International Journal for Numerical Methods in Engineering

WILEY

RESEARCH ARTICLE OPEN ACCESS

Matrix-Free Methods for Finite-Strain Elasticity: Automatic Code Generation With No Performance Overhead

Michał Wichrowski[1] | Mohsen Rezaee-Hajidehi[2] | Jože Korelc[3] | Martin Kronbichler[4] | Stanisław Stupkiewicz[2]

[1]Interdisciplinary Center for Scientific Computing, Heidelberg University, Heidelberg, Germany | [2]Institute of Fundamental Technological Research, Polish Academy of Sciences, Warsaw, Poland | [3]Faculty of Civil and Geodetic Engineering, University of Ljubljana, Ljubljana, Slovenia | [4]Applied Numerics, Faculty of Mathematics, Ruhr University Bochum, Bochum, Germany

Correspondence: Michał Wichrowski (mt.wichrowsk@uw.edu.pl)

Received: 25 May 2025 | **Revised:** 26 September 2025 | **Accepted:** 13 October 2025

Funding: This work was supported by the European High Performance Computing Joint Undertaking (Grant No. 101172493) and the European Commission (Grant No. 101008140).

Keywords: automatic differentiation | code generation | finite elements | finite-strain elasticity | high-performance computing | matrix-free

ABSTRACT

This study explores matrix-free tangent evaluations in finite-strain elasticity with the use of automatically generated code for the quadrature-point level calculations. The code generation is done via automatic differentiation (AD) with AceGen. We compare hand-written and AD-generated codes under two computing strategies: on-the-fly evaluation and caching intermediate results. The comparison reveals that the AD-generated code achieves superior performance in matrix-free computations.

1 | Introduction

Matrix-free methods rely on the elegance of well-optimized loops to evaluate the action of a linear operator on a vector without explicitly storing matrix entries [1–8]. While these methods significantly boost computational performance, including applications in solid mechanics [9–11], their adoption is hindered by the formidable challenge of deriving and implementing complex tangent operators. This study explores the applicability of automatically generated codes within matrix-free nonlinear solvers, focusing on finite-strain elasticity.

Typical finite-element computations involve operating on large sparse matrices and are, as a consequence, memory-bound. This implies that the throughput (time per unknown) of sparse matrix–vector products is limited by main memory access speed, leaving a vast majority of computational resources fallow. In [10, 12] it was estimated that for a modern

CPU, iterative sparse linear solvers saturate memory bandwidth at less than 2% of the processor's theoretical arithmetic computing power. As computing capabilities continue to outgrow memory bandwidth [13], this gap is further exacerbated. At the same time, higher-order elements, with denser global tangent matrix, can decrease throughput in classical matrix-based solvers. Consequently, linear elements are often preferred for large-scale simulations, despite their lower accuracy per degree of freedom [10, 14, 15] and susceptibility to locking [16].

Matrix-free operator evaluation alleviates the memory bottleneck by avoiding the explicit storage and manipulation of sparse matrices [4, 8, 17]. The most general matrix-free approach involves computing finite-element integrals in weak forms on the fly. Leveraging the tensor-product structure of the finite-element basis functions through sum factorization [18] allows to reduce the arithmetic load of computing the integrals for every operator evaluation. Additional optimizations can further reduce the computational effort associated with the one-dimensional interpolations in sum factorization, for example, by the even-odd decomposition proposed by [19] and analyzed in [6]. Due to the higher arithmetic intensity, hardware-specific optimizations such as the use of vectorization (SIMD instructions) have been developed. Some optimization strategies specific to triangles, tetrahedra, and prismatic elements have been also proposed [20, 21]. A review of high-performance solvers can be found in [22], which explores performance and space-time trade-offs for compute-intensive kernels of large-scale numerical solvers for PDEs. For solving the linear systems arising from elliptic partial differential equations, matrix-free methods are usually combined with multigrid solvers [23, 24], leading to overall linear complexity with respect to the number of degrees of freedom.

Employing the matrix-free method for solving problems of finite-deformation solid mechanics poses significant challenges. A primary challenge is the complexities involved in the implementation of the tangent operator, as the proposed quadrature-based approach needs to evaluate the constitutive terms in every evaluation step. A successful implementation of the matrix-free method for a (compressible) neo-Hookean hyperelastic model was presented by Davydov et al. [9]. They proposed different caching strategies to evaluate the resulting constitutive relations, using scalar quantities, second-order tensors or a fourth-order tensor, with the aim to optimize the performance of the matrix-free method. Nevertheless, their implementation relied on the explicit evaluation of the tangent operator, a requirement that limits its applicability to more complex models. More recent works have continued to explore matrix-free methods in solid mechanics. For instance, Schussnig et al. [11] derived and implemented closed-form expressions for the tangent of a neo-Hookean model with an isochoric-volumetric split, focusing on storage strategies to balance the compute load against memory access. Similarly, Brown et al. [10] discussed efficient matrix-free representations of Jacobians, demonstrating that automatic differentiation (AD) can accelerate the development of nonlinear material models for GPUs. Both studies reported significant speed-ups for higher-order polynomial elements. In a broader context, a transient fluid-structure interaction matrix-free solver involving an incompressible Mooney-Rivlin solid was presented in [25]; however, that work did not require the evaluation of the tangent operator.

The present work employs the general-purpose infrastructure provided by the `deal.II` finite-element library [26, 27], similar to the previous contributions [9, 11]. Here, the operations at quadrature points for both the residual and the associated tangent operator are specified in the application code. Conversely, the interpolation of values or gradients to quadrature points, summation for quadrature, the loop over mesh elements and the exchange of data between different processes in a parallel computation are using library code. This enables the use of code tuning and data access optimizations from previous contributions.

AD offers a powerful approach to generating efficient finite-element codes by reducing manual intervention in differentiation and coding [28–30]. This minimizes human error in deriving the residual vector and tangent operator, improving the reliability and efficiency of generated codes. Moreover, this automation can bring significant time savings in the code development phase. However, computational efficiency can be an issue, particularly when combined with matrix-free methods. For instance, in [10], an attempt to use AD through Enzyme [31] for finite-strain elasticity yielded unsatisfactory results, with performance over 30% slower than hand-written code due to the limitation of Enzyme to built-in types and incompatibility with vectorized operations.

In this study, we propose and evaluate the performance of a matrix-free implementation of AD-generated finite-element codes for neo-Hookean hyperelasticity models using the AceGen system (http://symech.fgg.uni-lj.si/), see also [32, 33]. AceGen employs a hybrid symbolic-numerical approach to automate the finite-element method (FEM). It leverages the symbolic and algebraic capabilities of the general computer algebra system Mathematica [34], combined with AD and code generation/optimization, to create finite-element user subroutines. Since AD provides the exact analytical derivative, the generated tangent operator is identical to a correct symbolic derivation, ensuring that the convergence behavior of

the Newton solver is unaffected. Furthermore, since AceGen generates pure C/C++ code, it can be templated to utilize vectorized data types, enabling the use of Single Instruction, Multiple Data (SIMD) operations, which are crucial for achieving high performance as demonstrated in [9].

AD tools can be broadly categorized by their implementation approach (operator overloading vs. source-to-source transformation) and by the mode of applying the chain rule (forward vs. backward/inverse). Pure operator overloading often suffers from low numerical efficiency. Consequently, most AD tools utilize some form of operator overloading to generate code in a simplified intermediate language, followed by a source-to-source transformation. AceGen uses a variant of the source-to-source implementation of AD in both backward and forward modes, specifically optimized for generating finite-element subroutines. A key feature is that the codes for a function and its derivatives are merged and optimized together. This means that when calculating higher derivatives or applying directional derivatives repeatedly on the same function, only one optimized subroutine is generated. All operations with tensors and matrices are performed and optimized component by component, completely avoiding loops and function calls. This is accompanied by several code optimization strategies so that uncontrollable growth of expressions is avoided. Finally, the generated code is self-sufficient, that is, no external subroutines or libraries are called by the generated code, which facilitates parallelization. These special features of AceGen's AD are ideal for an efficient automatic generation of subroutines for the matrix-free solution of linear systems. In addition, AceGen's AD allows modifications of the chain rule through an AD exception mechanism [35], which facilitates the differentiation-based description of mechanical models.

While this work is limited to hyperelastic materials, AceGen is actually suitable and successfully used (but not in the matrix-free framework) for much more complex constitutive models, for instance, finite-strain elastoplasticity [36] or phase-field method coupled with crystal plasticity [37]. In particular, it permits consistent linearization of the respective nested iterative-subiterative schemes, including a doubly nested one [38].

Our investigation focuses on a compressible neo-Hookean hyperelastic model, with efficient matrix-free implementations available as a baseline [9]. We compare our AD-based implementation with different caching-based implementations from that work and a conventional implementation using a sparse iterative solver [39]. Our investigation focuses on the trade-off between caching and computing, the overhead associated with AD, and the feasibility of storing partial results while maintaining a general solid mechanics solver. Our results show that there is no overhead caused by AD, and that the AceGen-generated code stands out as the best matrix-free implementation, at the same time being a general approach, not limited to a specific constitutive model. The key idea is that we employ AD with a seed-matrix technique [30] to evaluate quadrature-point directional derivatives (i.e., the action of the tangent on a given gradient) directly, thereby avoiding the explicit formation of the fourth-order tangent tensor. This particular setup of concentrating AD to the work at a single quadrature point enables code compactness, with outer loops over cells and quadrature points located outside the code generator. We also evaluate our AD-generated code against the hand-crafted implementation from the work by Schussnig et al. [11], where another variant of hyperelastic model (with isochoric-volumetric split of the energy) is considered, again demonstrating superior performance. Using a Hencky model, we assess AD on a more complex constitutive law to show how costly point-wise operations shift the recompute-cache trade-off.

The remainder of this paper is organized as follows. We detail the formulation of the nonlinear problem and the solution procedure using the Newton method in Section 2. In particular, we describe the matrix-free evaluation of the tangent operator and sum factorization in Section 2.4, and the matrix-free implementation using AD and caching strategies in Section 2.5. We assess and compare the performance of different matrix-free implementations in Section 3. Finally, we wrap up the paper by outlining the concluding remarks and potential future directions.

