\left(\underset{\sim}{\xi}\right) = \frac{\partial u_i}{\partial \xi_k}\left(\underset{\sim}{\xi}\right)J^{-1}_{kj}\left(\underset{\sim}{\xi}\right)$. Substituting the tensor FE approximation (6) into (11) yields (the argument $\underset{\sim}{\xi}$ is omitted for brevity)

$$c_c^h\left(\underset{\sim}{v}^h,\underset{\sim}{u}^h,\underset{\sim}{u}^h\right) = \mathsf{v}_a^k\int_T \delta_{ia}\phi_k\mathsf{u}_b^l\delta_{ib}\frac{\partial\phi_l}{\partial x_j}\mathsf{u}_c^m\delta_{jc}\phi_m|J^c| \,, \tag{12}$$

which allows trivial differentiation w.r.t. $\mathbf{u}$, we just omit u_b^l and then u_c^m from the expression and sum the two terms together:

$$\begin{aligned}\frac{\partial c_c^h}{\partial \mathbf{u}} &\overset{\text{reshape}}{=} \mathsf{v}_a^k\int_T\left(\delta_{ia}\phi_k\delta_{ib}\frac{\partial\phi_l}{\partial x_j}\mathsf{u}_c^m\delta_{jc}\phi_m + \delta_{ia}\phi_k\mathsf{u}_b^l\delta_{ib}\frac{\partial\phi_l}{\partial x_j}\delta_{jc}\phi_m\right)|J^c| \\ &= \mathsf{v}_a^k\int_T\left(\delta_{ia}\phi_k\delta_{ib}\frac{\partial\phi_l}{\partial x_j}\mathsf{u}_j^m\phi_m + \delta_{ia}\phi_k\mathsf{u}_i^l\frac{\partial\phi_l}{\partial x_j}\delta_{jc}\phi_m\right)|J^c| \,,\end{aligned} \tag{13}$$

where in the second identity we replaced back $\mathsf{u}_c^m\delta_{jc}$ by u_j^m and $\mathsf{u}_b^l\delta_{ib}$ by u_i^l, i.e. used the compact expression (4).

The weak form to einsum transpiler described in Section 5 allows automatic transformation of the description corresponding to the left-hand side of (11) to an einsum-ready descriptions of the discretized form (12) and (13).

5. Weak form to einsum transpiler

In this Section we propose a generalized einsum-like notation suitable for describing weak form integrals and introduce a transpiler that can translate expressions in this notation to regular einsum expressions and operands.

The generalized einsum-like syntax that we propose is both close to the mathematical notation and explicit in what tensor axes are contracted, i.e. the weak form definitions are self-documenting, compare e.g. with the very versatile [14] Unified Form Language [1] from the FEniCS project, where named symbolic operators such as `inner` or `grad` are used.

5.1. Overview

Let us denote by p a scalar unknown variable and by q the corresponding test function. Similarly, $\underset{\sim}{u} \equiv \{u_i\}$, $\underset{\sim}{v} \equiv \{v_i\}$ term a vector unknown and test variable pair.

In the proposed implementation, we generalize the einsum notation with symbols for derivatives and other quantities summarized in Table 1. Using those symbols it is possible to define all the (multi-)linear terms available in SfePy in a unified and easy manner. Several example definitions are shown in Table 2 — five of those weak form expressions are the subject of the performance analysis in the next Section.

The generalized einsum-like expressions are transformed to regular einsum expressions using a simple transpiler with the following features:

- several einsum evaluation backends, see Section 5.2;
- arbitrary memory layout of the operands;
- easy automatic differentiation, thanks to (multi-)linearity of the considered weak forms;
- various evaluation modes (see below).

The following evaluation modes, a feature available in SfePy, are supported:

- 'weak' modes for expressions involving a test variable:
  - 'residual' mode corresponding to the operator/matrix application to a DOF vector without the global sparse matrix assembly;
  - 'matrix' mode returning the local element matrices — derivatives of the expression w.r.t. a given variable;
- 'eval' mode returning the integral value in case all variables passed to the evaluation function have associated DOFs, i.e. no test variables are in the argument list.

All weak forms shown in Table 2 support all the above evaluation modes automatically, with the exception of the Cauchy stress expression: this form has no test variables and cannot be called in a 'weak' mode, only evaluated.

The transpiler allows performing simple transformations of the expressions and operands, namely:

- Memory layout application: the required memory layout of einsum operands can be specified using a letter symbol for each operand axis, listed in Table 3.
- Expression slicing: to reduce memory usage, an expression can be evaluated in a loop one cell at a time.
- Operand conversion: operands can be retyped to dask arrays with chunks defined in terms of cell slices.
- Transformations of expression indices and special operand storage: see the linear elasticity weak form example in Section 5.3.

The weak form evaluation function implementing the above features is outlined in Algorithm 3. In the actual implementation, results of various algorithm steps (e.g. the expression transpilation) are cached to speed up subsequent evaluations.

Require: *ge* ▷ the generalized einsum-like expression of a weak form
Require: *data* ▷ FE variables, material parameters, reference mappings etc.
Require: *layout* ▷ operands memory layout
Require: *backend* ▷ einsum expression evaluation backend
function EVALUATE(ge, data, backend)
    $ee \leftarrow transpile(ge)$ ▷ transpile *ge* to einsum expression (backend independent)
    $eval\_einsum \leftarrow get\_function(backend)$ ▷ backend-dependent evaluation function
    $ops \leftarrow get\_operands(data)$ ▷ operands from FE data
    $shape \leftarrow get\_output\_shape(ee, ops)$
    $ee, ops \leftarrow apply\_layout(ee, ops)$ ▷ apply layout, does not change output shape
    $ee, ops \leftarrow transform(ee, ops, backend)$ ▷ do backend-dependent transformations
    $paths \leftarrow get\_paths(ee, ops, backend)$ ▷ get optimized contraction paths
    return $eval\_einsum(shape, ee, ops, paths)$
end function

Algorithm 3. Outline of the weak form evaluation function based on the weak form to einsum transpiler.

5.2. Backends

A backend terms an expression evaluation function tailored for the specific library. The following backends are currently supported, the abbreviations in parentheses are used in Section 6:

- 'numpy' (np): the expression evaluation function uses `numpy.einsum()`.
- 'numpy_loop' (npl): `numpy.einsum()` is applied to a sliced expression sequentially one cell at a time.
- 'opt_einsum' (oe): the expression evaluation function uses `opt_einsum.contract()`, more path optimization options are available than with `'numpy'`;
- 'opt_einsum_loop' (oel): like 'numpy_loop' but with `opt_einsum.contract()`.
- 'jax' (jx): the expression evaluation function uses `jax.numpy.einsum()` and is JIT-compiled using `jax.jit()`.
- 'jax_vmap' (jxv): the expression evaluation function uses `jax.numpy.einsum()`, is JIT-compiled using `jax.jit()` and vectorized using `jax.vmap()`, which leads to a similar behavior to the '*_loop' backends.
- 'dask_single' (das), 'dask_threads' (dat): the expression evaluation function uses `dask.array.einsum()`, explicitly without or with multiple threads.

Table 4
Symbols used in the expression examples.

variable.array	meaning
v.det	a reference mapping Jacobian J multiplied by quadrature weights
u.bf	basis functions ψ_k corresponding to the approximation of u
u.bfg	$\frac{\partial \psi_k}{\partial \xi_i}$ corresponding to the approximation of u
u.I	the identity matrix
u.dofs	the DOFs of u in the matrix form $\mathbf{U}$
u.Psg	a tensor of shape $(D, D, D(D+1)/2)$ expressing the vector storage of the symmetric gradient
m1_m2.arg	an array m2 of a material m1

Table 5
The numbers of cells n_c, FE approximation orders, numbers of quadrature points, DOF counts per component, number of evaluation repetitions and the array sizes of the scalar and vector weak form residuals $|r_s|$, $|r_v|$ and matrices $|M_s|$, $|M_v|$.

#cells	order	#QP	#DOFs/comp.	repeat	$\|r_s\|$ [MB]	$\|M_s\|$ [MB]	$\|r_v\|$ [MB]	$\|M_v\|$ [MB]
1024	1	8	4100	6	0.1	0.5	0.2	4.7
	2	27	18,441	6	0.2	6.0	0.7	53.7
	3	64	49,168	5	0.5	33.6	1.6	302.0
	4	125	102,425	5	1.0	128.0	3.1	1,152.0
	5	216	184,356	4	1.8	382.2	5.3	3,439.9
2048	1	8	8196	6	0.1	1.0	0.4	9.4
	2	27	36,873	6	0.4	11.9	1.3	107.5
	3	64	98,320	5	1.0	67.1	3.1	604.0
	4	125	204,825	4	2.0	256.0	6.1	2,304.0
4096	1	8	16,388	6	0.3	2.1	0.8	18.9
	2	27	73,737	6	0.9	23.9	2.7	215.0
	3	64	196,624	5	2.1	134.2	6.3	1,208.0
8192	1	8	32,772	6	0.5	4.2	1.6	37.7
	2	27	147,465	5	1.8	47.8	5.3	430.0
	3	64	393,232	4	4.2	268.4	12.6	2,415.9
16,384	1	8	65,540	6	1.0	8.4	3.1	75.5
	2	27	294,921	5	3.5	95.6	10.6	860.0
32,768	1	8	131,076	6	2.1	16.8	6.3	151.0
	2	27	589,833	5	7.1	191.1	21.2	1,719.9
65,536	1	8	262,148	6	4.2	33.6	12.6	302.0
	2	27	1,179,657	5	14.2	382.2	42.5	3,439.9
131,072	1	8	524,292	6	8.4	67.1	25.2	604.0
262,144	1	8	1,048,580	5	16.8	134.2	50.3	1,208.0
524,288	1	8	2,097,156	5	33.6	268.4	100.7	2,415.9
1,048,576	1	8	4,194,308	5	67.1	536.9	201.3	4,831.8

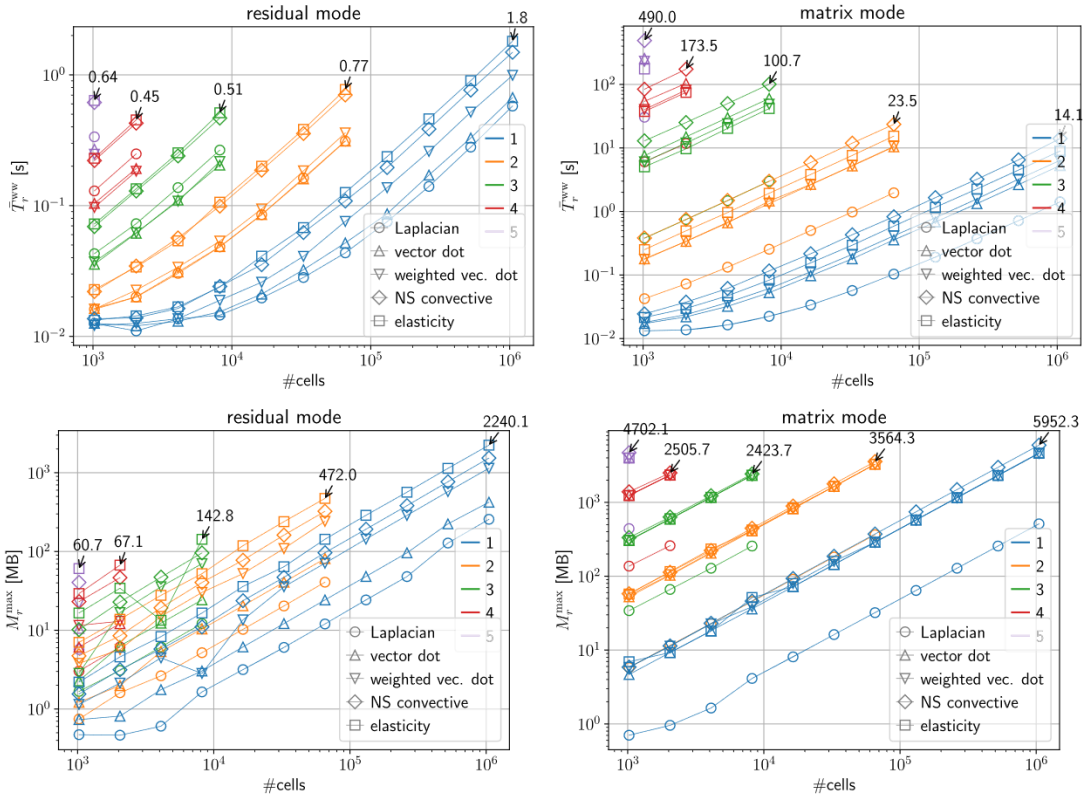


Fig. 2. Reference implementation performance in evaluating the five selected weak forms denoted by markers. Left: residual mode, right: matrix mode, top: the elapsed time mean without the worst case $\bar{T}_r^{\mathrm{ww}}$, bottom: the maximum memory consumption with respect to the repeat parameter $M_r^{\max}$. The highest values for each color-coded approximation order are annotated by arrows.

- 'opt_einsum_dask_single' (oedas), 'opt_einsum_dask_threads' (oedat): like 'opt_einsum', but the operands are converted to dask arrays with chunks defined in terms of cell slices and dask is used by opt_einsum to do the contraction, either without or with multiple threads. The 'auto' chunk size is used by default, i.e. suitable sizes are determined by dask.

5.3. Expression examples

Several examples below illustrate the functioning of the transpiler. The default layout, compatible with SfePy, is used, i.e. the row-major storage of arrays with the cqgvd0 array axes order. The following symbols defined in Table 4 will be used in the explanation.

5.3.1. Weak laplacian expression

The weak Laplacian integral defined by the expression ('0.i,0.i', v, u) gets translated in the matrix mode to einsum('cq,cqjd, cqje->cde', v.det, v.bfg, u.bfg), where det is the reference element mapping Jacobian and bfg is the matrix of basis functions gradients, as defined in Table 4. For a particular case of 1024 tri-quadratic hexahedral elements, the data sizes are:

```
cde (1024, 27, 27) =
v.det   cq    (1024, 27)
v.bfg   cqjd  (1024, 27, 3, 27)
u.bfg   cqje  (1024, 27, 3, 27)
path: [(0, 1), (0, 1)]
```

That is, the integral is over 1024 cells with 27 quadrature points each, j denotes the three components of the gradients and there are 27 DOFs per cell. The last line shows the contraction path resulting from the greedy optimization algorithm implemented in NumPy's einsum() function: a pair (i,j) denotes a contraction of operands i, j, then the operand i is removed from the list of operands and the operand j is replaced by the contraction result. This process is repeated until all the contractions are evaluated and a single operand, the result, remains.

Using either the 'numpy_loop' or the 'opt_einsum_loop' backend causes the expression to be compiled to einsum('q,qjd,qje->de', v.det[c], v.bfg[c], u.bfg[c]) that is evaluated for each cell c. In Section 6 we demonstrate that this slicing approach is an efficient method when applied to FE approximation orders greater or equal to three, regardless the weak form.