2 | The Nonlinear Problem

2.1 | Problem Formulation

In this section, we introduce basic concepts of finite-strain elasticity, which are standard but serve here as a background for the subsequent matrix-free implementation. We consider a hyperelastic body that in the reference configuration occupies the domain $\Omega \subset \mathbb{R}^d$ with the boundary partitioned into a Dirichlet part $\Gamma_{\mathrm{D}} \subset \partial\Omega$ and a Neumann part $\Gamma_{\mathrm{N}} \subset \partial\Omega$. We assume that the Dirichlet boundary Γ_{D} is fixed, while a conservative surface traction $\mathbf{T}^*$ is applied on the Neumann boundary Γ_{N}. The deformation of the body is described by the mapping $\boldsymbol{\varphi} : \Omega \to \mathbb{R}^d$ that links the reference configuration Ω to the current configuration ω, that is, $\boldsymbol{\varphi}(\Omega) = \omega$. The mapping $\boldsymbol{\varphi}$ is assumed to be a function having sufficient

regularity for the weak formulation and satisfying $J := \det \mathbf{F} > 0$ almost everywhere to preserve material orientation. The displacement field $\mathbf{u}$ is defined as the difference between the position in the deformed and reference configurations, that is, $\mathbf{u}(\mathbf{X}) = \boldsymbol{\varphi}(\mathbf{X}) - \mathbf{X}$, where $\mathbf{X}$ is the position vector in the reference configuration. The deformation gradient

$$\mathbf{F} := \operatorname{Grad} \boldsymbol{\varphi} = \mathbf{I} + \operatorname{Grad} \mathbf{u} \tag{1}$$

is the main kinematic quantity in the finite-deformation setting. We use Grad $(\cdot)$ to denote the gradient in the reference configuration.

The elastic response of the material is described by the strain energy density function Ψ, which is a differentiable function of the deformation gradient $\mathbf{F}$, or alternatively, of the right Cauchy-Green tensor $\mathbf{C} = \mathbf{F}^{\mathrm{T}} \cdot \mathbf{F}$. In this setting, we define the potential energy functional

$$\mathcal{E}(\mathbf{u}) := \int_{\Omega} \Psi(\mathbf{F}) \, \mathrm{d}V - \int_{\Gamma_{\mathrm{N}}} \mathbf{T}^{*} \cdot \mathbf{u} \, \mathrm{d}S. \tag{2}$$

We seek a displacement field $\mathbf{u} \in \mathbb{V}$ that renders the first variation of $\mathcal{E}$ equal to zero for all admissible variations $\delta\mathbf{u} \in \mathbb{V}$, where $\mathbb{V} = \{\mathbf{v} \in H^1(\Omega)^d \, : \, \mathbf{v}|_{\Gamma_{\mathrm{D}}} = \mathbf{0}\}$ is the space of admissible displacements. The corresponding stationarity condition yields the weak form, that is, the virtual work principle,

$$\begin{aligned} \mathcal{F}(\mathbf{u}, \delta\mathbf{u}) := \mathrm{D}_{\delta\mathbf{u}}\mathcal{E} &= \int_{\Omega} \frac{\partial \Psi}{\partial \operatorname{Grad} \mathbf{u}} \, : \operatorname{Grad} \delta\mathbf{u} \, \mathrm{d}V - \int_{\Gamma_{\mathrm{N}}} \mathbf{T}^{*} \cdot \delta\mathbf{u} \, \mathrm{d}S \\ &= \int_{\Omega} \mathbf{P} \, : \operatorname{Grad} \delta\mathbf{u} \, \mathrm{d}V - \int_{\Gamma_{\mathrm{N}}} \mathbf{T}^{*} \cdot \delta\mathbf{u} \, \mathrm{d}S = 0 \qquad \forall \, \delta\mathbf{u}, \end{aligned} \tag{3}$$

where $\mathrm{D}_{\delta\mathbf{u}}\mathcal{E}$ denotes the Gâteaux derivative in the direction $\delta\mathbf{u}$, and

$$\mathbf{P} := \frac{\partial \Psi}{\partial \mathbf{F}} = \frac{\partial \Psi}{\partial \operatorname{Grad} \mathbf{u}} \tag{4}$$

is the first Piola-Kirchhoff stress tensor.

To solve Equation (3) for the unknown displacement $\mathbf{u}$ we use Newton's method. Given a current approximation of the displacement $\overline{\mathbf{u}}$, we compute a Gateaux derivative of the functional with respect to the displacement in direction $\Delta\mathbf{u}$,

$$\mathcal{K}(\overline{\mathbf{u}}; \Delta\mathbf{u}, \delta\mathbf{u}) := \mathrm{D}_{\Delta\mathbf{u}} \mathcal{F}(\overline{\mathbf{u}}, \delta\mathbf{u}). \tag{5}$$

We compute the correction $\Delta\mathbf{u}$ by solving the linearized problem

$$\mathcal{F}(\overline{\mathbf{u}} + \Delta\mathbf{u}, \delta\mathbf{u}) \approx \mathcal{F}(\overline{\mathbf{u}}, \delta\mathbf{u}) + \mathcal{K}(\overline{\mathbf{u}}; \Delta\mathbf{u}, \delta\mathbf{u}) = 0 \qquad \forall \, \delta\mathbf{u}. \tag{6}$$

The operator $\mathcal{K}(\overline{\mathbf{u}}; \Delta\mathbf{u}, \delta\mathbf{u})$ is the tangent operator, bilinear with respect to $\Delta\mathbf{u}$ and $\delta\mathbf{u}$. For conservative loading, it is given by

$$\mathcal{K}(\overline{\mathbf{u}}; \Delta\mathbf{u}, \delta\mathbf{u}) = \int_{\Omega} \mathrm{D}_{\Delta\mathbf{u}}\mathbf{P} \, : \operatorname{Grad} \delta\mathbf{u} \, \mathrm{d}V = \int_{\Omega} \operatorname{Grad} \Delta\mathbf{u} \, : \mathbb{L} \, : \operatorname{Grad} \delta\mathbf{u} \, \mathrm{d}V, \tag{7}$$

where $\mathbb{L}$ is the fourth-order tangent stiffness tensor with the major symmetry ($L_{iAjB} = L_{jBiA}$),

$$\mathbb{L} := \frac{\partial \mathbf{P}}{\partial \mathbf{F}} = \frac{\partial^2 \Psi}{\partial \mathbf{F} \otimes \partial \mathbf{F}}. \tag{8}$$

Both the weak form (3) and the tangent operator (7) can be evaluated in the current configuration. Specifically, the volume integral in Equation (3) can be equivalently expressed as

$$\int_{\Omega} \mathbf{P} \, : \operatorname{Grad} \delta\mathbf{u} \, \mathrm{d}V = \int_{\Omega} \boldsymbol{\tau} \, : \operatorname{grad}^{\mathrm{s}} \delta\mathbf{u} \, \mathrm{d}V = \int_{\omega} \boldsymbol{\sigma} \, : \operatorname{grad}^{\mathrm{s}} \delta\mathbf{u} \, \mathrm{d}v, \tag{9}$$

where $\boldsymbol{\tau} = \mathbf{P} \cdot \mathbf{F}^{\mathrm{T}}$ is the Kirchhoff stress tensor, $\boldsymbol{\sigma} = \boldsymbol{\tau}/J$ is the Cauchy stress tensor, and $\mathrm{d}v = J\mathrm{d}V$. The gradient in the current configuration is denoted by Grad $(\cdot)$, and $\mathrm{grad}^{\mathrm{s}}\ (\cdot)$ denotes its symmetric part. Both $\boldsymbol{\tau}$ and $\boldsymbol{\sigma}$ are symmetric, hence only the symmetric part of the gradient of $\delta\mathbf{u}$ is involved.

Following [9], see also [16], the tangent operator in Equation (7) can be transformed to the current configuration as

$$\mathcal{K}(\overline{\mathbf{u}}; \Delta\mathbf{u}, \delta\mathbf{u}) = \int_{\omega} \mathrm{grad}^{\mathrm{s}}\Delta\mathbf{u} \,:\, \mathbb{c} \,:\, \mathrm{grad}^{\mathrm{s}}\delta\mathbf{u}\ \mathrm{d}v + \int_{\omega} \mathrm{grad}\delta\mathbf{u} \,:\, \left(\mathrm{grad}\Delta\mathbf{u} \cdot \boldsymbol{\sigma}\right)\ \mathrm{d}v, \tag{10}$$

where the first term is the material part of the tangent operator and the second term is the geometric part. Note that the respective formula (10) in [9] is expressed through integrals over the reference configuration Ω, hence $J\mathbb{c}$ and $\boldsymbol{\tau} = J\boldsymbol{\sigma}$ are used instead of $\mathbb{c}$ and $\boldsymbol{\sigma}$, respectively. The fourth-order spatial elasticity tensor $\mathbb{c}$ possesses both the minor and major symmetries ($c_{ijkl} = c_{jikl} = c_{klij}$) that we exploit in the implementation. It is the push-forward of the material tangent stiffness tensor $\mathbb{C}$,

$$J\mathbb{c} = \chi(\mathbb{C}), \qquad \mathbb{C} := 4\frac{\partial^2\Psi(\mathbf{C})}{\partial\mathbf{C} \otimes \partial\mathbf{C}}, \tag{11}$$

where Ψ is now considered as a function of the right Cauchy-Green tensor $\mathbf{C} := \mathbf{F}^{\mathrm{T}} \cdot \mathbf{F}$, and $\chi(\cdot)$ denotes the push-forward operation such that $\chi(\mathbb{C})_{ijkl} = F_{iA}F_{jB}F_{kC}F_{lD}C_{ABCD}$.

From the above formulation, it is evident that the crucial part of the evaluation involves expressions with derivatives of the strain energy function Ψ, particularly its second derivatives as shown in Equations (7, 8) and (10, 11).

2.2 | Numerical Solution of the Problem

To solve this problem numerically, we apply the FEM by introducing a mesh $\mathcal{T}_L$, that is, a collection (set) of elements, that subdivides the domain Ω into quadrilateral (2D) or hexahedral (3D) elements. We define a finite-element space $\mathbb{V}_L \subset \mathbb{V}$ of vector functions using the element $\mathbb{Q}_p$ of piecewise polynomials up to degree p in each direction, see for instance [40]. Introducing the finite-dimensional finite-element space reduces the problem to finding the solution of a nonlinear system of equations. This is achieved through the Newton method: at each iteration, we solve a linear system of equations involving operator $\mathcal{K}(\overline{\mathbf{u}}; \cdot, \cdot)$ to obtain a correction to the current solution.