5.3.2. Vector dot product expression

The dot product of vector variables, discussed in Section 4.2 is defined by the expression ('i,i', v, u), which is transpiled to

einsum('cq,qd,ir,qe,is->crdse', v.det, v.bf, v.I, u.bf, u.I) in the matrix mode, with data sizes for 1024 tri-linear hexahedral elements

```
crdse (1024, 3, 8, 3, 8) =
v.det   cq    (1024, 8)
v.bf    qd    (8, 8)
v.I     ir    (3, 3)
u.bf    qe    (8, 8)
u.I     is    (3, 3)
path: [(2, 4), (0, 1), (0, 2), (0, 1)]
```

and to einsum('cq, qd,ir,qe,cie->crd', v.det, v.bf, v.I, u.bf, u.dofs) in the vector mode, with data sizes

```
crd (1024, 3, 8) =
v.det   cq    (1024, 8)
v.bf    qd    (8, 8)
v.I     ir    (3, 3)
u.bf    qe    (8, 8)
u.dofs  cie   (1024, 3, 8)
```

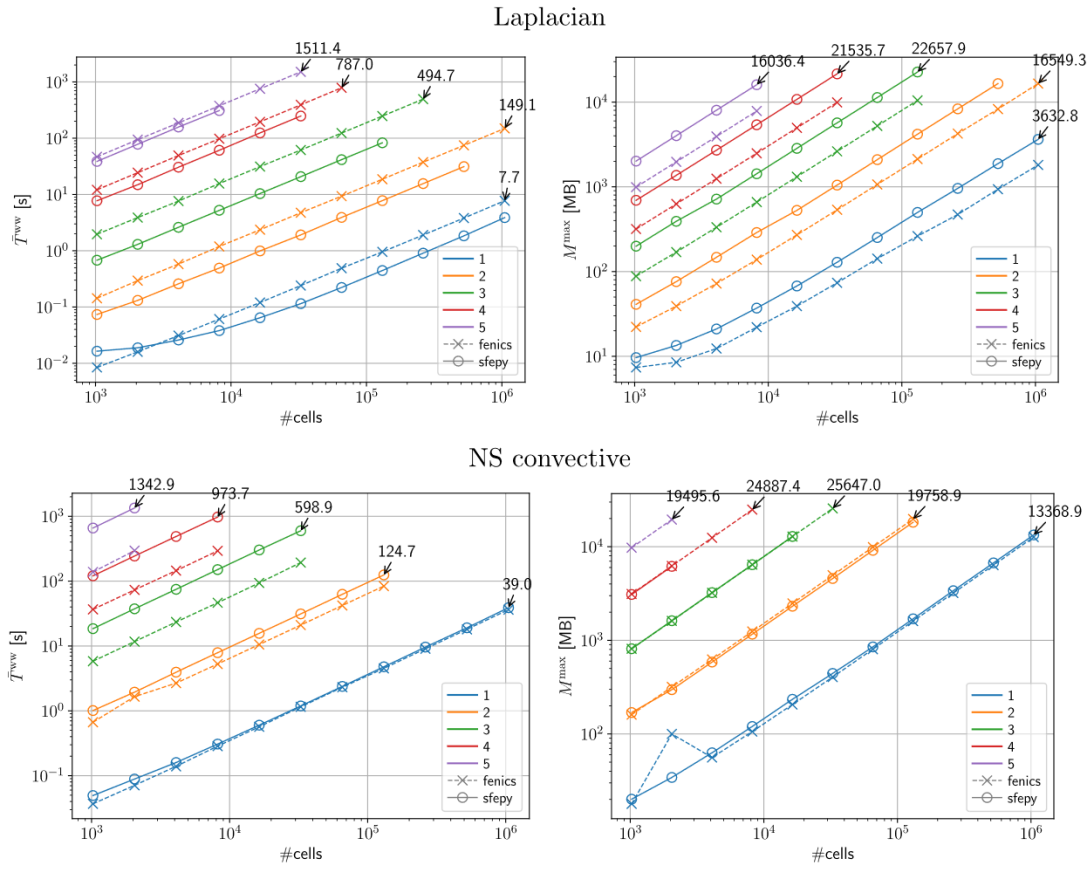


Fig. 3. Performance of matrix mode evaluations of the SfePy reference implementation (circles) in comparison with FEniCS (crosses) for various approximation orders. Left: the elapsed time mean without the worst case $\overline{T}^{ww}$, right: the memory consumption M^{max}. Top: the weak Laplacian, bottom: the convective term weak form. The highest values for each color-coded approximation order are annotated by arrows.

Table 6
The SfePy/FEniCS ratios of the elapsed time mean without the worst case $\overline{T}^{ww}$ and the memory consumption M^{max} for the weak Laplacian and convective term weak forms for various approximation orders. The values correspond to medians over various numbers of mesh cells, see Fig. 3.

	Laplacian					NS convective				
	1	2	3	4	5	1	2	3	4	5
$\text{med}(\overline{T}^{ww}_{\text{sfepy}} / \overline{T}^{ww}_{\text{fenics}})$	0.50	0.42	0.34	0.63	0.83	1.06	1.49	3.20	3.33	4.65
$\text{med}(M^{max}_{\text{sfepy}} / M^{max}_{\text{fenics}})$	1.75	1.96	2.16	2.17	2.04	1.06	0.93	1.00	0.99	-

`path: [(3, 4), (0, 3), (0, 2), (0, 1)]`

Note that the same expressions work also with scalar variables, but the transpilation result is different.

5.3.3. Linear elasticity expression

The last example covers the linear elasticity weak form and illustrates the transformations of expression indices and a special operand storage. The form is defined using the second order Cauchy strain tensor (a.k.a. symmetric gradient)

$$e_{ij}\left(\underline{u}\right) = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$$

and the fourth order tensor D_{ijkl} with elastic coefficients. Exploiting symmetry of those tensors allows a substantial memory saving: in 3D, e_{ij} is commonly stored as a vector of 6 components instead of a 3×3 matrix, and D_{ijkl} is stored as a 6×6 matrix, instead of the full tensor with 81 items. This storage and shape transformation is supported by our transpiler via the `s(i:j)->I` syntax, meaning a symmetric storage subset is taken from the tensor indices i, j and the result is put into a 1D array with the index I. Hence the linear elasticity weak form is defined simply using the expression `('IK,s(i:j)->I,s(k:l)->K', D, v, u)`, that is compiled to `einsum('cq,cqje,rjI,cqIK,cqlf,slK->cresf', v.det, v.bfg, v.Psg, m_D.arg, u.bfg, u.Psg)` in the matrix mode, with data sizes for 1024 tri-linear hexahedral elements

```
cresf (1024, 3, 8, 3, 8) =
v.det   cq    (1024, 8)
v.bfg   cqje  (1024, 8, 3, 8)
v.Psg   rjI   (3, 3, 6)
m_D.arg cqIK  (1024, 8, 6, 6)
u.bfg   cqlf  (1024, 8, 3, 8)
u.Psg   slK   (3, 3, 6)
```

Table 7
The fastest layouts per FE approximation order and backend in the case of the weak Laplacian einsum expression, $\overline{T}^{ww}/\overline{T}_r^{ww}$ terms the relative speed-up of the fastest layout w.r.t. the default layout.

order	lib	$\overline{T}^{ww}/\overline{T}_r^{ww}$	layout	$\overline{T}^{ww}/\overline{T}_d^{ww}$
1	das	1.65	cdgq	0.54
	jx	0.58	cgqd	0.79
	jxv	12.91	cdgq	0.54
	np	1.33	cdgq	0.50
	npl	57.76	cgqd	0.95
	oe	1.27	cdgq	0.48
	oedas	1.70	cdgq	0.54
	oel	60.52	gcdq	0.79
2	das	1.12	cdgq	0.43
	jx	0.87	cdgq	0.87
	jxv	1.83	cgqd	0.91
	np	0.95	cdgq	0.39
	npl	3.42	cgqd	0.98
	oe	1.06	cdgq	0.41
	oedas	1.94	cdgq	0.56
	oel	4.87	cqgd	0.99
3	das	0.88	cdgq	0.35
	jx	0.50	cgqd	0.78
	jxv	0.51	cgqd	0.81
	np	0.82	cdgq	0.33
	npl	0.75	gcdq	0.99
	oe	0.83	cdgq	0.33
	oedas	0.97	cdgq	0.36
	oel	0.87	cgdq	0.98

```
path: [(0, 1), (0, 1), (0, 1, 2, 3)]
```

The above contraction path corresponds again to the greedy optimization algorithm implemented in NumPy's `einsum()` function. Better performance can be obtained in this case e.g. by using the dynamic programming optimization algorithm of opt_einsum's `contract()` function, which returns the following path:

```
path: [(0, 4), (2, 3), (1, 3), (1, 2), (0, 1)]
```

Note also the special tensor `Psg` of shape $(D, D, D(D+1)/2)$ which extracts the basis function gradient components and stores them in a vector. This tensor has only D nonzeros (ones), however this sparsity is not exploited. Consequently, the current performance of the linear elastic form einsum expression is not optimal. Nevertheless, the ability to express this term using a single simple expression is very convenient.

6. Performance evaluation results

Five representative weak forms from Table 2 were chosen for studying elapsed time and memory consumption of the einsum evaluation backends from Section 5.2, namely the vector dot product, weighted vector dot product, weak Laplacian, Navier Stokes convection and linear elasticity terms. For comparison, their counterparts from SfePy were used, all implemented using a hand-crafted C code, called from Python via SfePy's lightweight Cython [4] wrapper. The reference implementation evaluates the forms cell-by-cell, and thus requires a very small memory additional to that occupied by the operands and the resulting array.

The einsum backend performance measurements were executed on a Linux workstation with the AMD Ryzen Threadripper 1920X 12-Core Processor, 32 GB RAM and Python version 3.8.5, NumPy 1.20.1, opt_einsum 3.3.0, JAX 0.2.9, Dask 2021.02.0 and SfePy 2021.1 (development version) installed.

We were interested mostly in the single-thread performance of the einsum backends because that was directly influenced by the contraction path optimality in terms of the number of floating point operations (flops) and some of the calculations were memory-demanding — the largest tasks used all the available RAM. To ensure single-threaded execution of all backends, the relevant environment variables were set appropriately and the CPU affinity parameter was set to a single value[4] Vice-versa, the environment variables and affinity were unset in the potentially multi-threaded runs in Appendix A.

A 1D bar made of a varying number of 3D hexahedral cells served as the finite element mesh. Note that the mesh topology is irrelevant for matrix mode linear weak form evaluations — it plays some role only for the nonlinear convective term form and for residual mode evaluations: the mesh cell connectivity is used to index the DOF vector. On this mesh, the finite element approximations of orders from 1 to 5 were defined. The used cell and DOF counts for the five forms are summarized in Table 5: the scalar weak Laplacian form employs variables with a single component per quadrature point, the other, vector, forms contain variables with three components per quadrature point, i.e. the largest vector problem involved 12,582,924 DOFs. For the sake of the performance measurements, the same numbers of quadrature points as given in the table were applied to integrating all the five weak forms, although in practice those numbers would lead to inexact integration e.g. in the convective term case.

Note that only the form evaluations (einsum function calls), leading to arrays of dense vectors/matrices, were measured, not the subsequent FE assembling into the global residual vector or sparse matrix forming the linear system, except the comparison results in Section 6.3. The residuals r_s, r_v in Table 5 are arrays of shapes (n_c, n_{dr}), where n_{dr} is the number of DOFs in one cell (the number of shape functions), the matrices M_s, M_v have shapes (n_c, n_{dr}, n_{dc}), i.e. the local element matrices are $n_{dr} \times n_{dc}$ and n_{dr}, n_{dc} depend on the approximation order and the type of the weak form, see also Algorithms 1, 2 in Section 4.1.

In relation to the reference implementation performance in Section 6.3, we first explore the influence of the operand memory layouts on the performance of the weak Laplacian and Navier-Stokes convection forms in Section 6.4 and then report both relative and absolute results for all the five forms in Section 6.5, considering only the default, SfePy-compatible, layout. All data presented below are available in the form of Pandas DataFrames [20] online [9].

6.1. Measuring memory consumption

A time-based memory usage including child processes was tracked using memory_profiler, version 0.58.0 [23]. The individual evaluation functions were tagged in the data, and the memory consumption of a function was defined as the maximum minus minimum in the tagged interval of that function. The sampling periods from 0.001 s to 0.1 s were used, depending on the problem size. Due to the time-based sampling nature of the memory tracking, values for very small problems may be skewed. In such cases the memory requirements were negligible with respect to the available memory. On the other hand, memory consumption values for large problems, which are of most interest, were measured accurately.

6.2. Choice of performance indicators

The first evaluation of a weak form using the Algorithm 3 is slower then subsequent evaluations because of the necessary first-time steps. Each weak form was evaluated using a single backend/optimization combination several times according to the repeat parameter for various mesh sizes and function approximation orders, see Table 5. To omit the additional setup time of the first evaluation, which is amortized e.g. in practical time-dependent problems, the mean elapsed times reported

[4] It is a known issue (https://github.com/google/jax/issues/1539) that JAX cannot run using a single thread, so the affinity workaround was used.

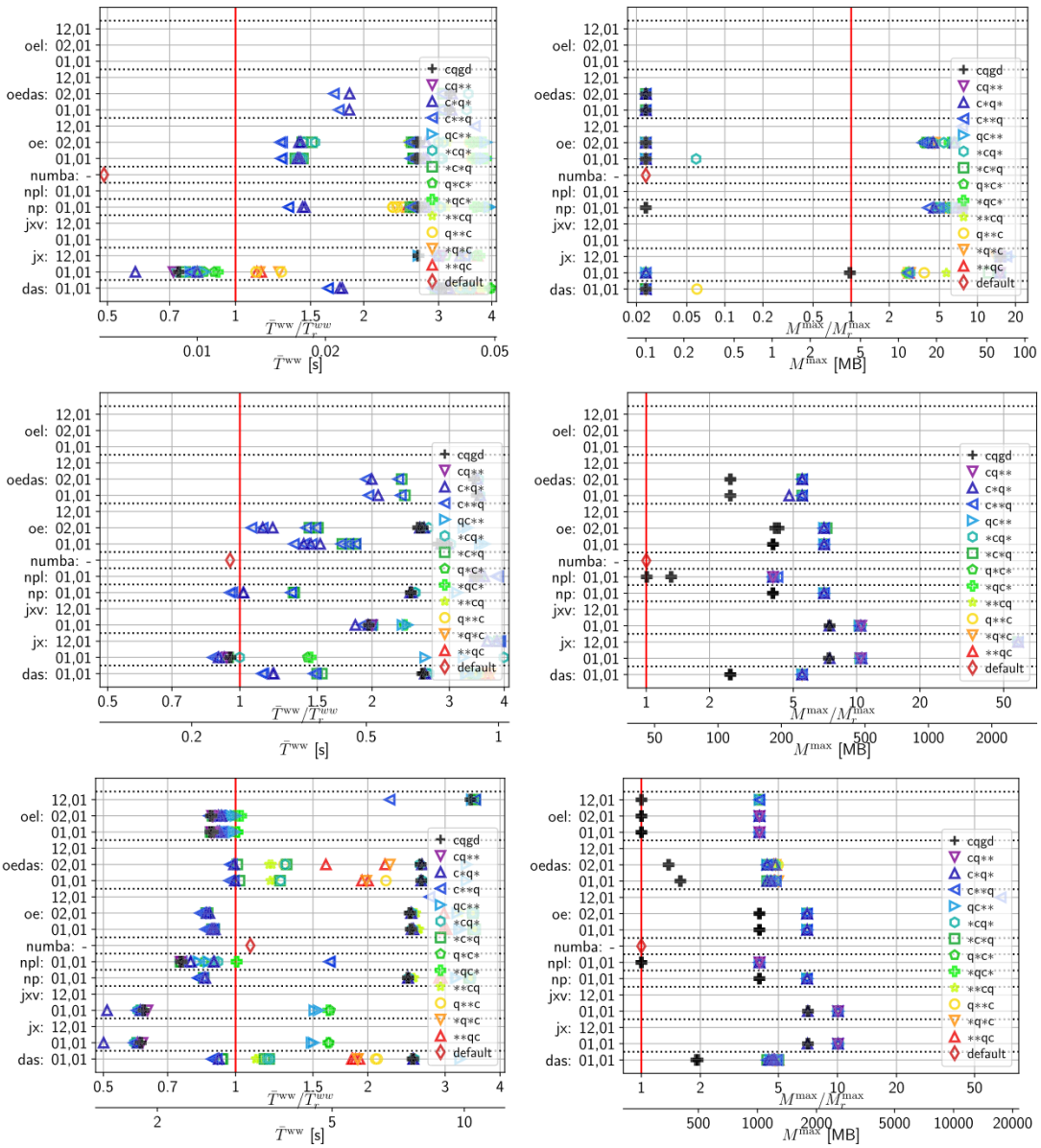


Fig. 4. Elapsed time and memory requirements of the weak Laplacian einsum expression depending on the operand memory layouts for 8192 cells, FE approximation orders 1 (top), 2 (middle), 3 (bottom). Left: (relative) elapsed time mean without the worst case ($\bar{T}^{ww}/\bar{T}_r^{ww}$, $\bar{T}^{ww}$). Right: (relative) maximum memory consumption (M^{max}/M_r^{max}, M^{max}). The reference values obtained by SfePy are marked by vertical red lines. The libraries and contraction paths are indicated in y axis labels. The colors/markers indicate positions of cell and quadrature axes in the layout string, the SfePy layout is marked by black crosses among einsum expression backends. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

below are the means without the worst case, denoted $\bar{T}^{ww}$, the worst case being the largest elapsed time among each set of repeated evaluations. This is important especially for the JAX-based backends because of the JIT-compilation during the first call. On the other hand, the memory consumption is given as the maximum over the repeated evaluations, denoted M^{max}, because it determines whether a problem can be calculated within available RAM or not. Although not included in the paper, the complete data [9] contain all the other statistics (means, minima, maxima, ranks etc.)