We choose a basis $\mathbb{B} = \{\boldsymbol{\phi}_i \in \mathbb{V}_L \mid i = 1, \ldots, \dim \mathbb{V}_L\}$ using shape functions $\boldsymbol{\phi}_i$ in space $\mathbb{V}_L$. This choice allows us to represent every element $\mathbf{v}$ of space $\mathbb{V}_L$ as a real vector v, corresponding to the coefficients of the basis functions in the expansion of $\mathbf{v}$. At each Newton iteration, for a given vector $\overline{\mathrm{U}}$ representing $\overline{\mathbf{u}}$, the problem is to find $\Delta\mathrm{U}$ with the corresponding $\Delta\mathbf{u}$ such that

$$\mathcal{K}(\overline{\mathbf{u}};\ \Delta\mathbf{u}\ , \boldsymbol{\phi}_i) = -\mathcal{F}(\overline{\mathbf{u}}, \boldsymbol{\phi}_i) \qquad \forall\boldsymbol{\phi}_i \in \mathbb{B}. \tag{12}$$

2.3 | Linear System and Solver

The linear system involving the tangent operator is typically large, and its solution often represents the most time-consuming step of the procedure. For matrix-free approaches, the choice of the solver is restricted to iterative methods, as the matrix is not explicitly formed. To achieve optimal convergence and performance, an effective preconditioner is crucial. In our case, we employ the conjugate gradient (CG) method in conjunction with a geometric multigrid preconditioner, which is particularly well-suited for matrix-free implementations since it can be formulated entirely in terms of matrix–vector products, simple operations such as the matrix diagonal, and grid transfer operations.

Defining the multigrid iteration requires a hierarchy of problems as well as establishing level operators, smoothers, and transfer operators. We build the multigrid procedure on the assumption that the finest mesh $\mathcal{T}_L$ is a result of refining a coarse mesh so that nestedness is obtained:

$$\mathcal{T}_0 \sqsubset \mathcal{T}_1 \sqsubset \cdots \sqsubset \mathcal{T}_L. \tag{13}$$

The symbol "$\sqsubset$" indicates that every cell of mesh $\mathcal{T}_{\ell+1}$ is obtained from a cell of mesh $\mathcal{T}_\ell$ by refinement. On every level ℓ, we define a finite-element space $\mathbb{V}_\ell$ in the same manner as on the finest one. We assume the existence of transfer

operators between these finite-element spaces, allowing us to define prolongation, restriction, and level tangent operators consistently across the hierarchy interpolate the current iterate $\overline{\mathbf{u}}$ onto level ℓ to obtain $\overline{\mathbf{u}}_\ell$, providing a definition for tangent operator $\mathcal{K}(\overline{\mathbf{u}}_\ell;\cdot,\cdot)$ at each level, which will be used as a level operator within the multigrid procedure.

For the smoother, we utilize the inverse of the diagonal of the level tangents, using an iteration with Chebyshev polynomials, as described in [4, 23, 41]. As a result, the matrix–vector multiplication is the dominant operation in the smoother, which is in turn the dominant operation in the overall solver. The optimizations enabled by the tensor-product structure of $\mathbb{Q}_p$ elements and the low memory access of the matrix–free method are thus addressing the most expensive step in the solver.

2.4 | Matrix-Free Evaluation of the Tangent Operator

Applying the tangent operator in a matrix-free manner involves computing a result vector $\mathtt{W}$ from an input vector $\Delta\mathtt{U}$ that represents the finite-element field $\Delta\mathbf{u}$. Each component of the result vector represents the application of the tangent operator to the input vector, tested with a corresponding basis function, such that $\mathtt{W}_i = \mathcal{K}(\overline{\mathbf{u}};\ \Delta\mathbf{u}\ ,\boldsymbol{\phi}_i)$ for each basis function $\boldsymbol{\phi}_i \in \mathbb{B}$. To compute these coefficients, we decompose the evaluation into cell-wise contributions and apply numerical integration over each cell. For a cell $K \in \mathcal{T}$, the local contribution to the i-th component is computed as:

$$\mathtt{W}_{K,i} = \int_K \text{Grad}\ \Delta\mathbf{u}\ :\mathbb{L}:\text{Grad}\ \boldsymbol{\phi}_i\ \mathrm{d}V \approx \sum_q \text{Grad}\ \Delta\mathbf{u}\ :\mathbb{L}:\text{Grad}\ \boldsymbol{\phi}_i\ J_q\ w_q. \tag{14}$$

This numerical integration uses $(p+1)^d$ quadrature points, where J_q denotes the Jacobian determinant at point q, and w_q represents the corresponding quadrature weight. The complete procedure for matrix-free evaluation is presented in Algorithm 1, where the code for computing the crucial product $\mathbb{L}:\text{Grad}\ \Delta\mathbf{u}$ is what we aim to generate automatically.

A naive implementation of this procedure typically requires $\mathcal{O}((p+1)^{2d})$ operations for a degree p polynomial basis in d-dimensional space. This complexity comes from the fact that evaluating the function values/gradients at each quadrature point involves looping over all basis functions, leading to quadratic growth in computational cost relative to the number of degrees of freedom per element. However, matrix-free methods typically use sum factorization [1, 6, 22] to bring down the cost of evaluating the solution gradients Grad $\Delta\mathbf{u}$ at quadrature points. This is achieved by exploiting the tensor-product structure and performing multidimensional evaluation as a series of 1D operations, the complexity is thus reduced to $\mathcal{O}((p+1)^{d+1}) = \mathcal{O}(N_c \sqrt[d]{N_c})$, where $N_c = (p+1)^d$ is the number of degrees of freedom per element. When evaluating gradients, the gradients of the mapping from the reference cell to the physical space are required, as the derivative has to be scaled by the Jacobian matrix of the transformation. These Jacobian matrices are precomputed and stored within the matrix-free data structures.

ALGORITHM 1 | Matrix-free evaluation of the tangent operator.

Given : $\bar{\mathtt{U}}$ – vector representing $\bar{\mathbf{u}}$
$\Delta\mathtt{U}$ – input vector representing $\Delta\mathbf{u}$
Return: $\mathtt{W}_i = \mathcal{K}(\bar{\mathbf{u}};\ \Delta\mathbf{u}\ ,\boldsymbol{\phi}_i) \quad \forall\boldsymbol{\phi}_i \in \mathbb{B}$

```
W = 0 ;                                                  // zero destination vector
foreach element K ∈ T_L do
    gather local vector values on this element;
    evaluate at each quadrature point:
    Grad ū,  Grad Δu ;                                   // Sum factorization
    foreach quadrature point q on K ;                    // Quadrature loop
    do
        compute G = 𝕃 : Grad Δu ;                        // AceGen-generated
        queue G for contraction;
    evaluate queued contractions: G : Grad φ_i ;         // Sum factorization
    scatter results to W
```

Sum factorization leads to substantial computational cost savings, for example, by a factor of around 16 when using a moderate polynomial degree of $p = 3$ in 3D compared to a full-matrix operation on a single cell with cost $\mathcal{O}(N_c^2)$. This growth in complexity for a single cell is typically transferred to the global matrix–vector multiplication if a sparse matrix is used. However, this reduction can be offset by operations done by the quadrature loop, which, although of linear complexity, can still be expensive. The efficiency of the quadrature loop is critical, as a large fraction of the computational time is spent there, as shown in [9].

2.5 | Point-Wise Evaluation and Code Generation

The computational efficiency of the point-wise operations performed at quadrature points (Algorithm 1) is a critical factor determining the overall matrix-free performance. To handle these crucial calculations, two primary strategies can be employed, differing fundamentally in their approach to balancing computational work and memory usage: direct (on-the-fly) computation, where all terms are recalculated as needed, and partial assembly, which involves precomputing and caching intermediate results. Specifically, in the on-the-fly approach, all constitutive quantities are recomputed at every quadrature point for each application of the operator (i.e., in every CG iteration), whereas in the partial assembly approach, quantities like the spatial tangent tensor and stress are computed once per Newton step and cached for all CG iterations within that step. Direct computation evaluates all constitutive terms during each operator application; it minimizes memory usage but can be computationally intensive, especially for complex models. On the other hand, partial assembly reduces computations during operator application at the cost of increased memory usage. Both approaches are broadly applicable, but neither represents an optimal solution for all scenarios. The optimal balance depends on the model complexity and hardware characteristics, particularly memory bandwidth.

In [9, 11], several caching strategies for neo-Hookean elasticity were explored, it was found that caching the fourth-order tensor (exploiting symmetries) was often the most effective strategy for higher polynomial degrees. It was further noted that this strategy could generalize to other models using the spatial tangent tensor, although simpler caching could also be competitive. In [11], a similar implementation was tested on newer hardware, finding that model-specific caching approaches with only scalar quantities provided the best performance.

Let us first discuss the direct computation approach, specifically when using the formulation in the reference configuration. In this case, referring to Equation (7), the core term to be computed is the product of the tangent stiffness tensor $\mathbb{L}$ and the referential gradient of the displacement correction $\Delta\mathbf{u}$, that is, $\mathbb{L} : \text{Grad}\, \Delta\mathbf{u}$. One can compute $\mathbb{L} : \text{Grad}\, \Delta\mathbf{u}$ on the fly. This requires a minimal amount of data and is more computationally intense than using precomputed values. While this might be the natural choice for simple problems, it may not pay off for more complex ones. Since all the model-dependent data is recomputed every time, efficient implementations are crucial for the performance. We use AceGen [32] for automatic derivation and coding of the quadrature-point expressions.

2.5.1 | Evaluation on the Fly

In a naive implementation, one would compute $\mathbb{L}$ and double-contract it with Grad $\Delta\mathbf{u}$. In the context of AD, $\mathbb{L}$ could be obtained by differentiating the strain energy Ψ twice with respect to the deformation gradient $\mathbf{F}$, see Equation (8). However, thanks to capabilities of the AD technique implemented in AceGen, we can use an approach that avoids explicitly forming and contracting $\mathbb{L}$. Note that the tangent stiffness tensor $\mathbb{L}$ itself is not needed, only its product with Grad $\Delta\mathbf{u}$,

$$\mathbf{G} := \mathbb{L} : \text{Grad}\, \Delta\mathbf{u}, \tag{15}$$

which is to be contracted with the referential gradient of $\delta\mathbf{u}$, see Equation (7).