6.3. Reference implementation performance

The reference implementation performance in evaluating the five selected weak forms is illustrated in Fig. 2. The reference results shown, i.e. the elapsed time mean without the worst case $\bar{T}_r^{ww}$ [s] and the maximum memory consumption M_r^{max} [MB], are with respect to the repeat parameter in Table 5.

The results demonstrate a linear dependence of the elapsed times as well as the memory consumption on the number of mesh cells. For small meshes and the approximation order 1 it is apparent — especially in the residual mode, where the resulting arrays are much smaller than in the matrix mode — that there is a small fixed time overhead and that this overhead is amortized for more than 10^4 cells. Similarly, the memory sampling measurements are not very accurate for very fast evaluations/small data even with the sampling period of 0.001 s, however, for larger data (e.g. the matrix mode) the measurements are reliable enough for comparison purposes. The relative data in subsequent sections are w.r.t. the results shown in Fig. 2.

Comparison with FEniCS

The aim of this auxiliary study is not to provide a comprehensive

Table 8
The fastest layouts per FE approximation order and backend in the case of the weak convective term einsum expression, $\overline{T}^{\mathrm{ww}}/\overline{T}_d^{\mathrm{ww}}$ terms the relative speed-up of the fastest layout w.r.t. the default layout.

order	lib	$\overline{T}^{ww}/\overline{T}_r^{ww}$	layout	$\overline{T}^{ww}/\overline{T}_d^{ww}$
1	das	1.05	vdgcq	0.82
	jx	0.47	cqgdv	0.98
	jxv	3.71	cqgvd	1.00
	np	2.18	gvdcq	0.89
	npl	25.73	cdgqv	1.00
	oe	0.58	vdcqg	0.82
	oedas	1.00	dvcqg	0.88
	oel	32.74	cqgdv	0.99
2	das	0.82	cgdqv	0.95
	jx	0.18	cqgdv	0.96
	jxv	0.28	vcqgd	0.99
	np	1.25	gdvcq	0.98
	npl	0.92	cvdgq	0.98
	oe	0.19	cgvdq	0.85
	oedas	0.28	cgvdq	0.90
	oel	1.20	gcdqv	0.95

comparison, indeed the results discussed below were obtained using a single software/hardware configuration, but to provide a broader context of the our main results in the following sections. We compare the serial performance of the reference implementation in SfePy with a widely acknowledged and used package FEniCS [18], version 2019.1.0. The weak Laplacian and convective term weak forms were used for the comparison. The results presented in Fig. 3 show the elapsed time mean without the worst case $\overline{T}^{\mathrm{ww}}$ [s] with respect to five repetitions and the maximum memory consumption M^{max} [MB] (max. - min. consumption over all the repetitions together). Unlike results in Fig. 2, the measurements correspond to combined weak form data allocation, evaluation and assembling to a sparse matrix, i.e. the whole procedure of FEM application to a weak form. The memory sampling measurements were done with the sampling period of 0.001 seconds. The results are summarized in Table 6, where the ratios SfePy/FEniCS are given for all approximation orders, using medians over the various numbers of mesh cells. Missing data are due to the time-out of 2000 seconds and/or insufficient memory.

The evaluation and assembling of the matrix corresponding to the weak Laplacian is faster in SfePy, almost three times for the order 3: the $\overline{T}^{\mathrm{ww}}$ ratios in Table 6 are decreasing up to the order 3 and then again increasing. However, SfePy needed about two times more memory than FEniCS, which caused calculation failures for several higher-order/large number of cells cases (missing circles in Fig. 3 top left). On the other hand, in the case of the convective term the memory consumption of both packages was comparable. The calculation speed was also essentially the same for the order 1, but SfePy was getting progressively worse with the increasing order: up to four and half times for the order 5, where SfePy timed-out and so the corresponding memory consumption data points are missing. Analogous behavior was observed also for the vector dot form (not shown here for brevity). To conclude, SfePy assembling performance seemed on par with FEniCS for low order approximations, and worse for higher order approximations with about the same memory requirements. Faster SfePy performance when assembling the weak Laplacian was connected with higher memory consumption. It is worth noting that the weak form evaluation algorithm based on the developments of Section 5 offers a significant performance boost w.r.t. the SfePy reference implementation especially for higher order approximations, as will be demonstrated in the following text.

6.4. Influence of tensor memory layouts

This section explores the influence of the various possible memory layouts on the weak form evaluation performance, to establish the position of the default, SfePy-compatible, layout within other possible layouts. The default layout elapsed time mean without the worst case is denoted by $\overline{T}_d^{\mathrm{ww}}$.

The default layout specification is constructed as follows. The scalar basis function gradient tensor has 4 dimensions (cells, quadrature points, gradient components, DOFs) and the layout `cqgd` (L_1). The DOFs tensor has 3 dimensions (cells, variable components, cell DOFs) with the layout `cvd` (L_2). Material parameter tensors default layout is `cq0` (L_3), where `0` is a placeholder for any number of axes. Compressing the three specifications into a single string yields the default specification `cqgvd0`. Other possible memory layouts of the weak form einsum expression operands were generated by permuting the layout specification letters of L_1, L_2, L_3 and compressing them together, leading to 216 different layouts[5], that were applied to the operands of the matrix mode einsum expression of the weak Laplacian and convective terms.

6.4.1. Weak laplacian layouts

The results below were obtained by a parametric study involving 1461 calculations run using soops [8], all with the mesh of 8192 cells. The FE approximation order 3 runs had a time-out set to 50 seconds for 4 repetitions, the order 1 and 2 runs used 6 and 5 repetitions, respectively, and no time-out. The number of runs that timed-out is excluded from the calculations count.

The dependence of the weak Laplacian einsum expression evaluation performance on the operand memory layouts is first summarized in Table 7, where the fastest layouts for each FE approximation order and einsum backend are listed, together with $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$ and $\overline{T}^{\mathrm{ww}}/\overline{T}_d^{\mathrm{ww}}$. The layouts are specified using permutations of the reduced default layout `cqgd`, because the FE space is scalar (no `v`) and the weak form has no material parameters (no `0`).

Elapsed time and memory requirements of the individual backends are shown in Fig. 4 for the FE approximation orders 1 (top), 2 (middle) and 3 (bottom). Both $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$ with $\overline{T}^{\mathrm{ww}}$ and $M^{\mathrm{max}}/M_r^{\mathrm{max}}$ with M^{max} are provided. The libraries and contraction paths are indicated in y axis labels. The contraction paths are used instead of the optimization algorithms because the paths actually significantly influence the performance, and furthermore the optimization time was negligible. The paths are written in a condensed way, where e.g. 12.01 corresponds to the $[(1, 2), (0, 1)]$, see Section 5.3. Note that several optimization algorithms can lead to a single contraction path, so each (library, path) pair corresponds to the number of calculations equal to 216 layouts times the number of optimizations with the given path. However, only the values at most 4× slower than the reference SfePy implementation are shown in Fig. 4, with the corresponding memory consumption. To reduce the color/marker legend size, only positions of `cq` axes are indicated in the legend, `*` denotes any letter from `gd`. For comparison, a hand-written weak Laplacian element matrix evaluation loop implementation, JIT-compiled using Numba (version 0.52.0) [16], was included in this study.

6.4.2. Weak convective term layouts

The results below were obtained by a parametric study involving 3602 calculations run using soops [8], all with the mesh of 8192 cells. The FE approximation order 1 and 2 runs used 6 and 5 repetitions, respectively, and no time-out.

The dependence of the weak convective term expression evaluation performance on the operand memory layouts is first summarized in Table 8, where the fastest layouts for each FE approximation order and einsum backend are listed, together with $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$ and $\overline{T}^{\mathrm{ww}}/\overline{T}_d^{\mathrm{ww}}$. The

[5] This is less than 4!3!3! because the operand layouts L_i share letters.

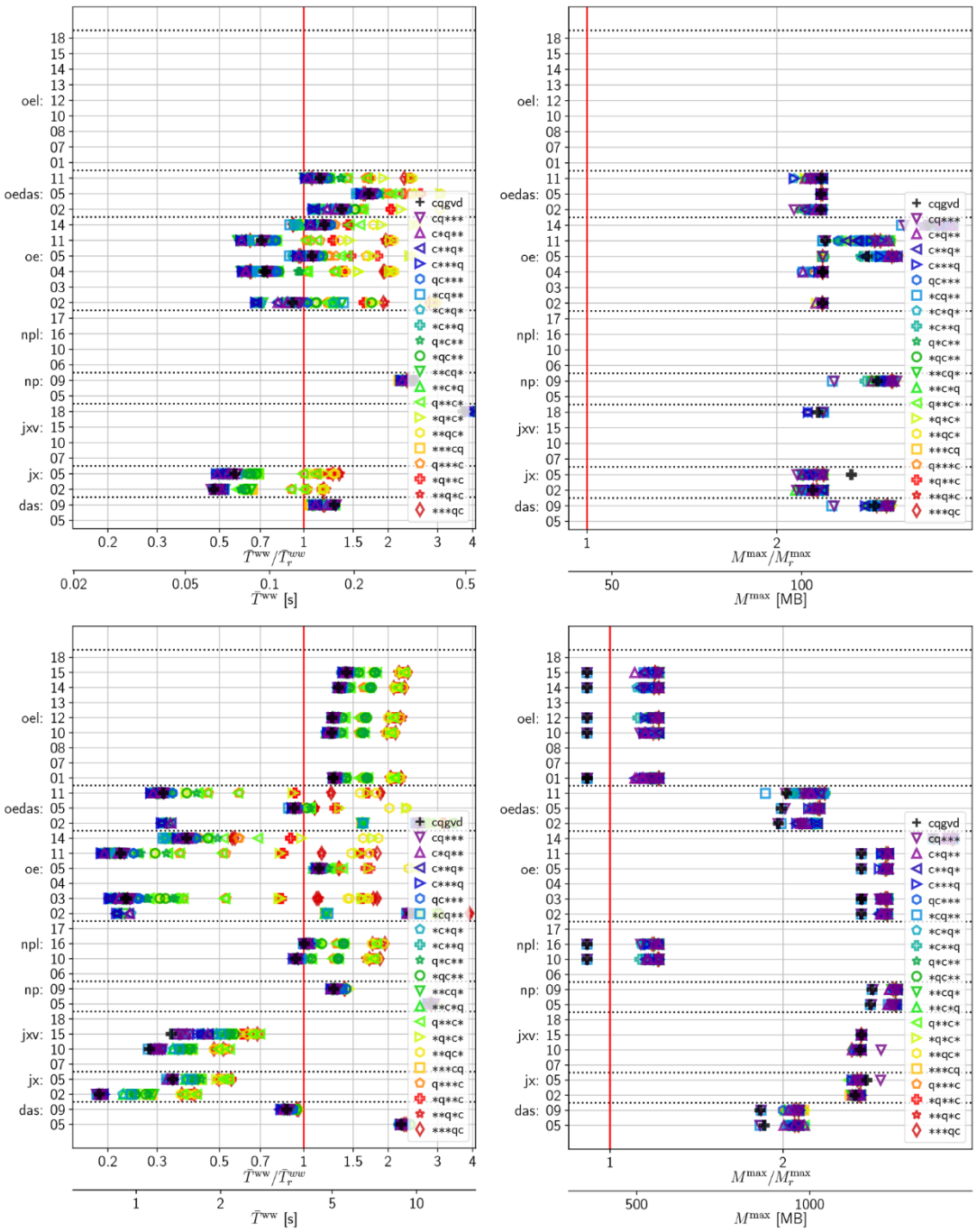


Fig. 5. Elapsed time and memory requirements of the weak convective term einsum expression depending on the operand memory layouts for 8192 cells, FE approximation orders 1 (top), 2 (bottom). Left: (relative) elapsed time mean without the worst case ($\bar{T}^{ww}/\bar{T}_r^{ww}$, $\bar{T}^{ww}$). Right: (relative) maximum memory consumption (M^{max}/M_r^{max}, M^{max}). The reference values obtained by SfePy are marked by vertical red lines. The libraries and contraction paths are indicated in y axis labels, see Table 9. The colors/markers indicate positions of cell and quadrature axes in the layout string, the SfePy layout is marked by black crosses among einsum expression backends. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

layouts are specified using permutations of the reduced default layout `cqgvd`, because the weak form has no material parameters (no `0`).

Elapsed time and memory requirements of the individual backends are shown in Fig. 5 for the FE approximation orders 1 (top), 2 (bottom). Both $\bar{T}^{ww}/\bar{T}_r^{ww}$ with $\bar{T}^{ww}$ and M^{max}/M_r^{max} with M^{max} are shown. The libraries and contraction paths are indicated in y axis labels. Again the values at most 4× slower than the reference SfePy implementation are shown, with the corresponding memory consumption. To reduce the color/marker legend size, only positions of `cq` axes are indicated in the legend, `*` denotes any letter from `gvd`. Abbreviations given in Table 9 are used to save space, the plus sign denotes addition of two einsum expressions due to the convective term nonlinearity.

6.4.3. *Layouts discussion*

Based on the above results, the following observations were made:

- For each approximation order, there was a single contraction path per library/optimization pair, independent of the layout.