Matrix–vector multiplication can be efficiently implemented in AD by the concept of the "seed" [30]. Let $\mathbf{y}(\mathbf{x})$ be a set of functions of independent variables $\mathbf{x}$. Within the AD, the Jacobian $\mathbf{J} := \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is calculated as $\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial \mathbf{x}}$, where $\frac{\partial \mathbf{x}}{\partial \mathbf{x}} = \mathbf{I}$ is a seed matrix. By assuming that $\mathbf{x}$ additionally depends on a fictitious variable ξ with derivatives $\frac{\partial \mathbf{x}}{\partial \xi} = \mathbf{s}$, the chain rule leads to $\frac{\partial \mathbf{y}}{\partial \xi} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{x}}{\partial \xi} = \mathbf{J} \cdot \mathrm{s}$, where $\mathbf{s}$ is an arbitrary seed vector. Thus, the matrix–vector multiplication is performed in a very efficient way without having to explicitly form the Jacobian matrix and do the multiplication. The concept of the seed is fundamental to AD and as such can be implemented by modern AD tools.

```
(* Initilization and input *)
<<AceGen`;
dim=3;
SMSInitialize["neoHooke_"<>ToString[dim]<>"D_Tangent", "Language"→"C++"];
SMSModule["tangent", Real[gradU$$[dim,dim],gradDU$$[dim,dim],out$$[dim,dim],mu$$,lambda$$]];
gradU ⊢ Table[SMSReal[gradU$$[i, j]], {i,1,dim}, {j,1,dim}];
gradΔU ⊢ Table[SMSReal[gradDU$$[i, j]], {i,1,dim}, {j,1,dim}];
μ ⊢ SMSReal[mu$$];
λ ⊢ SMSReal[lambda$$];
ℍ ⊨ ArrayPad[gradU,{0,3-dim}];
```

```
(* Strain energy function *)
𝔽 ⊨ IdentityMatrix[3]+ℍ;
ℂ ⊨ Transpose[𝔽].𝔽;
J2 ⊨ Det[ℂ];
Ψ ⊨ μ/2(Tr[ℂ]-3-Log[J2])+λ/2(Log[J2]/2)^2;
```

```
(* Tangent *)
ℙ ⊨ SMSD[Ψ, ℍ, "IgnoreNumbers"→True];
ξ ⊢ SMSFictive[];
𝔾 ⊨ SMSD[ℙ, ξ, "Dependency"→{Flatten[gradU],ξ,Flatten[gradΔU]}];
```

```
(* Output *)
SMSExport[ 𝔾[[1;;dim,1;;dim]], out$$ ];
SMSWrite[ "OptimizingLoops"→ 2 ];
```

BOX 1 | AceGen code for generating a module that computes $\mathbf{G} = \mathbb{L} : \text{Grad}\, \Delta\mathbf{u}$ according to Equation (17).

To arrive at the desired formulation, recall that the first Piola-Kirchhoff stress tensor $\mathbf{P}$ depends on the deformation gradient $\mathbf{F}$, thus $\mathbf{P} = \mathbf{P}(\mathbf{F})$, and that $\mathbb{L}$ is defined as the derivative of $\mathbf{P}$ with respect to $\mathbf{F}$. Further, assume that the deformation gradient depends on a fictitious scalar variable ξ such that the derivative of $\mathbf{F}$ with respect to ξ is equal to Grad $\Delta\mathbf{u}$, so that

$$\mathbf{F} = \mathbf{F}(\xi), \qquad \frac{\partial \mathbf{F}}{\partial \xi} := \text{Grad}\, \Delta\mathbf{u}. \tag{16}$$

Now, using the above assumptions and applying the chain rule, compute the derivative of $\mathbf{P}$ with respect to ξ:

$$\left.\frac{\partial \mathbf{P}}{\partial \xi}\right|_{\frac{\partial \mathbf{F}}{\partial \xi}=\text{Grad}\, \Delta\mathbf{u}} = \frac{\partial \mathbf{P}}{\partial \mathbf{F}} : \left.\frac{\partial \mathbf{F}}{\partial \xi}\right|_{\frac{\partial \mathbf{F}}{\partial \xi}=\text{Grad}\, \Delta\mathbf{u}} = \mathbb{L} : \text{Grad}\, \Delta\mathbf{u}, \tag{17}$$

where $\frac{\partial \mathbf{F}}{\partial \xi}$ serves as a seed. This formulation yields the desired quantity $\mathbf{G}$, as defined in Equation (15).

The above formulation can be implemented in AceGen using the so-called AD exception [35]. Assuming that the energy function $\Psi = \Psi(\mathbf{H})$ is defined as a function of the displacement gradient $\mathbf{H} = \text{Grad}\, \overline{\mathbf{u}}$, the evaluation of $\mathbf{G}$ is done with the AceGen/Mathematica code snippet shown in Box 1.

The elastic strain energy function Ψ is defined in the second block as a function of the displacement gradient $\mathbf{H}$. In the third block, the first Piola-Kirchhoff stress $\mathbf{P}$ is defined as the derivative of Ψ with respect to $\mathbf{H}$, see Equation (4). Subsequently, $\mathbf{G}$ is defined according to Equation (17). In the AceGen code, `SMSD[·, ·]` is a call to the AD procedure that evaluates the derivative of the first argument with respect to the second argument, while the option `"Dependency"` introduces an AD exception that intervenes in the AD procedure by overriding the actual dependence existing in the algorithm (here,

ALGORITHM 2 | Matrix-free application of the tangent operator using partial assembly. The evaluation at each quadrature point is done using the cached fourth-order spatial elasticity tensor $\mathbb{c}$ and Cauchy stress $\boldsymbol{\sigma}$, compare Algorithm 3 in [9].

Given : $\Delta\mathtt{U}$ - input vector representing $\Delta\mathbf{u}$
$\mathbb{c}$ for each quadrature point
$\boldsymbol{\sigma}$ for each quadrature point
Return: $\mathtt{W}_i = \mathcal{K}(\bar{\mathbf{u}}; \Delta\mathbf{u}, \boldsymbol{\phi}_i) \quad \forall \boldsymbol{\phi}_i \in \mathbb{B}$

$\mathbf{w} = 0$; // zero destination vector
foreach *element* $K \in \mathcal{T}_L$ **do**
  gather local vector values on this element;
  evaluate at each quadrature point:
  grad $\Delta\mathbf{u}$; // Sum factorization
  foreach *quadrature point q on K ;* // Quadrature loop
  do
    compute
      $\mathbf{g} = \mathbb{c} : \text{grad}^{\text{s}}\Delta\mathbf{u} + \boldsymbol{\sigma} \cdot \text{grad}^{\text{s}}\Delta\mathbf{u}$; // AceGen-generated
    queue $\mathbf{g}$ for contraction;
  evaluate queued contractions:$\mathbf{g} : \text{grad}\boldsymbol{\phi}_i$; // Sum factorization
  scatter results to $\mathtt{W}$

no dependence, as $\mathbf{P}$ does not depend on ξ) by the one specified by this option. The forward mode of AD is employed (selected automatically by AceGen), which performs better than the backward mode for the problem at hand.

The AD-exception-based seed technique is the key feature of AceGen exploited in this work; otherwise the formulation and numerical workflow are standard. The high computational efficiency of the AceGen-generated code is additionally achieved thanks to the general features of AceGen that are listed in the introduction. The generated source code was subsequently postprocessed to expose compile-time opportunities (e.g., inlining and vectorized types) enabling further compiler optimizations.[1]

2.5.2 | Partial Assembly

An alternative strategy is to precompute and store $\mathbb{L}$ at quadrature points. This kind of partial assembly might be beneficial as it does not require any problem-dependent evaluation during the application of the tangent. However, this significantly increases the amount of data stored and, as a consequence, the algorithm could become memory-bound.

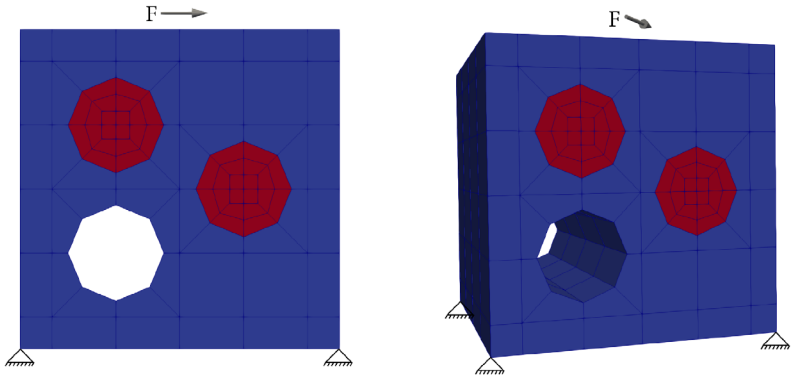
Significant reductions in storage can be achieved by exploiting the symmetries of the fourth-order tensor that are available when evaluating in the current configuration. Following Equation (10), the application of the tangent operator to $\Delta\mathtt{U}$ can be obtained using stored quadrature-point data, the fourth-order spatial elasticity tensors $\mathbb{c}$ and the second-order stress tensors $\boldsymbol{\sigma}$. We note that, due to the symmetries, $\mathbb{c}$ can be stored using only 21 unique real numbers and $\boldsymbol{\sigma}$ requires 6 real numbers in 3D.

The procedure, illustrated in Algorithm 2, is analogous to Algorithm 3 presented in [9]. However, while [9] utilized closed-form expressions derived manually for the quadrature-point computations, this work employs automatically generated code via AceGen for the same step (marked with a comment in the algorithm). The performance of this generated code will be compared against the hand-written implementation from [9].

3 | Performance of Matrix-Free Implementations

The ideas discussed above have been implemented in the C++ programming language by extending the code used by Davydov et al. [9], which builds on the `deal.II` finite element library [27].

We consider a compressible neo-Hookean model and its variant based on the isochoric-volumetric splitting. For the first model we can directly compare the state-of-the-art hand-written code [9] with the automatically generated code using



(a) 2D coarse mesh (b) 3D coarse mesh

FIGURE 1 | Discretization of the heterogeneous structure at the coarsest mesh level and the prescribed boundary conditions. Both the figure and mesh are taken from the paper by Davydov et al. [9]. (a) 2D coarse mesh, (b) 3D coarse mesh.

AceGen. We focus on the comparative performance on the node level and omit extensive scalability tests on many nodes, where our algorithm follows the state-of-the-art, see the results in [23, 42]. We test various caching strategies that can be employed to find a balance between computational cost and memory usage toward the goal of a minimal time to solution, see Section 2.5, see also Section 4 in [9].