Table 9
The mapping of convective term einsum expression contraction paths to short labels in Fig. 5.

contraction paths	abbreviations
-	-
01,02,12,03,02,01+01,02,04,13,02,01	01
01,02,12,03,02,01+06,04,02,03,02,01	02
03,34,34,03,01,01+03,12,01,12,02,01	03
03,34,34,03,01,01+15,13,14,03,02,01	04
03,34,34,03,01,01+34,05,14,23,01,01	05
24,34,04,13,02,01+01,02,04,02,01,01	06
24,34,04,13,02,01+01,02,04,13,01,01	07
24,34,04,13,02,01+05,05,02,03,02,01	08
24,34,04,13,02,01+06,04,02,03,02,01	09
24,34,04,13,02,01+06,13,14,23,02,01	10
24,34,04,13,02,01+15,13,14,13,02,01	11
24,34,04,13,02,01+24,25,34,03,12,01	12
24,34,24,01,12,01+24,01,03,13,01,01	13
24,34,24,03,02,01+34,45,34,23,02,01	14
56,35,01,23,01,01+34,01,34,13,02,01	15
56,35,01,23,01,01+34,05,14,23,02,01	16
56,35,23,01,02,01+34,25,34,01,02,01	17
56,35,23,01,02,01+34,25,34,01,20,01	18

- The layouts were both slower and faster than the default `cqgvd0`layout. Although there was not a single best layout, some patterns can be observed in Figs. 4, 5. According to Table 7, `cdgq` was 14 × , and `cgqd` 6× the fastest layout for the weak Laplacian, while, according to Table 8, `cqgdv` was 3 × , and `cgvdq` 2 × the fastest layout for the weak convective term.
- In the weak Laplacian case, 37.3%, of layouts were no more than 4 × slower and 3.2% were faster than the reference implementation for the approximation order 1 while for the approximation orders 2 it was 27.0% and 1.9%, respectively. For the approximation order 3, due to the 50 s time-out, all layouts times were within the 4 × limit and 49.1% were faster than the reference implementation. In the case of the weak convective term, 51.2% resp. 99.1% of layouts were no more than 4× slower and 20.1% resp. 44.8% were faster than the reference implementation for the approximation orders 1 resp. 2.
- In the weak Laplacian case, the jx backend performed the best in terms of the elapsed time, but required much more memory. For the approximation order 1, it was the only backend faster than the reference implementation, together with the hand-written Numba code, due to the low numerical intensity of the calculations. From the order 2, np and oe backends started to be competitive and for the order 3 also the loop-based npl and oel backends with their low memory footprint, as well as oedas.
- In the weak convective term case, being more computationally intensive, many oe backend layouts were faster than the reference implementation already for the approximation order 1, together with the again fastest jx layouts. For the order 2, also jxv and das, oedas (lower memory usage than oe) layouts out-performed the reference implementation.

We can conclude that alternative layouts can be used as another simple way of speeding the FE weak form evaluation calculations using einsum expressions, additional to optimized contraction paths.

6.5. Performance of selected weak forms contractions

A parametric study run using soops [8] lead to 6478 finished simulations[6], each evaluating a single weak form using a single backend/optimization combination several times according to the repeat parameter for various mesh sizes and function approximation orders, see Table 5. The default `cqgvd0` layout of the weak form einsum expression operands and result arrays was used.

First the overall results, i.e. the fastest backend for each weak form, number of cells and FE approximation order, are presented in Section 6.5.1. Because many backends/contraction paths performed similarly, we provide also a detailed comparison in Sections 6.5.3–6.5.7, where both the elapsed time differences among the backends/contraction paths are shown, alongside the memory consumption. The discussions of evaluation performance are included directly in the respective sections.

6.5.1. Overall results

The overall results are summarized in Table 10 for the residual mode and in Table 11 for the matrix mode evaluations. For each weak form, mesh size and FE approximation order, the shortest elapsed time mean without the worst case $\overline{T}^{\mathrm{ww}}$ across all backends is given, as well as the corresponding backend and $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$.

In the residual mode, the oe and np backends were the fastest for the weak Laplacian, oe, followed by np and jx for vector dot product weak form and mostly jx, with several occurrences of oe and np for the other three weak forms with the exception of the order 4 and 5 of the linear elastic form, where the loop-based backends npl and oel performed the best. According to the relative elapsed times means, the evaluations were mostly faster or comparable with the reference implementation, except the case of the linear elastic weak form.

In the case of the much more memory demanding matrix mode, the jx backend was mostly the fastest when evaluating the weak Laplacian, the convective and the linear elasticity weak forms. Notable exceptions occurred for the largest mesh size, where the memory-demanding jx was replaced by oedas (convective term) and oe (elasticity), see also the detailed analysis in the following sections. Note also that jx was much slower for small problems in the $\overline{T}$ metric, because of the JIT initialization in the first evaluation, omitted in $\overline{T}^{\mathrm{ww}}$. The jxv backend, followed by npl, were mostly the fastest in the case of the vector dot product form, while for the weighted vector dot product, the oe (most orders 1) and np (most orders 2) backends were the fastest.

Analogical tables summarizing the results of the simulations without the single-thread limitation are presented in Appendix A. A detailed comparison of the backends and tensors contraction paths follows in the next sections.

6.5.2. Detailed weak form analyses — common settings

Two kinds of figures are presented below for the five selected weak forms:

- To allow comparison among the five weak forms, the highest throughput of each backend is presented in Figs. 6–14. The throughput is defined as the size (in MB) of the resulting array of vectors (residual mode) or matrices (matrix mode) divided by the shortest elapsed time means without the worst case $\overline{T}^{\mathrm{ww}}$ for each backend, i.e. $|r|/\overline{T}^{\mathrm{ww}}$, resp. $|M|/\overline{T}^{\mathrm{ww}}$, corresponding to the highest number of megabytes each backend evaluates per second. The backends are indicated by markers, while the fastest einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.
- Relative elapsed time means without the worst case $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$ and relative maximum memory consumption $M^{\mathrm{max}}/M_r^{\mathrm{max}}$ are reported in Figs. 7–15 using the x axis logarithmic scale. The backends are indicated by markers, while the einsum contraction paths by color, the same as in the throughput figures. The cell counts (y axis) are grouped by the approximation orders. Only cases no more than 5× slower than the reference implementation are shown.

[6] This includes the reference implementation simulations shown in Fig. 2. 26 backend/optimization pairs, 25 mesh size/order combinations, 5 terms and 2 evaluation modes, 6500 in total, 22 failed due to insufficient memory.

Table 10
Summary results of residual evaluation mode. For each weak form, mesh size and FE approximation order, the shortest elapsed time mean without the worst case $\overline{T}^{ww}$ [s] across all backends is given, as well as the corresponding backend and $\overline{T}^{ww}/\overline{T}_r^{ww}$ (in parentheses).

#cells	order	Laplacian	v. dot	weighted v. dot	NS convective	elasticity
1024	1	0.01 (oe 0.9)	0.01 (oe 0.9)	0.01 (jx 1.0)	0.01 (jx 1.0)	0.01 (jx 1.1)
	2	0.01 (oe 0.9)	0.01 (oe 0.8)	0.01 (jx 0.9)	0.02 (jx 0.8)	0.03 (jx 1.5)
	3	0.02 (oe 0.6)	0.02 (jx 0.5)	0.02 (jx 0.5)	0.04 (jx 0.6)	0.12 (jx 1.6)
	4	0.06 (oe 0.5)	0.03 (jx 0.3)	0.03 (jx 0.4)	0.11 (jx 0.5)	0.41 (jx 1.8)
	5	0.15 (oe 0.4)	0.06 (jx 0.2)	0.07 (jx 0.3)	0.30 (jx 0.5)	0.74 (npl 1.2)
2048	1	0.01 (oe 1.1)	0.01 (jx 1.0)	0.01 (jx 1.0)	0.01 (jx 1.0)	0.02 (jx 1.1)
	2	0.02 (np 0.9)	0.02 (jx 0.8)	0.02 (jx 0.8)	0.02 (jx 0.6)	0.06 (jx 1.8)
	3	0.04 (np 0.5)	0.02 (jx 0.4)	0.03 (jx 0.5)	0.07 (jx 0.5)	0.22 (jx 1.7)
	4	0.11 (oe 0.4)	0.05 (jx 0.3)	0.06 (jx 0.3)	0.20 (jx 0.5)	0.81 (jx 1.8)
4096	1	0.01 (oe 0.9)	0.01 (np 1.0)	0.02 (jx 1.1)	0.01 (jx 0.9)	0.02 (jx 1.1)
	2	0.02 (oe 0.8)	0.02 (jx 0.6)	0.02 (jx 0.7)	0.03 (jx 0.5)	0.11 (jx 2.1)
	3	0.06 (oe 0.5)	0.04 (jx 0.4)	0.05 (jx 0.4)	0.12 (jx 0.5)	0.45 (jx 1.8)
8192	1	0.01 (oe 1.0)	0.01 (jx 1.0)	0.02 (jx 0.9)	0.02 (jx 0.7)	0.03 (jx 1.2)
	2	0.03 (oe 0.7)	0.03 (jx 0.6)	0.04 (jx 0.7)	0.06 (jx 0.6)	0.21 (jx 2.0)
	3	0.12 (oe 0.4)	0.07 (jx 0.3)	0.10 (jx 0.5)	0.23 (jx 0.5)	0.82 (jx 1.6)
16,384	1	0.02 (np 1.0)	0.02 (jx 0.9)	0.02 (jx 0.9)	0.02 (jx 0.7)	0.05 (jx 1.3)
	2	0.06 (np 0.7)	0.05 (jx 0.5)	0.07 (jx 0.7)	0.10 (jx 0.5)	0.41 (jx 2.0)
32,768	1	0.03 (oe 0.9)	0.03 (jx 0.9)	0.04 (jx 0.9)	0.04 (jx 0.6)	0.09 (jx 1.4)
	2	0.10 (oe 0.6)	0.08 (jx 0.5)	0.13 (jx 0.7)	0.19 (jx 0.5)	0.81 (jx 2.1)
65,536	1	0.04 (oe 0.9)	0.05 (jx 0.9)	0.08 (jx 1.1)	0.06 (jx 0.6)	0.17 (jx 1.3)
	2	0.20 (oe 0.6)	0.17 (jx 0.5)	0.29 (jx 0.8)	0.39 (jx 0.6)	1.61 (jx 2.1)
131,072	1	0.07 (oe 0.9)	0.08 (oe 0.9)	0.13 (jx 1.0)	0.12 (jx 0.6)	0.33 (jx 1.4)
262,144	1	0.13 (oe 1.0)	0.15 (oe 0.9)	0.30 (jx 1.1)	0.24 (jx 0.6)	0.66 (jx 1.4)
524,288	1	0.26 (oe 0.9)	0.30 (oe 0.9)	0.58 (jx 1.1)	0.47 (jx 0.6)	1.33 (jx 1.5)
1,048,576	1	0.50 (oe 0.9)	0.58 (oe 0.9)	1.13 (jx 1.1)	0.92 (jx 0.5)	2.63 (jx 1.5)

Table 11
Summary results of matrix evaluation mode. For each weak form, mesh size and FE approximation order, the shortest elapsed time mean without the worst case $\overline{T}^{ww}$ [s] across all backends is given, as well as the corresponding backend and $\overline{T}^{ww}/\overline{T}_r^{ww}$ (in parentheses).

#cells	order	Laplacian	v. dot	weighted v. dot	NS convective	elasticity
1024	1	0.01 (jx 1.0)	0.02 (jx 1.0)	0.02 (jx 1.0)	0.02 (jx 0.8)	0.03 (jx 1.4)
	2	0.04 (jx 1.0)	0.10 (np 0.5)	0.10 (np 0.5)	0.14 (jx 0.4)	0.37 (jx 1.5)
	3	0.24 (jx 0.6)	0.33 (npl 0.0)	0.73 (np 0.1)	0.83 (jx 0.1)	1.99 (jx 0.4)
	4	1.72 (jx 0.3)	1.29 (npl 0.0)	3.57 (jx 0.1)	4.38 (jxv 0.1)	13.89 (jx 0.4)
	5	5.07 (jx 0.2)	4.78 (npl 0.0)	15.53 (jx 0.1)	17.71 (jx 0.0)	46.30 (jx 0.3)
2048	1	0.01 (jx 1.0)	0.02 (jx 1.0)	0.02 (np 0.9)	0.02 (jx 0.7)	0.05 (jx 1.7)
	2	0.07 (jx 0.9)	0.18 (jx 0.5)	0.18 (np 0.5)	0.27 (jx 0.4)	0.70 (jx 1.4)
	3	0.49 (jx 0.7)	0.65 (npl 0.0)	1.45 (jx 0.1)	1.62 (jx 0.1)	4.01 (jx 0.4)
	4	3.48 (npl 0.3)	2.54 (npl 0.0)	7.18 (jx 0.1)	8.60 (jxv 0.0)	27.70 (jx 0.4)
4096	1	0.02 (jx 1.0)	0.05 (jx 1.4)	0.03 (oe 0.9)	0.04 (jx 0.6)	0.09 (jx 1.9)
	2	0.12 (jx 0.9)	0.39 (jxv 0.6)	0.34 (np 0.5)	0.53 (jx 0.4)	1.39 (jx 1.4)
	3	1.01 (jxv 0.7)	1.27 (npl 0.0)	2.92 (np 0.1)	3.31 (jx 0.1)	8.25 (jx 0.4)
8192	1	0.02 (jx 1.0)	0.06 (jx 1.2)	0.05 (oe 0.9)	0.08 (jx 0.7)	0.19 (jx 2.2)
	2	0.25 (jx 1.0)	0.69 (jxv 0.5)	0.66 (np 0.5)	1.05 (jx 0.4)	2.79 (jx 1.4)
	3	1.85 (jx 0.6)	2.55 (npl 0.0)	5.70 (jx 0.1)	6.50 (jx 0.1)	15.97 (jx 0.4)
16,384	1	0.04 (jx 1.1)	0.19 (jxv 1.9)	0.10 (oe 0.8)	0.14 (jx 0.7)	0.34 (jx 2.2)
	2	0.47 (jx 0.9)	1.21 (jxv 0.5)	1.33 (np 0.5)	2.04 (jxv 0.3)	5.44 (jx 1.4)
32,768	1	0.06 (jx 1.1)	0.28 (jxv 1.5)	0.18 (oe 0.8)	0.28 (jx 0.6)	0.67 (jx 2.3)
	2	0.95 (jx 1.0)	2.26 (jxv 0.4)	2.66 (np 0.5)	3.94 (jxv 0.3)	11.12 (jx 1.5)
65,536	1	0.13 (jx 1.2)	0.44 (jxv 1.2)	0.35 (oe 0.9)	0.54 (jx 0.7)	1.32 (jx 2.2)
	2	1.84 (jx 0.9)	4.74 (jxv 0.5)	5.29 (np 0.5)	7.70 (jxv 0.3)	22.29 (jx 1.4)
131,072	1	0.24 (jx 1.2)	0.79 (jxv 1.2)	0.68 (oe 0.9)	1.07 (jx 0.6)	2.62 (jx 2.3)
262,144	1	0.46 (jx 1.2)	1.49 (jxv 1.1)	1.39 (oe 0.9)	2.11 (jx 0.7)	5.23 (jx 2.3)
524,288	1	0.95 (jx 1.3)	2.87 (jxv 1.1)	2.79 (oe 0.9)	4.30 (jx 0.7)	10.55 (jx 2.4)
1,048,576	1	1.86 (jx 1.3)	5.64 (jxv 1.1)	5.56 (oe 0.9)	13.75 (oedas 1.0)	57.33 (oe 6.4)

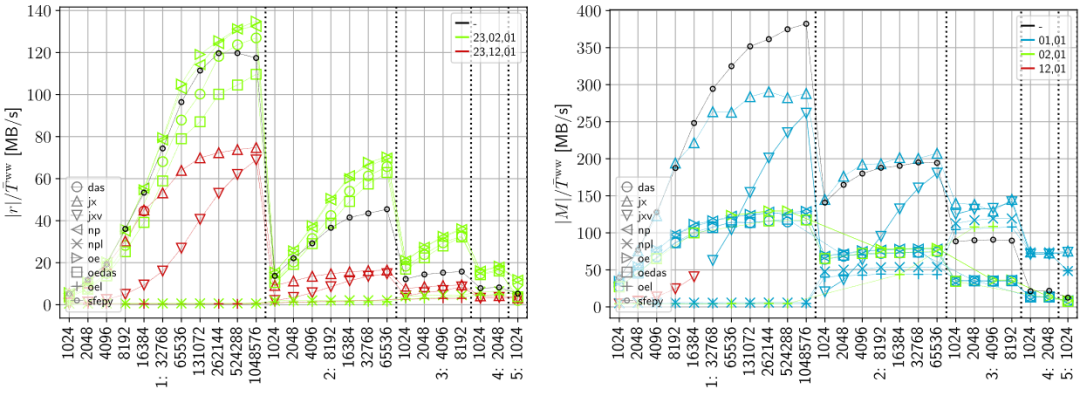


Fig. 6. The highest throughput of each backend (number of megabytes evaluated per second) for the weak Laplacian einsum expression. Left: residual mode evaluations $|r|/\overline{T}^{\mathrm{ww}}$, right: matrix mode evaluations $|M|/\overline{T}^{\mathrm{ww}}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

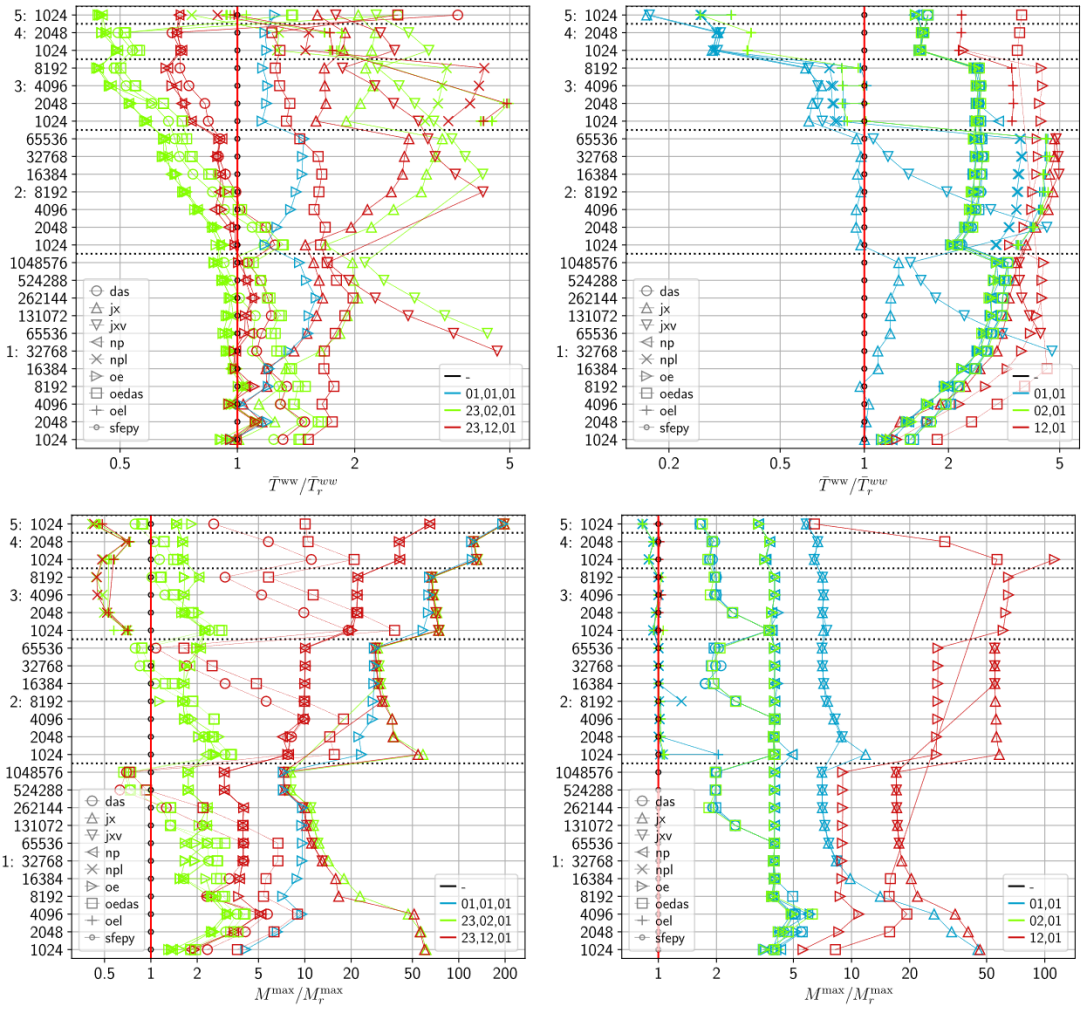


Fig. 7. Relative elapsed time and memory requirements of the weak Laplacian einsum expression w.r.t. the reference implementation. Left: residual mode, right: matrix mode, top: $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$, bottom: $M^{\max}/M_r^{\max}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (y axis) are grouped by the approximation orders.

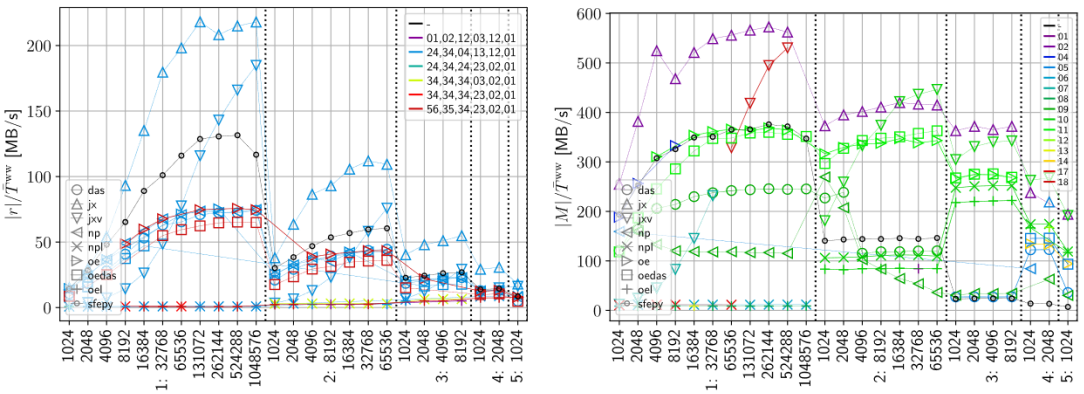


Fig. 8. The highest throughput of each backend (number of megabytes evaluated per second) for the weak convective term einsum expression. Left: residual mode evaluations $|r|/\overline{T}^{\mathrm{ww}}$, right: matrix mode evaluations $|M|/\overline{T}^{\mathrm{ww}}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

6.5.3. *Weak laplacian*

The performance of the einsum backends applied to the evaluation of the weak Laplacian einsum expression is shown in Figs. 6 (evaluation throughput) and 7 (relative performance).

In the residual mode, the oe backend was the fastest in most cases, followed by np and das with oedas. It was faster than the reference implementation even for the order 1. From the order 2, also das and oedas, and for the order 5 also npl, oel backends out-performed the reference implementation. The memory consumption was in general higher than the reference one for smaller data, while for large problems

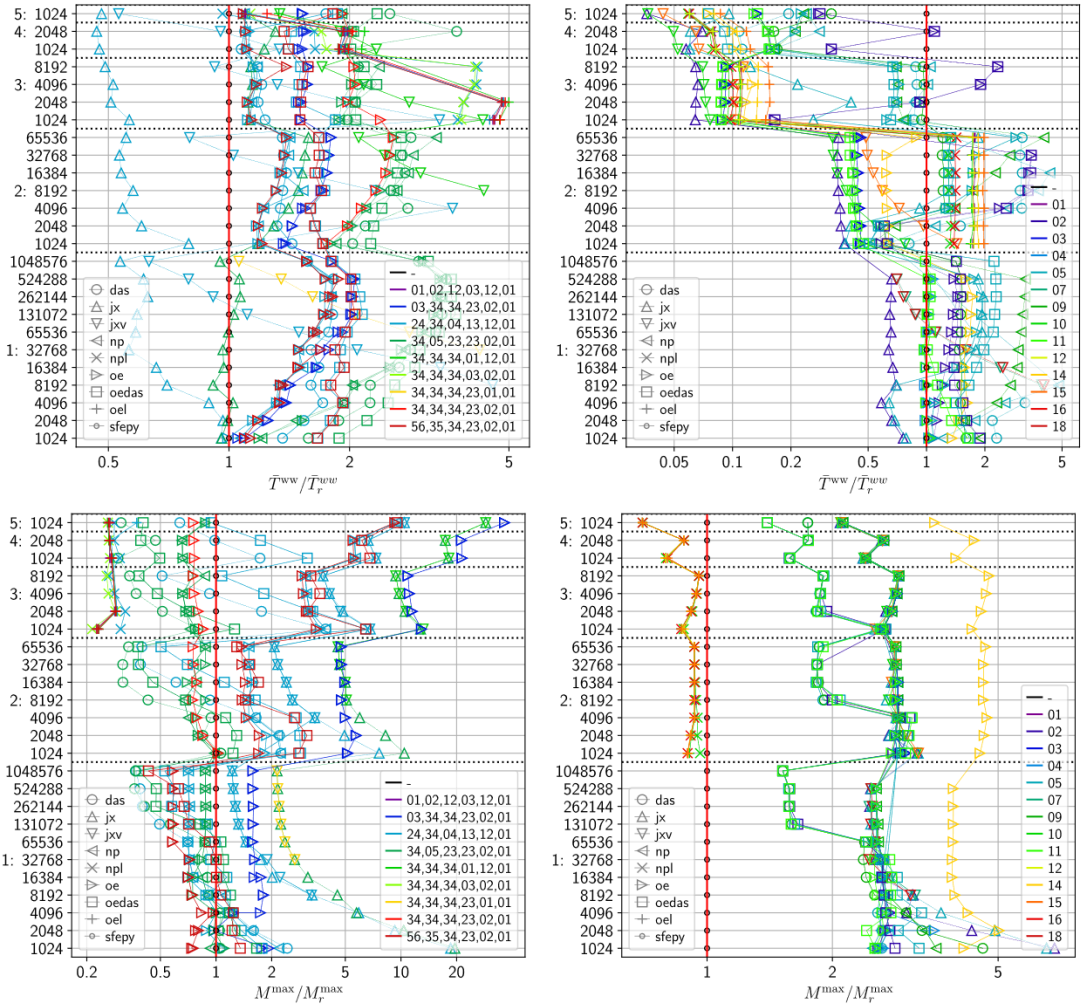


Fig. 9. Relative elapsed time and memory requirements of the weak convective term einsum expression w.r.t. the reference implementation. Left: residual mode, right: matrix mode, top: $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$, bottom: $M^{\max}/M_r^{\max}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (y axis) are grouped by the approximation orders.

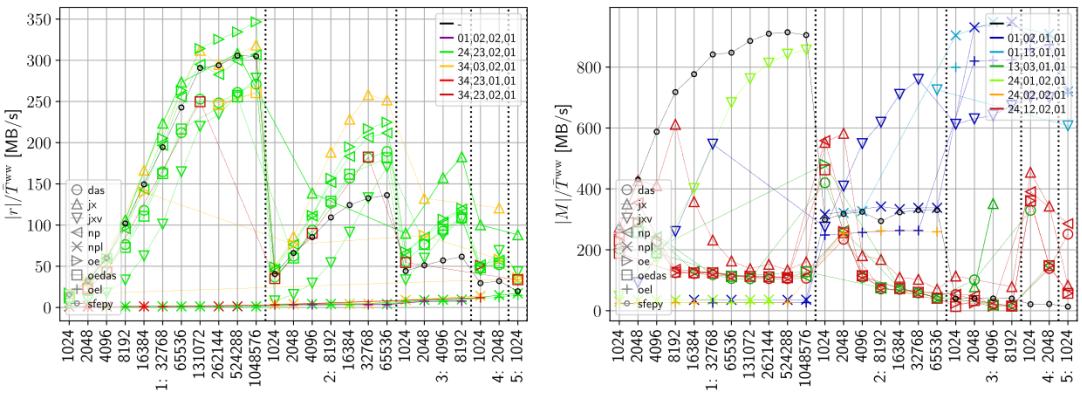


Fig. 10. The highest throughput of each backend (number of megabytes evaluated per second) for the vector dot product einsum expression. Left: residual mode evaluations $|r|/\overline{T}^{ww}$, right: matrix mode evaluations $|M|/\overline{T}^{ww}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

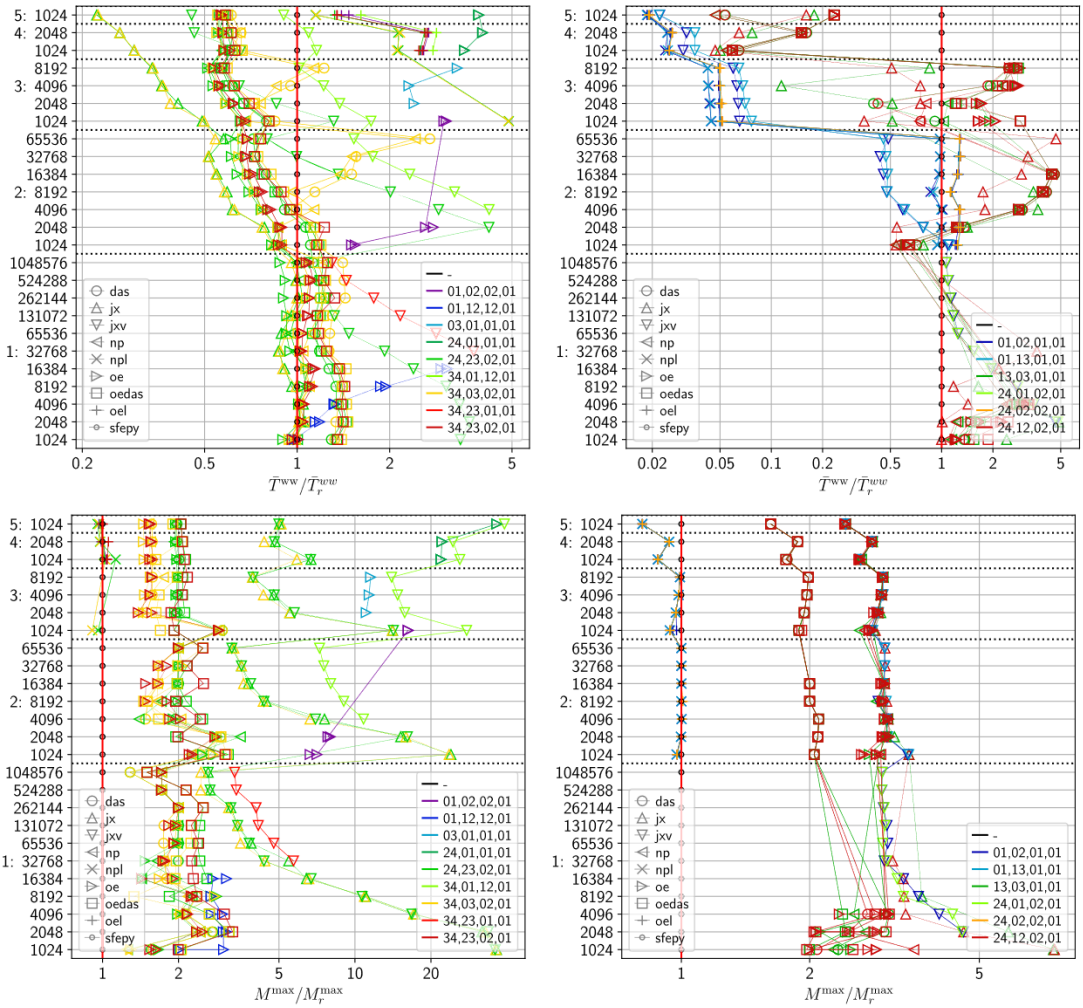


Fig. 11. Relative elapsed time and memory requirements of the vector dot product einsum expression w.r.t. the reference implementation. Left: residual mode, right: matrix mode, top: $\overline{T}^{ww}/\overline{T}_r^{ww}$, bottom: M^{max}/M_r^{max}. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (y axis) are grouped by the approximation orders.