3.1 | Setup for Performance Evaluation

All the codes were compiled with `GCC` 11.4.0 using the following optimization options:

```
-O3 -ffast-math -funroll-loops -ffp-contract=fast -march=native
```

We conduct a series of experiments on a dual-socket AMD EPYC 7282 machine. We measure the arithmetic floating point processing rate using LIKWID-bench [43] to describe the efficiency of the implementation. The machine's peak performance, obtained via the `peakflops_avx` test, is 50 GFLOP/s per core. With a total of 32 cores, a peak performance of 747 GFLOP/s has been measured. The machine is equipped with 256 GB of DDR4 memory, with a memory bandwidth of 320 GB/s. The experiments in parallel are conducted using 32 MPI ranks, which is the maximum number that can be used without oversubscribing the machine.

Figure 1 illustrates the geometry and the discretization of the heterogeneous structure from the test case used in [9] at the coarsest mesh level. The 2D structure consists of a square matrix material (depicted in blue) with a hole and two circular inclusions (depicted in red). The inclusions are 100 times stiffer than the surrounding material. The 3D geometry is an extrusion of the 2D geometry, resulting in a cube with edge length of 1000 mm. We use five elements in the extrusion direction for the coarse 3D mesh. The matrix material has a Poisson ratio of 0.3 and a shear modulus of $\mu = 0.4225 \times 10^6$ N/mm^2.

The bottom surface is fixed, while a distributed load is applied incrementally (in five loading steps) at the top. The loading is along the $(1, 0)$ direction and possesses an intensity of 12.5×10^3 N/mm^2 for the 2D problem. For the 3D problem, the loading is along the $(1, 1, 0)$ direction and possesses an intensity of $12.5\sqrt{2} \times 10^3$ N/mm^2. The Newton solver uses a displacement tolerance of 10^{-5} and a residual force tolerance of 10^{-8}, while the linear solver is set to a relative threshold of the residual of 10^{-6}.

3.2 | Model Problem: Neo-Hookean Hyperelasticity

The neo-Hookean model is a widely adopted hyperelasticity model in the study of rubber-like materials due to its simplicity and good predictive capabilities. In its compressible form, the neo-Hookean elastic strain energy is expressed as:

$$\Psi = \frac{\mu}{2}(\mathrm{tr}\mathbf{C} - \mathrm{tr}\mathbf{I} - 2\log J) + \frac{\lambda}{2}\log^2 J. \tag{18}$$

where μ is the shear modulus and λ is the Lamé constant.[2] We refer to [44–47] for general discussions on the formulation of the elastic strain energy for compressible and incompressible hyperelastic materials. The model presented in Equation (18) will be used as the main element for the evaluation of our matrix-free implementation. This hyperelasticity model was also examined in the studies by [9, 11], where the tangent stiffness tensor was derived manually. The simple form of this energy expression enabled several cache optimizations in these works. These results provide a baseline for comparison against the automatically generated code approach. Throughout this text, the model presented in Equation (18) will be referred to as the *compressible neo-Hookean model*.

For the implementation, we generate code using AceGen. We note that the evaluation using partial assembly could be further optimized as nearly half of the values of the stored tensors are zero. While this optimization is easy to implement with the presented tools, we do not consider it in this work. The reason is that the evaluation presented here using the partial assembly does not depend on the specific material model, and we want to keep the implementation and the results as general as possible.

3.2.1 | Efficiency of Evaluation at Quadrature Points

As the first test, we run the program in serial and measure the execution time of a matrix–vector multiplication with the matrix-free strategy. This setting ensures that arithmetic costs are most clearly identified, as synchronization overhead is eliminated and memory access costs through a shared interface are minimized. We use $\mathbb{Q}_2$ elements on a relatively coarse mesh (one refinement level) in 3D. Each of the four cores of the CPU had access to 16 MiB of L3 cache (64 MiB in total). This capacity is sufficient to accommodate most of the data required by the various caching strategies, ensuring that cache size did not pose a limiting factor in the performance comparisons.

Table 1 presents timings, data storage requirements for matrix-free tangent application, and floating point operations (measured via LIKWID [43]). It compares our automatically generated (AD) strategies with hand-written ones from [9]. The AD strategies include `ADstore` (storing the fourth-order tensor, see Algorithm 2) and `ADrecompute` (on-the-fly computation, see Algorithm 1). Among the hand-written strategies from [9]: `tensor4` (Algorithm 3 in [9]) is analogous to `ADstore` by also storing the fourth-order tensor. While our `ADrecompute` approach has no exact hand-written counterpart, `scalar_ref` (see the Appendix in [9]) is the most comparable, as it also involves minimal data storage by caching only $\log(J)$ and performing evaluation in the referential configuration. For completeness, other hand-written strategies considered are `scalar` (Algorithm 1 in [9]), which uses the same cached data as `scalar_ref` but evaluates in the deformed configuration, and the model-specific `tensor2` (Algorithm 2 in [9]), caching a second-order tensor.

The first three rows in Table 1 present the results for the automatically generated code. In the first row, we show the results obtained using a naive approach to AD (`naive_AD` formulation), where the tangent stiffness tensor is explicitly formed through standard differentiation and then contracted with the gradient, rather than using the efficient AD exception technique described in Section 2.5.1. The second row (`ADrecompute`) implements the computation of the product

TABLE 1 | Performance metrics for different caching strategies in the quadrature loop for compressible neo-Hookean model. Timings of vmult, floating point operations (FLOPs) per point, processing rate in GFLOP/s, and total cache size for Q2 elements in 3D on a mesh with one refinement level (75,072 degrees of freedom).

Formulation	Timing [ms]	FLOP/point	[GFLOP/s]	Cache [Mb]
`naive_AD`	5.56	6913	33.9	2
`ADrecompute`	3.43	3287	25.3	2
`ADstore`	3.33	1566	18.5	18
`scalar_ref` [9]	4.71	3148	14.8	4
`scalar` [9]	4.11	2138	13.2	5
`tensor4` [9]	4.44	1588	13.5	29
`tensor2` [9]	2.41	1146	12.4	9

$\mathbb{L}$: Grad $\Delta\mathbf{u}$ directly without explicitly forming $\mathbb{L}$, resulting in significantly improved performance. The third row shows results for storing the fourth-order tensor (`ADstore`). We observe that even for such a small problem, strategy `ADstore` performs only slightly better than the one without caching (`ADrecompute`), despite its lower number of operations per quadrature point. We expect that this effect can be attributed to the significantly larger cache size.

The next four rows in Table 1 detail the performance of the hand-written strategies from [9]. The `scalar_ref` strategy, despite having fewer operations per quadrature point than the AD-based `ADrecompute` approach, is slower. The `scalar` strategy also shows lower performance than the AD-based strategies. For both `scalar_ref` and `scalar`, this difference can be attributed to a lower processing rate in the hand-written codes, likely due to compiler-generated overhead and more computationally expensive operations. The processing rate of the AD-generated `ADrecompute` strategy is nearly twice that of the hand-written `scalar` strategy.

The hand-written `tensor4` strategy is clearly slower and requires significantly more data than its AD counterpart (`ADstore`), as not all tensor symmetries are exploited in the manual implementation. The automatically generated version also achieves an over 50% higher processing rate. (For reference, the sum factorization part of the matrix–vector product was performed at around 29.8 GFLOPS/s, although for very high-order elements, the peak performance was around 44 GFLOPS/s.)

Finally, `tensor2`, the fastest hand-written approach, has the lowest number of operations per quadrature point and a cache size only twice than that of `scalar_ref`. This approach is not generalizable as it relies on the specific form of the compressible neo-Hookean model, Equation (18), leading to a particularly simple spatial tangent tensor, see Equations (15) and (16) in [9].

3.2.2 | Parallel Performance

For a more realistic balance between compute capability and memory bandwidth, we here test our program in parallel by running the code with 32 MPI processes across a range of polynomial degrees and refinement levels. To ensure a balanced comparison, the number of degrees of freedom is maintained within the same order of magnitude, as detailed in Tables 2a and 2b. For instance, in 3D, when using polynomial degree $p = 1$, we apply four refinement levels, while for $p = 4$, we apply only one refinement level to maintain a similar computational scale.

Below we compare the timing of a single matrix–vector multiplication for various evaluation methods, including the "classical" way, that is, involving a sparse-matrix operation from the Epetra package of the Trilinos project [39]. As a normalized measure of performance, we record the processing rate in DoFs/second. The obtained results are depicted in Figure 2. We have tested all the formulations listed in Table 1, however, here we skip the results for `scalar_ref` as its performance is comparable to, but slightly worse than, that of `scalar`. The results obtained with automatically generated code are depicted with solid lines, while the ones obtained with hand-written code are shown with dotted lines. We observe that for $p > 1$ all matrix-free operators outperform the sparse-matrix one, with the gap growing with the degree p. This can be attributed to the more favorable ratio between the number of degrees of freedom and the number of quadrature points per cell at higher p. This can be accounted to exploiting the tensor-product structure of the shape functions, as discussed in Section 2.4. For degree $p = 4$ in 3D, matrix-free evaluations are up to 40 times faster than the sparse-matrix approach. When using linear shape functions the sparse-matrix approach is still slower than the matrix-free one by a factor of 4 in 3D. These results highlight the trade-off between memory access and computational work, which is a key factor in determining the best-performing strategy.

In 2D the hand-written approach with caching a second-order tensor (`tensor2`) is the best, closely followed by automatically generated code that caches the fourth-order tensor (`ADstore`). The superior performance of caching approaches is expected, as in 2D the data size is smaller than in 3D. In contrast, in 3D, the operator without caching performs comparably to the hand-written implementation that caches a second-order tensor. We recall that the latter implementation is limited to the considered model.

To explain the increasing efficiency for higher polynomial degrees let us consider a Cartesian mesh with n_c cells in each direction. For a finite element of degree p, we have $n_{\text{dof}} = (pn_c + 1)^d$ degrees of freedom and $\left(n_c(p+1)\right)^d$ quadrature points. Then the ratio between the total number of quadrature points to degrees of freedom is

TABLE 2 | Parameters for the benchmark: p is the polynomial degree, q is the number of quadrature points in 1D, N_{gref} is the number of global mesh refinements, N_{el} is the number of elements and N_{DoF} is the number of DoFs.