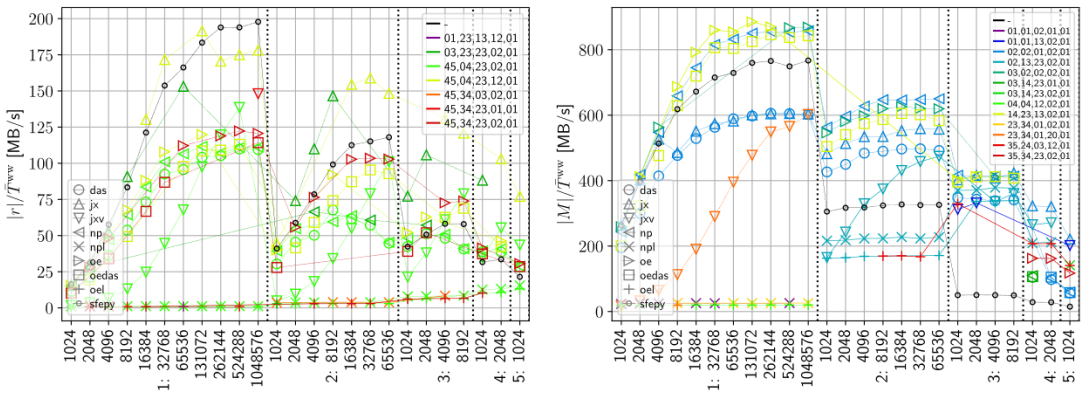


Fig. 12. The highest throughput of each backend (number of megabytes evaluated per second) for the weighted vector dot product einsum expression. Left: residual mode evaluations $|r|/\bar{T}^{ww}$, right: matrix mode evaluations $|M|/\bar{T}^{ww}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

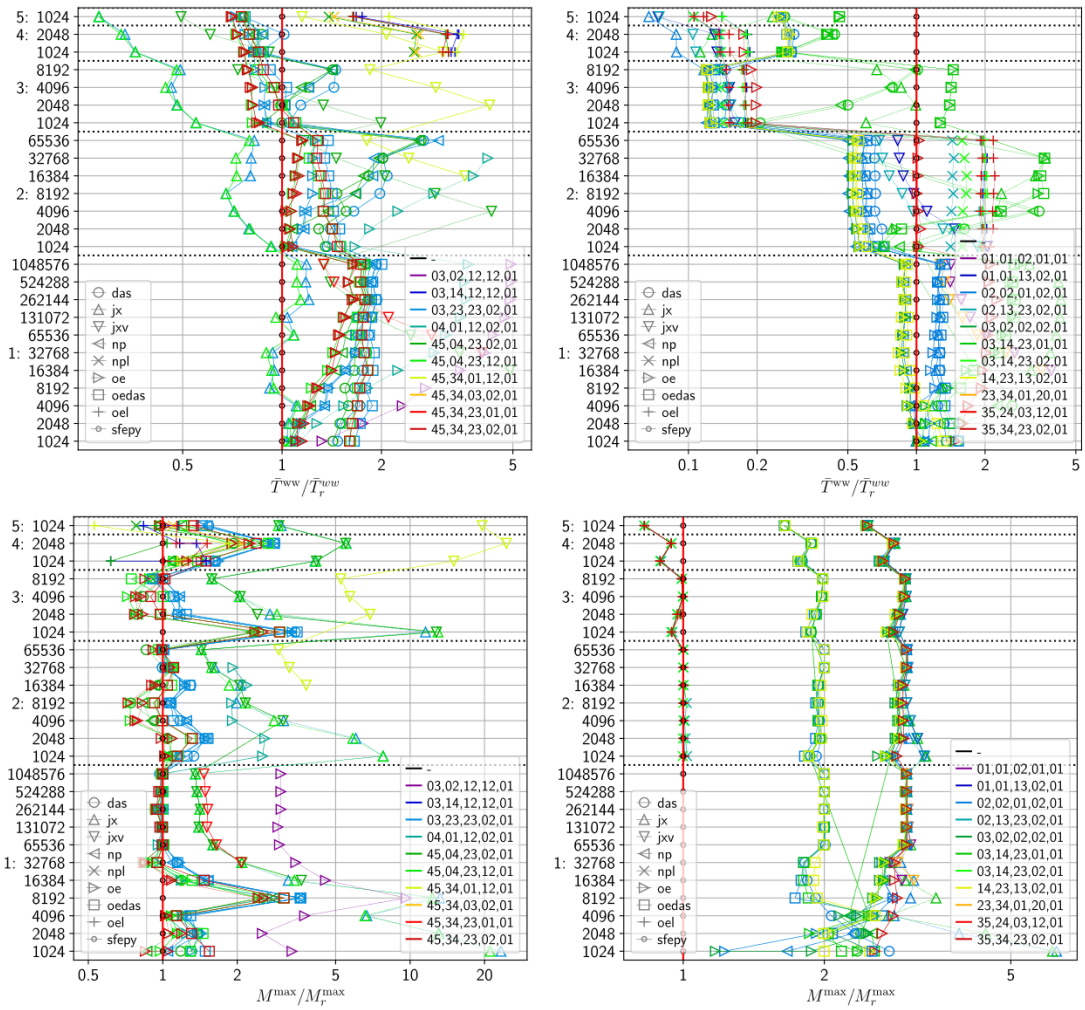


Fig. 13. Relative elapsed time and memory requirements of the weighted vector dot product einsum expression w.r.t. the reference implementation. Left: residual mode, right: matrix mode, top: $\bar{T}^{ww}/\bar{T}_r^{ww}$, bottom: M^{max}/M_r^{max}. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (y axis) are grouped by the approximation orders.

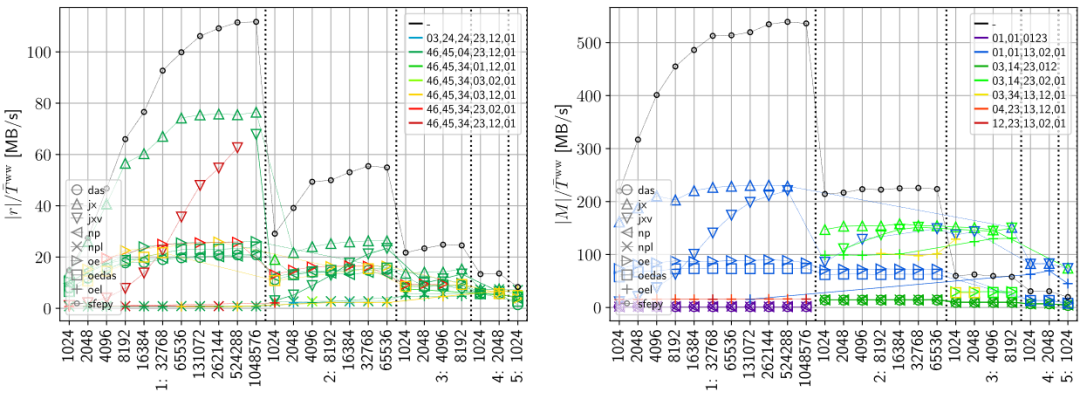


Fig. 14. The highest throughput of each backend (number of megabytes evaluated per second) for the weak linear elasticity term einsum expression. Left: residual mode evaluations $|r|/\bar{T}^{ww}$, right: matrix mode evaluations $|M|/\bar{T}^{ww}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

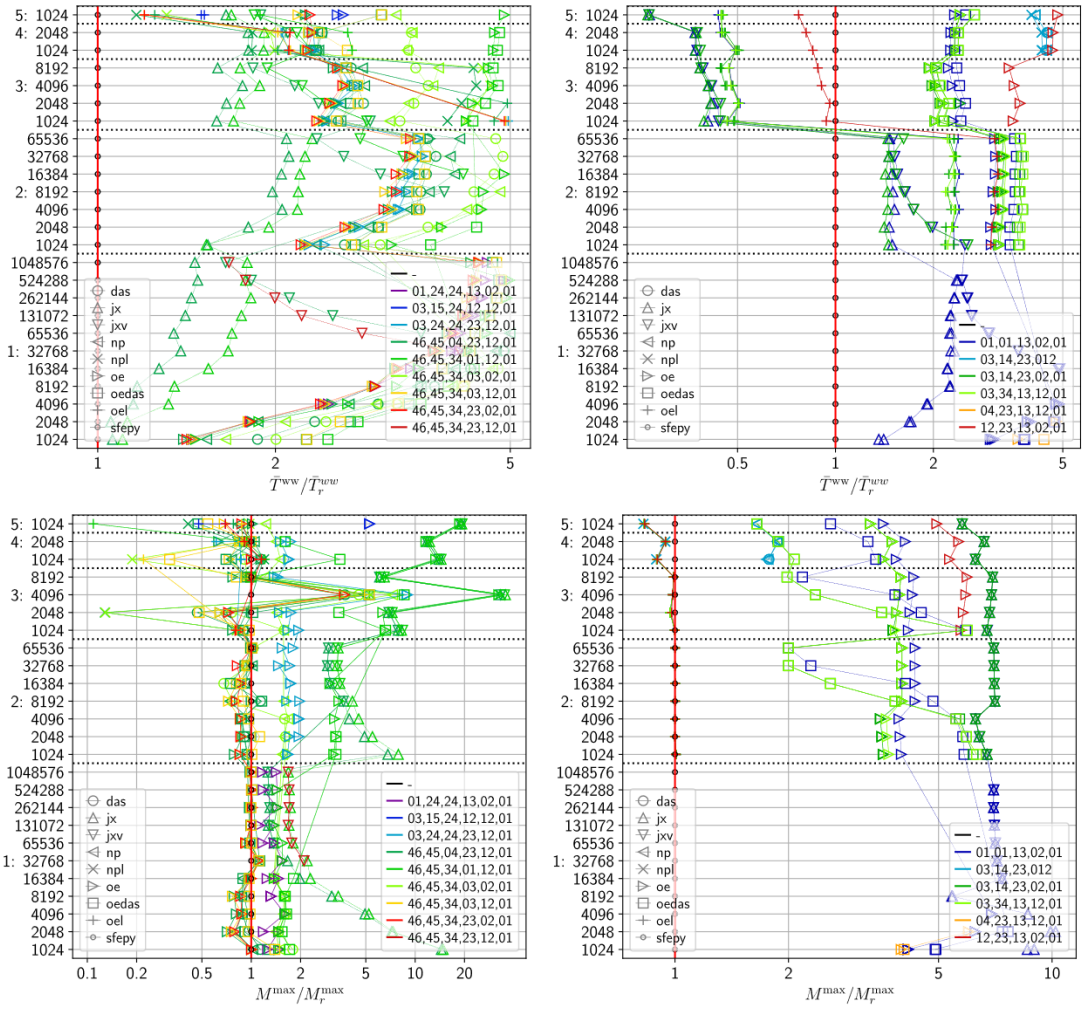


Fig. 15. Relative elapsed time and memory requirements of the weak linear elasticity term einsum expression w.r.t. the reference implementation. Left: residual mode, right: matrix mode, top: $\bar{T}^{ww}/\bar{T}_r^{ww}$, bottom: M^{max}/M_r^{max}. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (y axis) are grouped by the approximation orders.

several backends had a lower memory footprint: the das, oedas backends due to array chunking, and npl, oel thanks to looping over the mesh cells.

In the matrix mode, the jx backend was the fastest and out-performed the reference implementation from the order 2. From the order 3, also npl, oel and jxv backends were faster than the reference implementation. The cell loop backends (npl, oel) performance seemed to be less dependent on the particular contraction paths. We attribute this to the fact that the restriction of the operands to a cell is small and so also the temporary allocations are small — compare with the huge memory consumption differences among the contraction paths of other backends. The npl backend was faster than oel, probably due to a simpler code path, important again for the small cell-restricted data.

6.5.4. Weak convective term

The performance of the einsum backends applied to the evaluation of the weak convective term einsum expression is shown in Figs. 8 (evaluation throughput) and 9 (relative performance).

In the residual mode, the jx backend was the fastest by a large margin, followed by jxv. The jx backend out-performed the reference implementation, while jxv only in some cases and all the other backends were slower (mostly no more than $2\times$). However, the fastest contraction paths memory consumption of jx, jxv was up to $10\times$ more than reference consumption. For higher order approximations, npl, oel, das, oedas but also eo and np approached the reference times and had lower memory consumption.

In the matrix mode, the jx backend was again the fastest, but its lead over jxv, oedas or oe was small from the order 2. The jx backend was faster than the reference implementation from the order 1, jxv, oedas and oe from the order 2 and essentially all backends from the order 3. The npl, oel backends performed very well from the order 3, and unlike other backends (especially jx, jxv) had a lower memory consumption than the reference implementation. Overall the memory consumption stayed less than $5\times$ reference, in contrast to the weak Laplacian in the previous section.

6.5.5. Vector dot product

The performance of the einsum backends applied to the evaluation of the vector dot product einsum expression is shown in Figs. 10 (evaluation throughput) and 11 (relative performance).

In the residual mode, the jx backend was the fastest, together with oe for the order 1, where it out-performed the reference implementation. From the order 2 also np, oedas, das and from the order 3 jxv backends were faster than the reference code. The memory consumption mostly stayed less than $5\times$ reference, except jx, jxv and oe for some contraction paths.

In the matrix mode, jxv replaced jx as the overall fastest backend for orders 1, 2, but from the order 3 it was replaced by npl and oel. The jxv, npl backends were faster than the reference implementation from the order 2, jx from the order 3 and das, oedas from the order 4. The memory requirements stayed within $3\times$ reference, npl and oel backends had memory consumption lower than reference from the order 3.

6.5.6. Weighted vector dot product

The performance of the einsum backends applied to the evaluation of the weighted vector dot product einsum expression is shown in Figs. 12 (evaluation throughput) and 13 (relative performance).

In the residual mode, the jx backend was the fastest, followed by oe and np. From the order 2 it consistently out-performed the reference implementation, from the order 3 also oe, np, oedas and jxv and from the order 4 also das. The memory consumption stayed mostly less than $5\times$ reference.

In the matrix mode, most backends performed well. The oe and np backends were the fastest in most cases and out-performed the reference implementation. From the order 2 also jx, oedas, das and jxv, from the order 3 npl and oel were faster than the reference code. The jx and jxv backends were the fastest from the order 4. The memory consumption stayed mostly less than $3\times$ reference, the npl and oel backends had smaller footprint from the order 3.

6.5.7. Linear elasticity

The performance of the einsum backends applied to the evaluation of the weak linear elasticity term einsum expression is shown in Figs. 14 (evaluation throughput) and 15 (relative performance).

In the residual mode, the jx backend was the fastest but mostly almost $2\times$ slower than the reference implementation. For the order 5, the npl and oel backends performed better than jx, but they were still slower than the reference code. The memory consumption of oel, npl and das was smaller and of oedas, das smaller or comparable to the reference implementation. Overall it mostly stayed less than $5\times$ reference, except for jx and jxv from the order 3, where it reached almost $30\times$ reference values.

In the matrix mode, the jx, jxv backends performed the best and were faster than reference from the order 3, together with the oel backend. The memory consumption was mostly less than $7\times$ reference values.

The relatively low performance in this case, especially in the residual mode, was caused by not exploiting the sparsity of the `Psg` tensor, see the last paragraph of Section 5.3.3.