(a) 2D

p	q	N_{gref}	N_{el}	N_{DoF}
1	2	7	1,441,792	2,887,680
2	3	6	360,448	2,887,680
3	4	5	90,112	1,625,088
4	5	5	90,112	2,887,680
5	6	5	90,112	4,510,720
6	7	4	22,528	1,625,088
7	8	4	22,528	2,211,328
8	9	4	22,528	2,887,680

(b) 3D

p	q	N_{gref}	N_{el}	N_{DoF}
1	2	4	1,441,792	4,442,880
2	3	3	180,224	4,442,880
3	4	2	22,528	1,891,008
4	5	2	22,528	4,442,880

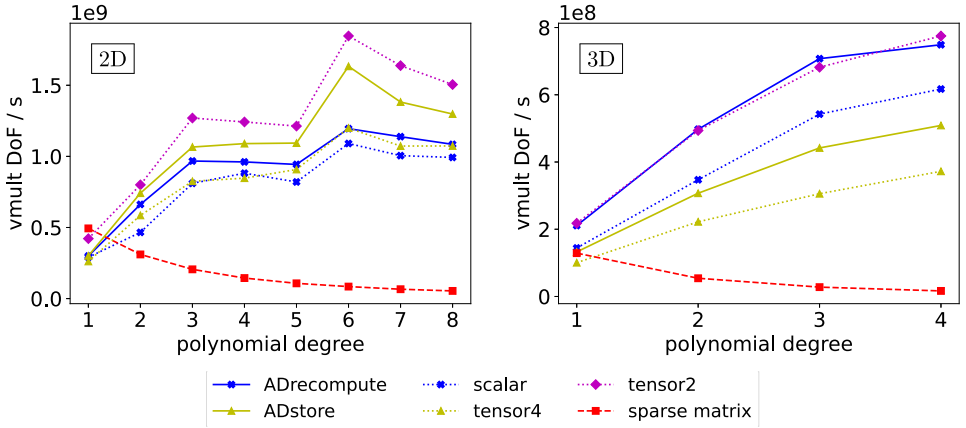


FIGURE 2 | Measured throughput of matrix–vector operator evaluation for the compressible neo-Hookean model. The processing rate is expressed in DoFs/second. The data is shown for 2D (left) and 3D (right). The results obtained with automatically generated code are depicted with solid lines, while the ones obtained with hand-written code [9] are shown with dotted lines. The sparse-matrix vmult is shown by red dashed line.

$$\left(\frac{(p+1)n_c}{pn_c+1}\right)^d \approx \left(\frac{p+1}{p}\right)^d.$$

This ratio for $p = 1$ is 4 in 2D and 8 in 3D, and approaches 1 for higher degrees p, see also [4].

In Figure 3 we plot the storage size required for the application of the matrix–vector operator. We express the size in the number of floating point numbers per degree of freedom. Especially in 3D, it is visible that memory usage impacts computing time as the storage of fourth-order tensors is the slowest strategy, even though it involves the lowest number of computations. The `ADstore` caching strategy requires storing 27 numbers per quadrature point, in addition to the standard quadrature-point data needed for any matrix-free operator: Jacobian matrices of the transformation (9 numbers) leading to total storage of 36 numbers. Note that there are also 3 degrees of freedom per node, meaning that the minimal storage per degree of freedom is $36/3 = 12$, while the observed ratio for `ADstore` with $p = 4$ in 3D is 20.3. As a reference,

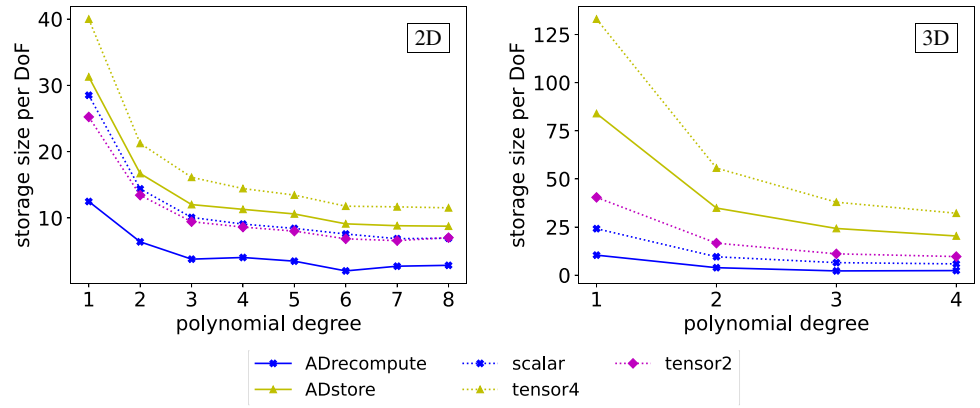


FIGURE 3 | Memory requirements per degree of freedom for matrix–vector operator application for the compressible neo-Hookean model. The storage size is expressed in the number of floating point numbers per DoF. The data is shown for 2D (left) and 3D (right). The results obtained with automatically generated code are depicted with solid lines, while the ones obtained with hand-written code [9] are shown with dotted lines.

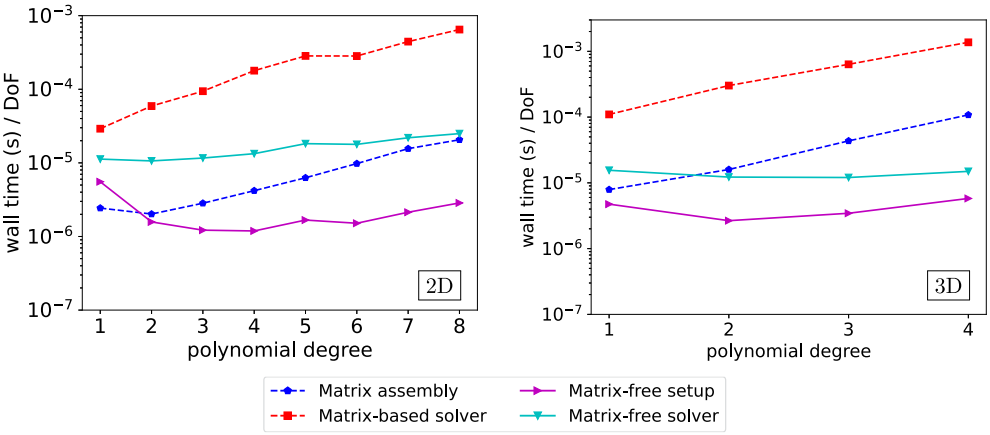


FIGURE 4 | Comparison of time to solution for matrix-free and sparse-matrix approaches across different polynomial degrees in 2D and 3D for the compressible neo-Hookean model. Computations using the sparse-matrix approach are shown with dashed lines, while the matrix-free approach is shown with solid lines.

at degree $p = 4$, the sparse-matrix approach requires 51 times as much memory, and over 750 times more memory than the matrix-free operator without intermediate result storage.

Finally, we also plot the time to solution for matrix-free versus sparse-matrix approaches. The results, depicted in Figure 4, clearly demonstrate the superior performance of matrix-free methods. The matrix-free approach consistently outperforms the sparse-matrix-based one, with the gap widening as the polynomial degree increases. For polynomial degree $p = 4$ in 3D, the matrix-free solver is 80 times faster than the sparse-matrix approach.

We observe that the time to solution does not follow the same trend as the processing rate for the matrix-free operator. This is due to the dependence of the preconditioner on the polynomial degree, which leads to an increase in the number of iterations for the solver for higher degrees. This issue is associated with the smoother and can be resolved by using a more sophisticated smoother [48].

3.3 | Neo-Hookean Model With Isochoric-Volumetric Split

Another popular variant of the neo-Hookean hyperelasticity model is the one that splits the elastic strain energy into isochoric and volumetric parts. Among the various formulations, the following form of the elastic strain energy is adopted here [47]

$$\Psi = \frac{\mu}{2}(\mathrm{tr}\overline{\mathbf{C}} - \mathrm{tr}\mathbf{I}) + \frac{\kappa}{2}\left(\frac{1}{2}(J^2 - 1) - \log J\right), \tag{19}$$

where $\overline{\mathbf{C}} = J^{-\frac{2}{3}}\mathbf{C}$ denotes the isochoric part of the right Cauchy-Green tensor $\mathbf{C}$ and $\kappa = \lambda + \frac{2}{3}\mu$ is the bulk modulus (in 2D, $\kappa = \lambda + \mu$). This model is henceforth referred to as *split neo-Hookean*.

Our rationale for selecting this model stems from the additional complexities it introduces in the evaluation of the residual vector and tangent matrix. This model was also considered in recent work [11] on matrix-free elasticity solvers, where an explicit derivation of the tangent operator was provided.

3.3.1 | Performance of Matrix-Free Implementation

With these preliminary observations in mind, our goal is to investigate how the inherent complexities of the model translate to the matrix-free implementation. We first measure floating point operations (FLOPs) per quadrature point and their execution rate as for the previous model. Table 3 summarizes these results.

Following our previous approach, we evaluate the solver performance across various grid sizes and polynomial degrees by measuring the times of matrix–vector operations. The corresponding results are shown in Figure 5. Since the evaluation timing and memory usage for the `ADstore` caching strategy are model-independent, we use these as reference values. The results confirm our earlier findings: caching with `ADstore` is advantageous in 2D, while on-the-fly computation proves more efficient in 3D. This of course will change for more complex models, where storing the fourth-order tensor is advantageous, as it decouples the tangent evaluation from the specific model.

For this model, a direct performance comparison between our automatically generated code and hand-written implementations is not feasible. Instead, we use our implementation of the compressible neo-Hookean model as a reference and rescale the throughput reported in [11] to account for differences in hardware and test problems. Specifically, we compute the ratio between our measured throughput and those reported in [11] for both scalar caching and fourth-order tensor

TABLE 3 | Performance metrics for different caching strategies in the quadrature loop for the split neo-Hookean model. Timings of vmult, FLOPs per point, processing rate in GFLOP/s, and total cache size for Q2 elements in 3D on a mesh with one refinement level (75,072 degrees of freedom).