7. Conclusion

In this paper, we proposed and described a simple transpiler from generalized einsum-like expressions, suitable for describing multi-linear finite element weak forms, to regular tensor contraction expressions. We applied the transpiler and presented a very simple implementation of several standard weak forms. The transpiler supports several Python libraries as backends implementing the einsum function which allowed us to perform a large numerical study comparing the backends (NumPy, opt_einsum, Dask and JAX) mutually and with the reference implementations of the weak forms in the finite element SfePy.

An indispensable feature of the backends was their ability to optimize the tensor contraction paths. Our study confirmed that using contraction paths optimizations is crucial for good performance. However, the theoretical speedup, as e.g. calculated by opt_einsum, was found to be not a good metric for estimating performance when applied to simple einsum expressions occurring in the FE calculations — here the additional memory consumed by temporary storage was much more important. This was also one of the motivations for realizing the presented numerical study.

Concerning the study results, we first studied the influence of the memory layout of the expression operands on the evaluation performance, to establish the position of the default, SfePy compatible layout, with other possible layouts. We found that marked improvements could be achieved by using alternative memory layouts.

Then five selected weak forms were evaluated for various mesh sizes and FE approximation orders using the default layout. The evaluations were repeated several times for each parameter set and statistics such as means without the worst case were computed.

Significant differences in performance were found between the residual (operator/matrix application to a DOF vector without the global sparse matrix assembly) and matrix (calculating local element matrices) evaluation modes as well as among the various weak forms — no single

backend/contraction path was the best in all the cases.

Overall, JAX based backends were the fastest, but very memory demanding. The JIT compilation warm-up was disregarded by omitting the worst case repetition in calculating the elapsed time mean. Without this, the JAX backends were much slower for small problems. GPU/TPU performance was not considered in this study.

A good compromise between speed and memory overhead w.r.t. the reference implementation was achieved by using the opt_einsum or NumPy backends. Especially their cell-loop based variants had very low memory requirements and offered good performance for FE approximation orders greater or equal to three in the matrix evaluation mode. Also the Dask-based backends were suitable for large problems, where the automatic chunking of operands decreased the memory consumption. Dask allows out-of-core calculations with data not fitting into RAM, but this feature was not used in this study.

In many cases the einsum expression based evaluations were faster than the reference implementation (a hand-crafted C code, called from Python via SfePy's lightweight Cython wrappers) which is in our opinion a good result considering the simplicity of the transpiler implementation and especially of the weak form definitions. For forms with low numerical intensity (e.g. the weak Laplacian in matrix mode) or unused sparsity (the linear elasticity term), going beyond einsum might be required for further speed-up. A very preliminary result included in the memory layouts study indicates that using Numba might be promising.

The transpiler and einsum expression based weak form implementations are available in SfePy from version 2021.1, allowing a rapid prototyping of multi-physical finite element models and subsequent calculations in various fields such as biomechanics [30] or solid state physics [22]. All data used in preparation of this paper are available online [9].

CRediT authorship contribution statement

Robert Cimrman: Conceptualization, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The work was supported from European Regional Development Fund — Project "Application of Modern Technologies in Medicine and Industry" (No. CZ.02.1.01/0.0/0.0/17_048/0007280).

Appendix A. Performance of Selected Weak Forms Contractions with Threads

This appendix complements Section 6.5 by using the same setup but the single-threaded execution was not enforced — default settings of each backend library were in effect. Two additional backends — dat and oedat — were included, i.e. Dask based backends with multiple threads enabled.

A parametric study run using soops [8] lead to 8326 finished simulations[7], each evaluating a single weak form using a single backend/optimization combination several times according to the repeat parameter for various mesh sizes and function approximation orders, see Table 5. The default `cqgvd0` layout of the weak form einsum expression operands and result arrays was used.

The overall results are summarized in Table A.12 for the residual mode and in Table A.13 for the matrix mode evaluations. The evaluation throughput of the einsum backends applied to the evaluation of five selected einsum expression is shown in Figs. A.16–A.20.

Table A.12
Summary results of residual evaluation mode, no single-thread limitation. For each weak form, mesh size and FE approximation order, the shortest elapsed time mean without the worst case $\overline{T}^{\mathrm{ww}}$ [s] across all backends is given, as well as the corresponding backend and $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$ (in parentheses).

#cells	order	Laplacian	v. dot	weighted v. dot	NS convective	elasticity
1024	1	0.01 (oe 1.0)	0.01 (oe 1.0)	0.01 (jx 1.0)	0.01 (jx 1.0)	0.02 (jx 1.2)
	2	0.01 (np 0.9)	0.01 (np 0.9)	0.02 (jx 1.0)	0.02 (jx 0.9)	0.04 (jx 1.6)
	3	0.02 (oe 0.6)	0.02 (jx 0.4)	0.02 (jx 0.6)	0.04 (jx 0.6)	0.09 (jx 1.3)
	4	0.05 (dat 0.3)	0.02 (jx 0.2)	0.03 (jx 0.3)	0.11 (dat 0.5)	0.20 (dat 0.8)
	5	0.12 (oedat 0.3)	0.03 (jx 0.1)	0.04 (jx 0.2)	0.19 (dat 0.3)	0.25 (oedat 0.4)
2048	1	0.01 (oe 1.0)	0.01 (np 1.0)	0.01 (jx 1.1)	0.02 (jx 1.1)	0.02 (jx 1.4)
	2	0.02 (oe 0.8)	0.01 (jx 0.7)	0.02 (jx 0.8)	0.03 (jx 0.8)	0.06 (jx 1.7)
	3	0.04 (oe 0.5)	0.02 (jx 0.3)	0.03 (jx 0.4)	0.07 (jx 0.5)	0.17 (jx 1.2)
	4	0.06 (dat 0.2)	0.03 (jx 0.1)	0.04 (jx 0.2)	0.15 (dat 0.3)	0.22 (dat 0.5)
4096	1	0.01 (oe 1.0)	0.01 (np 1.0)	0.02 (oe 1.2)	0.02 (jx 1.2)	0.03 (jx 1.5)
	2	0.02 (oe 0.8)	0.02 (jx 0.6)	0.02 (jx 0.7)	0.04 (jx 0.7)	0.09 (jx 1.6)
	3	0.05 (dat 0.3)	0.03 (jx 0.2)	0.04 (jx 0.4)	0.12 (jx 0.5)	0.25 (dat 0.9)
	4	0.09 (oedat 0.2)	0.04 (jx 0.1)	0.08 (jx 0.2)	0.19 (dat 0.2)	0.32 (oedat 0.3)
8192	1	0.01 (np 1.0)	0.02 (jx 1.0)	0.02 (jx 1.0)	0.02 (jx 1.0)	0.03 (jx 1.3)
	2	0.04 (oe 0.7)	0.02 (jx 0.5)	0.03 (jx 0.6)	0.06 (jx 0.6)	0.17 (jx 1.5)
	3	0.06 (dat 0.2)	0.04 (jx 0.2)	0.08 (jx 0.3)	0.16 (dat 0.3)	0.28 (dat 0.5)
16,384	1	0.02 (oe 1.0)	0.02 (jx 0.9)	0.03 (jx 0.9)	0.03 (jx 1.0)	0.06 (jx 1.5)
	2	0.04 (dat 0.5)	0.03 (jx 0.4)	0.06 (jx 0.6)	0.10 (jx 0.5)	0.27 (dat 1.3)
	3	0.10 (dat 0.2)	0.06 (jx 0.1)	0.12 (jx 0.3)	0.23 (dat 0.2)	0.42 (oedat 0.4)

(*continued on next page*)

[7] This includes the reference implementation simulations shown in Fig. 2. 31 backend/optimization pairs, 27 mesh size/order combinations, 5 terms and 2 evaluation modes, 8370 in total, 44 failed due to insufficient memory.

Table A.12 (*continued*)

#cells	order	Laplacian	v. dot	weighted v. dot	NS convective	elasticity
32,768	1	0.03 (oe 1.0)	0.02 (jx 0.8)	0.04 (jx 0.9)	0.04 (jx 0.7)	0.10 (jx 1.5)
	2	0.06 (dat 0.4)	0.05 (jx 0.3)	0.10 (jx 0.5)	0.15 (dat 0.4)	0.33 (dat 0.8)
65,536	1	0.04 (oe 0.9)	0.03 (jx 0.6)	0.07 (jx 0.9)	0.07 (jx 0.7)	0.18 (jx 1.4)
	2	0.10 (dat 0.3)	0.09 (jx 0.3)	0.19 (jx 0.5)	0.22 (dat 0.3)	0.50 (oedat 0.6)
131,072	1	0.07 (dat 0.9)	0.05 (jx 0.6)	0.11 (jx 0.8)	0.12 (jx 0.6)	0.33 (jx 1.4)
262,144	1	0.08 (dat 0.6)	0.10 (jx 0.6)	0.22 (jx 0.8)	0.24 (jx 0.6)	0.57 (dat 1.2)
524,288	1	0.15 (oedat 0.5)	0.18 (jx 0.6)	0.42 (jx 0.8)	0.38 (dat 0.5)	0.87 (dat 0.9)
1,048,576	1	0.23 (dat 0.4)	0.35 (jx 0.6)	0.81 (jx 0.8)	0.60 (dat 0.4)	1.68 (oedat 0.9)

Table A.13
Summary results of matrix evaluation mode, no single-thread limitation. For each weak form, mesh size and FE approximation order, the shortest elapsed time mean without the worst case $\overline{T}^{\mathrm{ww}}$ [s] across all backends is given, as well as the corresponding backend and $\overline{T}^{\mathrm{ww}}/\overline{T}_r^{\mathrm{ww}}$ (in parentheses).

#cells	order	Laplacian	v. dot	weighted v. dot	NS convective	elasticity
1024	1	0.01 (jx 1.0)	0.01 (jx 0.9)	0.02 (oe 1.0)	0.02 (jx 0.8)	0.03 (jx 1.6)
	2	0.04 (jx 1.0)	0.06 (jx 0.3)	0.07 (jx 0.4)	0.08 (jx 0.2)	0.27 (jx 1.1)
	3	0.17 (jx 0.4)	0.30 (jxv 0.0)	0.30 (jx 0.0)	0.38 (jx 0.0)	0.97 (jx 0.2)
	4	0.51 (jx 0.1)	0.88 (jxv 0.0)	1.09 (jx 0.0)	1.33 (jxv 0.0)	3.71 (jx 0.1)
	5	1.33 (jx 0.0)	2.58 (jxv 0.0)	3.50 (jx 0.0)	3.92 (jxv 0.0)	11.56 (jx 0.1)
2048	1	0.02 (jx 1.1)	0.02 (jx 0.9)	0.02 (oe 0.9)	0.03 (jx 0.7)	0.05 (jx 1.7)
	2	0.06 (jx 0.9)	0.10 (jx 0.3)	0.11 (jx 0.3)	0.14 (jx 0.2)	0.50 (jx 1.0)
	3	0.33 (jx 0.4)	0.51 (jxv 0.0)	0.56 (jx 0.0)	0.72 (jx 0.0)	1.92 (jx 0.2)
	4	1.01 (jx 0.1)	1.71 (jxv 0.0)	2.12 (jx 0.0)	2.46 (jxv 0.0)	7.28 (jx 0.1)
4096	1	0.02 (jx 1.2)	0.03 (jx 0.9)	0.03 (jx 0.8)	0.03 (jx 0.5)	0.09 (jx 1.8)
	2	0.11 (jx 0.8)	0.18 (jx 0.3)	0.21 (jx 0.3)	0.25 (jx 0.2)	0.99 (jx 1.0)
	3	0.66 (jx 0.4)	1.00 (jxv 0.0)	1.12 (jx 0.0)	1.37 (jxv 0.0)	3.80 (jx 0.2)
	4	1.95 (jx 0.1)	3.30 (jxv 0.0)	4.15 (jx 0.0)	7.09 (oedat 0.0)	44.14 (oedat 0.3)
8192	1	0.03 (jx 1.1)	0.05 (jx 0.9)	0.05 (oe 0.9)	0.06 (jx 0.5)	0.15 (jx 1.9)
	2	0.20 (jx 0.8)	0.34 (jx 0.3)	0.39 (jx 0.3)	0.48 (jx 0.2)	1.93 (jx 1.0)
	3	1.22 (jx 0.4)	1.84 (jxv 0.0)	2.10 (jx 0.0)	2.60 (jxv 0.0)	7.45 (jx 0.2)
16,384	1	0.04 (jx 1.1)	0.08 (jx 0.8)	0.10 (oe 0.9)	0.11 (jx 0.5)	0.30 (jx 1.9)
	2	0.40 (jx 0.8)	0.67 (jx 0.3)	0.75 (jx 0.3)	0.93 (jx 0.2)	3.85 (jx 1.0)
	3	2.05 (dat 0.3)	3.60 (jxv 0.0)	4.17 (jx 0.0)	9.03 (oedat 0.0)	32.33 (oedat 0.4)
32,768	1	0.06 (jx 1.0)	0.13 (jx 0.7)	0.17 (jx 0.8)	0.19 (jx 0.5)	0.56 (jx 1.9)
	2	0.51 (dat 0.5)	1.34 (jx 0.3)	1.58 (jx 0.3)	1.78 (jx 0.1)	5.93 (oedat 0.8)
65,536	1	0.11 (jx 1.1)	0.26 (jx 0.7)	0.33 (jx 0.8)	0.36 (jx 0.4)	1.11 (jx 1.9)
	2	0.86 (oedat 0.4)	2.60 (jxv 0.2)	3.11 (jx 0.3)	3.55 (jx 0.1)	10.01 (oedat 0.6)
131,072	1	0.20 (jx 1.1)	0.49 (jx 0.7)	0.63 (jx 0.8)	0.66 (jx 0.4)	2.16 (jx 1.9)
262,144	1	0.40 (jx 1.1)	0.98 (jxv 0.7)	1.21 (oedat 0.8)	1.32 (jx 0.4)	3.93 (oedat 1.7)
524,288	1	0.54 (dat 0.7)	1.84 (jxv 0.7)	2.17 (oedat 0.7)	2.60 (jx 0.4)	6.53 (oedat 1.4)
1,048,576	1	0.89 (dat 0.6)	3.65 (jxv 0.7)	4.07 (oedat 0.6)	17.78 (oedas 1.4)	24.93 (oedat 2.7)

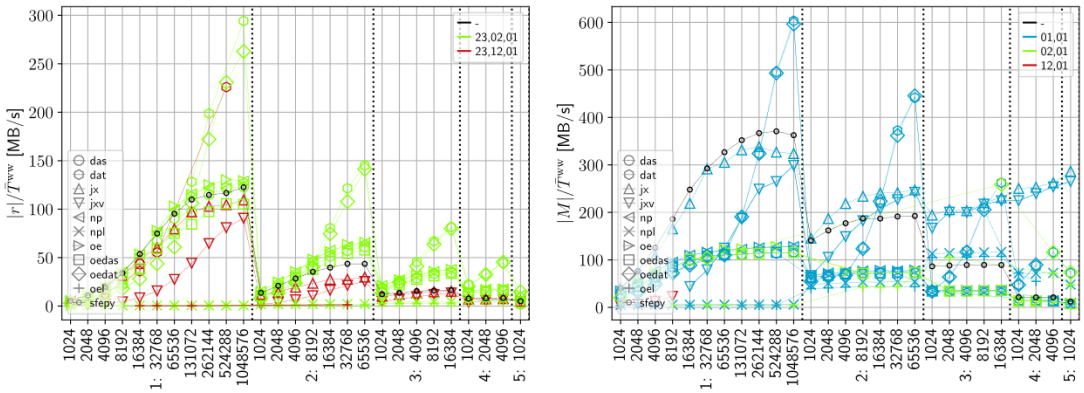


Fig. A.16. The highest throughput of each backend (number of megabytes evaluated per second) for the weak Laplacian einsum expression with allowed threads. Left: residual mode evaluations $|r|/\overline{T}^{\mathrm{ww}}$, right: matrix mode evaluations $|M|/\overline{T}^{\mathrm{ww}}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