Formulation	Timing [ms]	FLOPs per point	[GFLOP/s]	Cache [Mb]
`ADrecompute`	3.91	3634	27.0	2
`ADstore`	3.27	1566	19.5	18

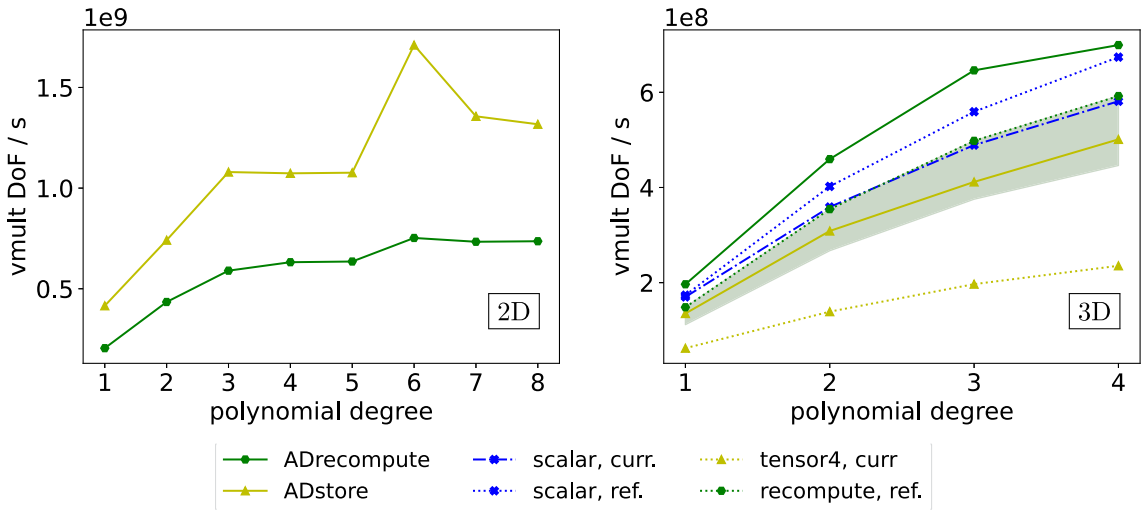


FIGURE 5 | Measured throughput during application of matrix–vector operator for the split neo-Hookean model. The processing rate is expressed in DoFs/second. The data is shown for 2D (left) and 3D (right). The results obtained with automatically generated code are depicted with solid lines, while the estimates for the ones obtained with hand-written code [11] are shown with dotted lines. To show the uncertainty, we indicate possible variations of *recompute all* throughput with the green area.

caching scenarios. To ensure a conservative estimate in each case, we take the maximum ratio across all polynomial degrees p considered. For the *recompute all* strategy reported in their work, we adopt the conservative ratio resulting from scalar caching. Note that this approach likely results in an overestimation of the implementation performance for [11]. This results in a conservative comparison, as a direct comparison on identical hardware, while desirable for definitive confirmation, is not possible. To reflect the uncertainty and the range of performance variation due to different polynomial degrees, we depict the resulting spread in Figure 5 as a green area, indicating the range of possible performance ratios. Nevertheless, our implementation of `ADrecompute` still outperforms even these generous estimates for the hand-written code.

3.4 | Hencky-Type Hyperelasticity

To provide further insight into the performance of the on-the-fly computation strategy, we conduct an additional test on a more complex hyperelastic model of Hencky type. The strain energy function for this model is given by

$$\Psi(\mathbf{H}) = \frac{1}{2}\lambda(\mathrm{tr}\mathbf{H})^2 + \mu\ \mathrm{tr}\,(\mathbf{H}\cdot\mathbf{H})\,, \tag{20}$$

where $\mathbf{H} = \frac{1}{2}\log\mathbf{C}$ is the Hencky (logarithmic) strain tensor, and λ and μ are the Lamé constants. A key challenge in implementing this model is the evaluation of the matrix logarithm, $\log\mathbf{C}$, and its first and second derivatives, which are required for the residual and tangent operator.

Since direct computation of the matrix logarithm is computationally intensive, here we adopt the approach proposed in [49], which utilizes Padé approximants. A Padé approximant of order $[m/m]$ approximates a function by a ratio of two polynomials of degree m. For a matrix argument $\mathbf{X}$, this translates to $\log(\mathbf{X}) \approx P_m^*(\mathbf{X})\cdot Q_m^*(\mathbf{X})^{-1}$, where P_m^* and Q_m^* are matrix polynomials corresponding to the approximation of $\log(\mathbf{X})$ in the vicinity of $\mathbf{X} = \mathbf{I}$. For instance, for $m = 2$, we have $\log(\mathbf{X}) \approx 3(\mathbf{X}^2 - \mathbf{I})\cdot(\mathbf{X}^2 + 4\mathbf{X} + \mathbf{I})^{-1}$; for other approximation orders see Table 1 in [49].

In the present context, utilization of a Padé approximant for the Hencky-type hyperelasticity model (20) involves evaluating the matrix polynomials and then inverting a small matrix at each quadrature point. Since the strain energy function is then an explicit function of the deformation gradient, it can be differentiated directly, and this can be done efficiently using AD.

The order m of the Padé approximant controls the trade-off between accuracy and computational cost. A higher value of m yields a more accurate approximation of the logarithm but increases the computational workload due to the higher order of the Padé approximant involved [49]. We test various orders m to analyze this trade-off. The results, shown in Figure 6, demonstrate that increasing the Padé order from $m = 1$ to $m = 3$ leads to a noticeable decrease in throughput. For $m = 1$, the performance is comparable to that of `ADstore`. We recall that whether `ADstore` or `ADrecompute` is faster is strongly hardware-dependent: on CPUs with higher arithmetic throughput `ADrecompute` can exploit the extra compute capacity and become relatively faster.

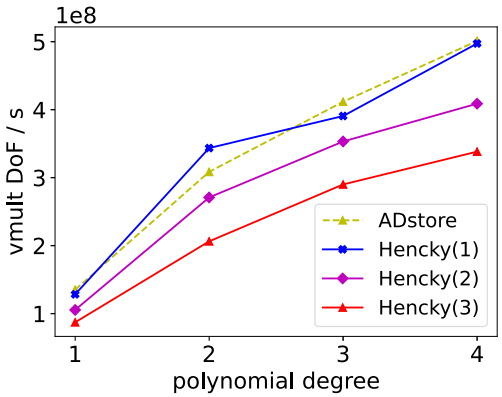


FIGURE 6 | Measured throughput during application of matrix–vector operator for the Hencky-type model. The processing rate is expressed in DoFs/second for the 3D problem. Results for the Hencky model using the `ADrecompute` strategy (for various Padé orders m) are shown with solid lines; the `ADstore` (model-independent) strategy is shown with a dashed line.

4 | Conclusion and Outlooks

This work demonstrates that automating code generation for matrix-free methods in finite-strain elasticity, through tools like AceGen, significantly enhances development efficiency and yields computationally superior code. By leveraging AD combined with symbolic and stochastic simplifications, the generated code not only surpasses CPU processing speeds of traditional hand-written codes but is also inherently less prone to human error, leading to more robust, efficient, and maintainable implementations.

Direct comparisons for the neo-Hookean model reveal that the automatically generated code outperforms its hand-written counterpart. Our automated approach also demonstrates excellent performance for the more complex neo-Hookean model with an isochoric-volumetric split, highlighting its versatility.

Our investigation into caching strategies suggests that for the particular test cases considered, on-the-fly computation with minimal data storage is more efficient in 3D, while storing intermediate results is beneficial in 2D. This observation reflects a specific balance between memory access and computational work, and it is not clear whether this trend would hold universally for multi-physics problems or space-time formulations. For more complex constitutive models, computing everything on the fly may not be optimal. In such scenarios, caching the fourth-order tensor emerges as a robust strategy, offering a good compute-storage balance by decoupling tangent evaluation from the specific material model and still delivering substantially better performance than sparse-matrix methods. Looking ahead, memory bandwidth may become a bottleneck on future hardware, a challenge potentially mitigated by reorganizing computational operations [48]. While the present study focused on a standard hyperelastic model to establish a performance baseline, future work will extend validation to more complex constitutive laws. Such investigations will be crucial to determine how the trade-off between memory access and re-computation, which is problem- and hardware-dependent, affects performance for different models and architectures, making such models accessible for efficient high-performance computing.

Further research avenues include addressing current solver limitations. The reliance of the geometric multigrid preconditioner on a multilevel hierarchy of nested meshes can be restrictive. While the matrix-free framework and sum factorization can be extended to non-conformal and unstructured meshes with little performance penalty on operator evaluation, constructing an efficient preconditioner without a mesh hierarchy is challenging. Besides standard techniques such as p-multigrid combined with algebraic multigrid at $p = 1$ [10, 11], unfitted approaches such as cutFEM [50] and Shifted Boundary Method [51], which have recently demonst rated to be compatible with matrix-free techniques [52–55], offer a promising alternative for complex geometries. Moreover, specialized techniques are crucial for applying multigrid to nearly incompressible solids. This challenge can be tackled by developing more advanced smoothers or by employing mixed formulations with robust block solvers [56]. These areas remain open for investigation with automatically generated matrix-free operators.

Acknowledgments

The authors declare the use of language models (ChatGPT, Gemini, and Claude) to improve the clarity and readability of the manuscript. All scientific content and technical claims are solely the responsibility of the authors.

M.R.H. and S.S. acknowledge support from the EU through the EffectFact project (No. 101008140) funded within the H2020-MSCA-RISE-2020 programme, and wish to thank Dr. Tomaž Šuštar for useful discussions and kind hospitality while visiting C3M, Ljubljana, Slovenia. M.K. acknowledges support by the EU through the EuroHPC joint undertaking Centre of Excellence dealii-X (No. 101172493) as well as the German Federal Ministry of Research, Technology and Space (BMFTR) through the project "PDExa: Optimized software methods for solving partial differential equations on exascale supercomputers", grant agreement no. 16ME0637K and the European Union – NextGenerationEU. Open Access funding enabled and organized by Projekt DEAL.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Endnotes

[1] The Mathematica notebook and the post-processing script are available in the project repository: https://github.com/mwichro/solid-matrix-free.

[2] Note that the factor of $\frac{1}{2}$ is missing in the respective Equation (11) in [9], which effectively scales λ. For consistency, the comparisons with [9] reported in this section are performed using the form of strain energy used in [9]. Note also that the code in Box 1 corresponds to the correct form of the energy in Equation (18).

References

1. S. A. Orszag, "Spectral Methods for Problems in Complex Geometries," *Journal of Computational Physics* 37, no. 1 (1980): 70–92.

2. J. Brown, "Efficient Nonlinear Solvers for Nodal High-Order Finite Elements in 3D," *Journal of Scientific Computing* 45, no. 1–3 (2010): 48–63.

3. C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly, "From h to p Efficiently: Strategy Selection for Operator Evaluation on Hexahedral and Tetrahedral Elements," *Computers & Fluids* 43 (2011): 23–28.

4. M. Kronbichler and K. Kormann, "A Generic Interface for Parallel Cell-Based Finite Element Operator Application," *Computers & Fluids* 63 (2012): 135–147.