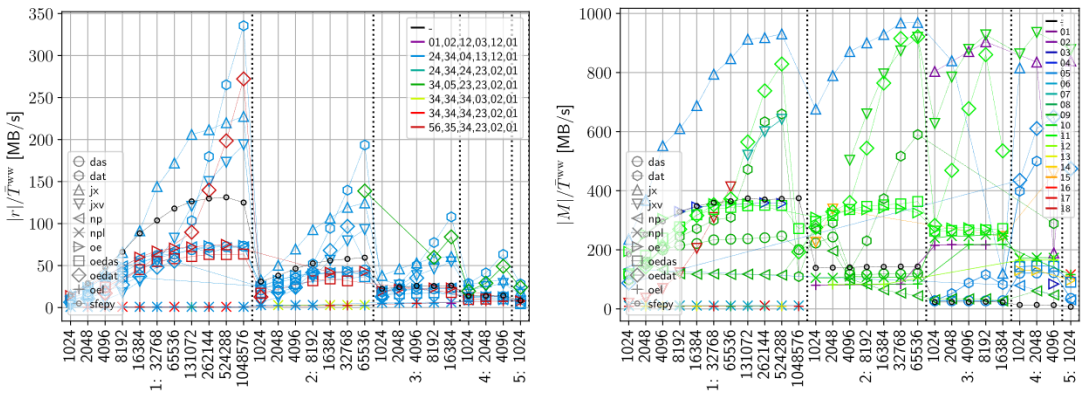


Fig. A.17. The highest throughput of each backend (number of megabytes evaluated per second) for the weak convective term einsum expression with allowed threads. Left: residual mode evaluations $|r|/\overline{T}^{\mathrm{ww}}$, right: matrix mode evaluations $|M|/\overline{T}^{\mathrm{ww}}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

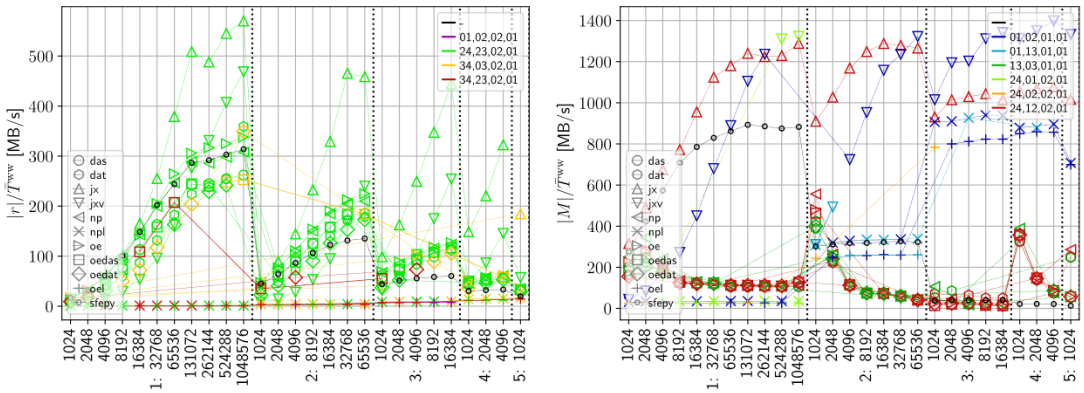


Fig. A.18. The highest throughput of each backend (number of megabytes evaluated per second) for the vector dot product einsum expression with allowed threads. Left: residual mode evaluations $|r|/\overline{T}^{\mathrm{ww}}$, right: matrix mode evaluations $|M|/\overline{T}^{\mathrm{ww}}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

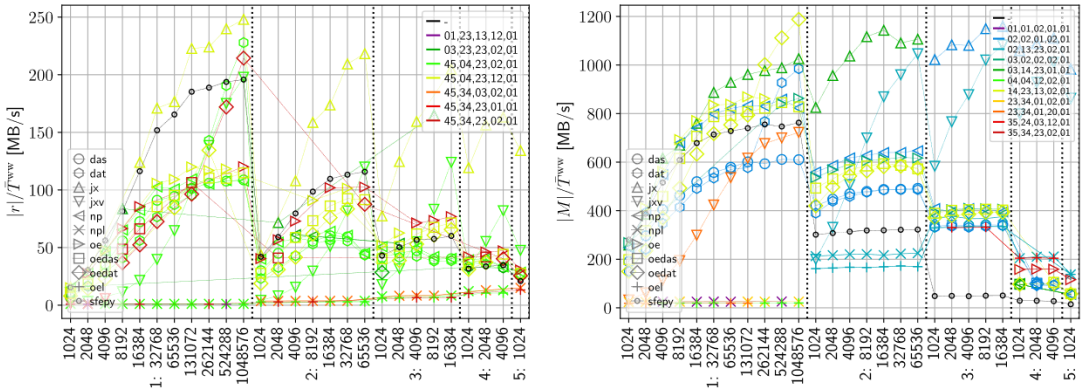


Fig. A.19. The highest throughput of each backend (number of megabytes evaluated per second) for the weighted vector dot product einsum expression with allowed threads. Left: residual mode evaluations $|r|/\overline{T}^{\mathrm{ww}}$, right: matrix mode evaluations $|M|/\overline{T}^{\mathrm{ww}}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

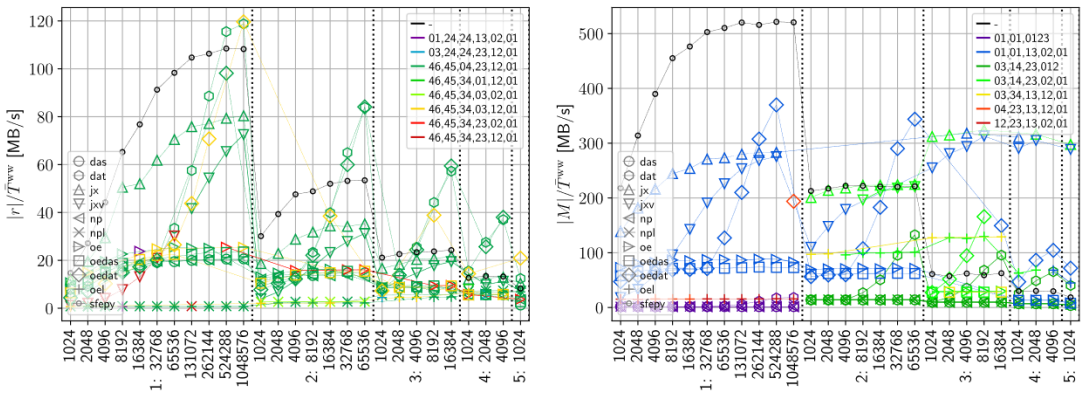


Fig. A.20. The highest throughput of each backend (number of megabytes evaluated per second) for the weak linear elasticity term einsum expression with allowed threads. Left: residual mode evaluations $|r|/\bar{T}^{ww}$, right: matrix mode evaluations $|M|/\bar{T}^{ww}$. The backends are indicated by markers, the einsum contraction paths by color. The cell counts (x axis) are grouped by the approximation orders.

The performance of most backends was similar to the single-threaded case with the exception of the JAX-based backends jx and jxv. Those backends had a significantly increased performance for large problems, both in terms of the mesh size (the number of cells) and the FE approximation order, most visible in the matrix mode evaluations. The new backends dat and oedat also performed very well for the largest problems considered, often replacing jx in Table A.12 (Laplacian, NS convective and linear elasticity weak forms) and Table A.13 (Laplacian, weighted vector dot and linear elasticity weak forms). Their performance, when applied to the same contraction paths, was very similar.

By comparing with data in Section 6.5, we can conclude that enabling threads resulted in about halved times for the largest problems considered.

References

[1] Alnæs MS, Logg A, Ølgaard KB, Rognes ME, Wells GN. Unified form language: a domain-specific language for weak formulations of partial differential equations. ACM Trans Math Software 2014;40(2). 9:1–9:37, doi:10.1145/2566630.

[2] Babuška I, Guo BQ. The h, p and h-p version of the finite element method; basis theory and applications. Adv Eng Softw 1992;15(3):159–74. https://doi.org/10.1016/0965-9978(92)90097-Y.

[3] Barra V, Brown J, Thompson J, Dudouit Y. High-performance operator evaluations with ease of use: libCEED's Python interface. Python in science conference, Austin, Texas. 2020. p. 85–90. https://doi.org/10.25080/Majora-342d178e-00c.URL https://conference.scipy.org/proceedings/scipy2020/libceed-paper.html

[4] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K. Cython: the best of both worlds. Comput Sci Eng 2011;13(2):31–9. https://doi.org/10.1109/MCSE.2010.118.

[5] Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, Pozo R, Remington K, Whaley RC. An updated set of basic linear algebra subprograms (BLAS). ACM Trans Math Softw 2002;28(2):135–51. https://doi.org/10.1145/567806.567807.

[6] Bordas S, Nguyen PV, Dunant C, Guidoum A, Nguyen-Dang H. An extended finite element library. Int J Numer Methods Eng 2007;71(6):703–32. https://doi.org/10.1002/nme.1966.

[7] Bradbury J., Frostig R., Hawkins P., Johnson M.J., Leary C., Maclaurin D., Wanderman-Milne S.. JAX: composable transformations of Python+NumPy programs. URL https://github.com/google/jax, Ver. 0.2.9; 2021.

[8] Cimrman R.. soops: run parametric studies runner and collect results. URL https://github.com/rc/soops,Ver. 2021.1; 2021b.

[9] Cimrman R. Performance measurements of Python tensor contraction packages in the finite element context. Zenodo; 2021. https://doi.org/10.5281/zenodo.4750560.

[10] Cimrman R, Lukeš V, Rohan E. Multiscale finite element calculations in Python using SfePy. Adv Comput Math 2019. https://doi.org/10.1007/s10444-019-09666-0.

[11] Dask Development Team. Dask: library for dynamic task scheduling. URL https://dask.org, Ver. 2021.02.0; 2021.

[12] Gray J, Kourtis S. Hyper-optimized tensor network contraction. Quantum 2021;5:410. https://doi.org/10.22331/q-2021-03-15-410.

[13] Gustafsson T, McBain G. scikit-fem: a Python package for finite element assembly. J Open Source Softw 2020;5(52):2369. https://doi.org/10.21105/joss.02369.

[14] Hale JS, Brunetti M, Bordas SPA, Maurini C. Simple and extensible plate and shell finite element models through automatic code generation tools. Comput Struct 2018;209:163–81. https://doi.org/10.1016/j.compstruc.2018.08.001.URL https://www.sciencedirect.com/science/article/pii/S0045794918306126

[15] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, van Kerkwijk MH, Brett M, Haldane A, del Río JF, Wiebe M, Peterson P, Gérard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE. Array programming with NumPy. Nature 2020;585(7825):357–62. https://doi.org/10.1038/s41586-020-2649-2.

[16] Lam SK, Pitrou A, Seibert S. Numba: a LLVM-based Python JIT compiler. Proceedings of the second workshop on the LLVM compiler infrastructure in HPC. LLVM '15. New York, NY, USA: Association for Computing Machinery; 2015, ISBN 978-1-4503-4005-2. p. 1–6. https://doi.org/10.1145/2833157.2833162.

[17] Liang L, Xu J, Deng L, Yan M, Hu X, Zhang Z, Li G, Xie Y. Fast search of the optimal contraction sequence in tensor networks. IEEE J Sel Top Signal Process 2021;15(3):574–86. https://doi.org/10.1109/JSTSP.2021.3051231.

[18] Automated solution of differential equations by the finite element method: the FEniCS book. In: Logg A, Mardal K-A, Wells G, editors. Lecture notes in computational science and engineering. Berlin Heidelberg: Springer-Verlag; 2012, ISBN 978-3-642-23098-1. https://doi.org/10.1007/978-3-642-23099-8.URL https://www.springer.com/gp/book/9783642230981

[19] Ma L., Ye J., Solomonik E.. AutoHOOT: automatic high-order optimization for tensors. 2020arXiv:2005.04540 [cs, math].

[20] McKinney W. Data structures for statistical computing in Python. Python in science conference, Austin, Texas. 2010. p. 56–61. https://doi.org/10.25080/Majora-92bf1922-00a.URL https://conference.scipy.org/proceedings/scipy2010/mckinney.html

[21] Moës N, Dolbow J, Belytschko T. A finite element method for crack growth without remeshing. Int J Numer Methods Eng 1999;46(1):131–50. https://doi.org/10.1002/(SICI)1097-0207(19990910)46:1<131::AID-NME726>3.0.CO;2-J.

[22] Novák Ms, Vackář Jr, Cimrman R. Evaluating Hellmann–Feynman forces within non-local pseudopotentials. Comput Phys Commun 2020;250:107034. https://doi.org/10.1016/j.cpc.2019.107034.URL http://www.sciencedirect.com/science/article/pii/S0010465519303698

[23] Pedregosa F., et al. memory_profiler: a Python module for monitoring memory consumption. URL https://github.com/pythonprofilers/memory_profiler, Ver. 0.58.0; 2021.

[24] Rahman T, Valdman J. Fast MATLAB assembly of FEM matrices in 2D and 3D: nodal elements. Appl Math Comput 2013;219(13):7151–8. https://doi.org/10.1016/j.amc.2011.08.043.URL http://www.sciencedirect.com/science/article/pii/S0096300311010836

[25] Ran S-J, Tirrito E, Peng C, Chen X, Tagliacozzo L, Su G, Lewenstein M. Tensor network contractions: methods and applications to quantum many-body systems. Lecture Notes in Physics, vol. 964. Cham: Springer International Publishing; 2020. https://doi.org/10.1007/978-3-030-34489-4.ISBN: 978-3-030-34488-7,978-3-030-34489-4

[26] Samaniego E, Anitescu C, Goswami S, Nguyen-Thanh VM, Guo H, Hamdia K, Zhuang X, Rabczuk T. An energy approach to the solution of partial differential equations in computational mechanics via machine learning: concepts, implementation and applications. Comput Methods Appl Mech Eng 2020;362:112790. https://doi.org/10.1016/j.cma.2019.112790.URL https://www.sciencedirect.com/science/article/pii/S0045782519306826

[27] Schindler F, Jermyn AS. Algorithms for tensor network contraction ordering. Mach Learn 2020;1(3):035001. https://doi.org/10.1088/2632-2153/ab94c5.
[28] Smith DGA, Gray J. opt_einsum - a Python package for optimizing contraction order for einsum-like expressions. J Open Source Softw 2018;3(26):753. https://doi.org/10.21105/joss.00753.
[29] Świrydowicz K, Chalmers N, Karakus A, Warburton T. Acceleration of tensor-product operations for high-order finite element methods: Int J High Perform Comput Appl 2019. https://doi.org/10.1177/1094342018816368.
[30] Turjanicová J, Rohan E, Lukeš V. Homogenization based two-scale modelling of ionic transport in fluid saturated deformable porous media. Comput Math Appl 2019;78(9):3211–35. https://doi.org/10.1016/j.camwa.2019.05.028.URL https://www.sciencedirect.com/science/article/pii/S0898122119302913
[31] Zienkiewicz OC, Taylor RL. The finite element method: the basis. Butterworth-Heinemann; 2000, ISBN 978-0-7506-5049-6.