5. B. Gmeiner, M. Huber, L. John, U. Rüde, and B. Wohlmuth, "A Quantitative Performance Study for Stokes Solvers at the Extreme Scale," *Journal of Computational Science* 17 (2016): 509–521.

6. M. Kronbichler and K. Kormann, "Fast Matrix-Free Evaluation of Discontinuous Galerkin Finite Element Operators," *ACM Transactions on Mathematical Software* 45, no. 3 (2019): 1–40.

7. A. Abdelfattah, V. Barra, N. Beams, et al., "GPU Algorithms for Efficient Exascale Discretizations," *Parallel Computing* 108 (2021): 102841.

8. T. Kolev, P. Fischer, M. Min, et al., "Efficient Exascale Discretizations: High-Order Finite Element Methods," *International Journal of High Performance Computing Applications* 35, no. 6 (2021): 527–552.

9. D. Davydov, J.-P. Pelteret, D. Arndt, M. Kronbichler, and P. Steinmann, "A Matrix-Free Approach for Finite-Strain Hyperelastic Problems Using Geometric Multigrid," *International Journal for Numerical Methods in Engineering* 121, no. 13 (2020): 2874–2895.

10. J. Brown, V. Barra, N. Beams, et al., "Performance Portable Solid Mechanics via Matrix-Free *p*-Multigrid," arXiv preprint arXiv:2204.01722 (2022).

11. R. Schussnig, N. Fehn, P. Munch, and M. Kronbichler, "Matrix-Free Higher-Order Finite Element Methods for Hyperelasticity," *Computer Methods in Applied Mechanics and Engineering* 435 (2025): 117600.

12. S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM* 52, no. 4 (2009): 65–76.

13. A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "AI and Memory Wall," arXiv preprint arXiv:2403.14123 (2024).

14. T. Schneider, Y. Hu, X. Gao, J. Dumas, D. Zorin, and D. Panozzo, "A Large-Scale Comparison of Tetrahedral and Hexahedral Elements for Solving Elliptic PDEs With the Finite Element Method," *ACM Transactions on Graphics* 41, no. 3 (2022): 1–14.

15. A. Düster, S. Hartmann, and E. Rank, "P-FEM Applied to Finite Isotropic Hyperelastic Bodies," *Computer Methods in Applied Mechanics and Engineering* 192, no. 47–48 (2003): 5147–5166.

16. P. Wriggers, *Nonlinear Finite Element Methods* (Springer Verlag, 2008).

17. D. A. May, J. Brown, and L. Le Pourhiet, "pTatin3D: High-Performance Methods for Long-Term Lithospheric Dynamics," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE, 2014), 274–284.

18. J. M. Melenk, K. Gerdes, and C. Schwab, "Fully Discrete Hp-Finite Elements: Fast Quadrature," *Computer Methods in Applied Mechanics and Engineering* 190, no. 32–33 (2001): 4339–4364.

19. A. Solomonoff, "A Fast Algorithm for Spectral Differentiation," *Journal of Computational Physics* 98 (1992): 174–177.

20. D. Moxey, R. Amici, and M. Kirby, "Efficient Matrix-Free High-Order Finite Element Evaluation for Simplicial Elements," *SIAM Journal on Scientific Computing* 42, no. 3 (2020): C97–C123.

21. D. Still, N. Fehn, W. A. Wall, and M. Kronbichler, "Matrix-Free Evaluation Strategies for Continuous and Discontinuous Galerkin Discretizations on Unstructured Tetrahedral Grids," arXiv preprint arXiv:2509.10226 (2025).

22. P. Fischer, M. Min, T. Rathnayake, et al., "Scalability of High-Performance PDE Solvers," *International Journal of High Performance Computing Applications* 34, no. 5 (2020): 562–586.

23. M. Kronbichler and W. A. Wall, "A Performance Comparison of Continuous and Discontinuous Galerkin Methods With Fast Multigrid Solvers," *SIAM Journal on Scientific Computing* 40, no. 5 (2018): A3423–A3448.

24. T. C. Clevenger, T. Heister, G. Kanschat, and M. Kronbichler, "A Flexible, Parallel, Adaptive Geometric Multigrid Method for FEM," *ACM Transactions on Mathematical Software* 47 (2020): 1–27.

25. M. Wichrowski, P. Krzyżanowski, L. Heltai, and S. Stupkiewicz, "Exploiting High-Contrast Stokes Preconditioners to Efficiently Solve Incompressible Fluid–Structure Interaction Problems," *International Journal for Numerical Methods in Engineering* 124, no. 24 (2023): 5446–5470.

26. D. Arndt, W. Bangerth, D. Davydov, et al., "The Deal.II Finite Element Library: Design, Features, and Insights," *Computers & Mathematics With Applications* 81 (2021): 407–422.

27. D. Arndt, W. Bangerth, M. Bergbauer, et al., "The Deal. II Library, Version 9.7, Preprint," (2025).

28. J. Korelc, "Automatic Generation of Finite-Element Code by Simultaneous Optimization of Expressions," *Theoretical Computer Science* 187 (1997): 231–248.

29. C. Bischof, H. Buecker, B. Lang, A. Rasch, and J. Risch, "Extending the Functionality of the General-Purpose Finite Element Package Sepran by Automatic Differentiation," *International Journal for Numerical Methods in Engineering* 58 (2003): 2225–2238.

30. A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (SIAM, 2008).

31. W. S. Moses, V. Churavy, L. Paehler, et al., "Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Association for Computing Machinery, 2021), 1–16.

32. J. Korelc, "Multi-Language and Multi-Environment Generation of Nonlinear Finite Element Codes," *Engineering With Computers* 18 (2002): 312–327.

33. J. Korelc and P. Wriggers, *Automation of Finite Element Methods* (Springer International Publishing, 2016).

34. Wolfram Research, Inc., *Mathematica, Version 14.1*. (Wolfram Research, Inc., 2024).

35. J. Korelc, "Automation of Primal and Sensitivity Analysis of Transient Coupled Problems," *Computational Mechanics* 44 (2009): 631–649.

36. J. Korelc and S. Stupkiewicz, "Closed-Form Matrix Exponential and Its Application in Finite-Strain Plasticity," *International Journal for Numerical Methods in Engineering* 98 (2014): 960–987.

37. M. Rezaee-Hajidehi, P. Sadowski, and S. Stupkiewicz, "Deformation Twinning as a Displacive Transformation: Finite-Strain Phase-Field Model of Coupled Twinning and Crystal Plasticity," *Journal of the Mechanics and Physics of Solids* 163 (2022): 104885.

38. P. Sadowski, K. Kowalczyk-Gajewska, and S. Stupkiewicz, "Consistent Treatment and Automation of the Incremental Mori–Tanaka Scheme for Elasto-Plastic Composites," *Computational Mechanics* 60 (2017): 493–511.

39. M. A. Heroux, R. A. Bartlett, V. E. Howle, et al., "An Overview of the Trilinos Project," *ACM Transactions on Mathematical Software* 31, no. 3 (2005): 397–423.

40. P. G. Ciarlet, "The Finite Element Method for Elliptic Problems," *Journal of Applied Mechanics* 45 (1978): 968–969.

41. M. Adams, M. Brezina, J. Hu, and R. Tuminaro, "Parallel Multigrid Smoothing: Polynomial Versus Gauss–Seidel," *Journal of Computational Physics* 188, no. 2 (2003): 593–610.

42. D. Arndt, N. Fehn, G. Kanschat, et al., "ExaDG: High-Order Discontinuous Galerkin for the Exa-Scale," in *Software for Exascale Computing - SPPEXA 2016–2019*, ed. H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. Nagel (Springer International Publishing, 2020), 189–224.

43. T. Gruber, J. Eitzinger, G. Hager, and G. Wellein, "Likwid," Version v5, 2, 20 (2022).

44. R. W. Ogden, "Large Deformation Isotropic Elasticity–on the Correlation of Theory and Experiment for Incompressible Rubberlike Solids," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 326 (1972): 565–584.

45. L. R. Treloar, "The Mechanics of Rubber Elasticity," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 351 (1976): 301–330.

46. J. C. Simo and K. S. Pister, "Remarks on Rate Constitutive Equations for Finite Deformation Problems: Computational Implications," *Computer Methods in Applied Mechanics and Engineering* 46 (1984): 201–215.

47. J. C. Simo, "Numerical Analysis and Simulation of Plasticity," in *Handbook of Numerical Analysis*, vol. VI, ed. P. Ciarlet and J. Lions (Elsevier Science B.V, 1998), 184–499.

48. M. Wichrowski, P. Munch, M. Kronbichler, and G. Kanschat, "Smoothers With Localized Residual Computations for Geometric Multigrid Methods for Higher-Order Finite Elements," *SIAM Journal on Scientific Computing* 47, no. 3 (2025): B645–B664.

49. M. Rezaee-Hajidehi, K. Tůma, and S. Stupkiewicz, "A Note on Padé Approximants of Tensor Logarithm With Application to Hencky-Type Hyperelasticity," *Computational Mechanics* 68, no. 3 (2021): 619–632.

50. P. Hansbo, M. G. Larson, and K. Larsson, "Cut Finite Element Methods for Linear Elasticity Problems," in *Geometrically Unfitted Finite Element Methods and Applications: Proceedings of the UCL Workshop 2016* (Springer, 2017), 25–63.

51. N. M. Atallah, C. Canuto, and G. Scovazzi, "The Shifted Boundary Method for Solid Mechanics," *International Journal for Numerical Methods in Engineering* 122, no. 20 (2021): 5935–5970.

52. M. Bergbauer, P. Munch, W. A. Wall, and M. Kronbichler, "High-Performance Matrix-Free Unfitted Finite Element Operator Evaluation," *SIAM Journal on Scientific Computing* 47, no. 3 (2025): B665–B689.

53. M. Wichrowski, "A Geometric Multigrid Preconditioner for Discontinuous Galerkin Shifted Boundary Method," arXiv preprint arXiv:2506.12899 (2025).

54. M. Wichrowski, "Matrix-Free Evaluation of High-Order Shifted Boundary Finite Element Operators," arXiv preprint arXiv:2507.17053 (2025).

55. M. Wichrowski, "Matrix-Free Ghost Penalty Evaluation via Tensor Product Factorization," arXiv preprint arXiv:2503.00246 (2025).

56. K.-A. Mardal and R. Winther, "Preconditioning Discretizations of Systems of Partial Differential Equations," *Numerical Linear Algebra With Applications* 18, no. 1 (2011): 1–40.