Computer Methods in Applied Mechanics and Engineering 452 (2026) 118728

Contents lists available at ScienceDirect

Comput. Methods Appl. Mech. Engrg.

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

COMMET: Orders-of-magnitude speed-up in finite element method via batch-vectorized neural constitutive updates

Benjamin Alheit [a,b], Mathias Peirlinck [a,*], Siddhant Kumar [b,*]

[a] Department of BioMechanical Engineering, Faculty of Mechanical Engineering, Delft University of Technology, Netherlands
[b] Department of Materials Science and Engineering, Faculty of Mechanical Engineering, Delft University of Technology, Netherlands

ARTICLE INFO

Keywords:
Finite element method
Batch-vectorization
Neural constitutive models
High-performance computing
Compute graph optimization
Automatic differentiation
Distributed-memory parallelism (MPI)

ABSTRACT

Constitutive evaluations often dominate the computational cost of finite element (FE) simulations whenever material models are complex. Neural constitutive models (NCMs), i.e., neural network-based constitutive models, offer a highly expressive and flexible framework for modeling complex material behavior in solid mechanics. However, their practical adoption in large-scale FE simulations remains limited due to significant computational costs, especially in repeatedly evaluating stress and stiffness. NCMs thus represent an extreme case: their large computational graphs make stress and stiffness evaluations prohibitively expensive, restricting their use to small-scale problems. In this work, we introduce COMMET, an open-source FE framework whose architecture has been redesigned from the ground up to accelerate high-cost constitutive updates. Our framework features a novel assembly algorithm that supports batched and vectorized constitutive evaluations, compute-graph-optimized derivatives that replace automatic differentiation, and distributed-memory parallelism via MPI. These advances dramatically reduce runtime, with speed-ups exceeding three orders of magnitude relative to traditional non-vectorized automatic differentiation-based implementations. While we demonstrate these gains primarily for NCMs, the same principles apply broadly wherever for-loop based assembly or constitutive updates limit performance, establishing a new standard for large-scale, high-fidelity simulations in computational mechanics.

1. Introduction

The use of neural constitutive models (NCMs), i.e., neural network-based constitutive models in solid mechanics, has gained significant traction due to their exceptional expressivity, especially when compared to traditional constitutive models. This growing interest is largely motivated by the universal approximation theorem [1] which states that even relatively simple neural networks can approximate arbitrary continuous functions. This insight enables a paradigm shift in material modeling from human postulation of constitutive models to data-driven learning of material responses.

Traditionally, constitutive models were formulated by collecting limited experimental data and subsequently postulating physically admissible equations to fit this data. This process is inherently suboptimal, as it relies on the intuition of individual mechanicians to derive suitable equations – an approach unlikely to consistently yield the best representations of material behavior. NCMs offer an attractive alternative: they can flexibly learn to reproduce observed material responses, obviating the need to craft distinct models for different materials manually.

* Corresponding authors. Both senior authors contributed equally: author order decided by a coin toss.
E-mail addresses: mplab-me@tudelft.nl (M. Peirlinck), sid.kumar@tudelft.nl (S. Kumar).

https://doi.org/10.1016/j.cma.2026.118728
Received 30 September 2025; Received in revised form 17 December 2025; Accepted 2 January 2026
Available online 14 January 2026

Nonetheless, ensuring physical admissibility in NCMs remains crucial for model generalizability. This is of particular importance for the integration of NCMs in finite element (FE) solvers where non-physical material behavior typically leads to instabilities. Consequently, recent work has focused on incorporating mathematical constraints into neural network architectures to enforce physical principles as an inductive bias while preserving the networks' expressive capacity [2–20]. Moreover, notable work has been done to obtain frameworks for appropriately training such highly-parameterized NCMs in the context of solid mechanics [16,17,21–24]. As a result, NCMs have been successfully applied to model a broad range of material behaviors, including (anisotropic) hyperelasticity [6,9,21,25–27], viscoelasticity [28,29], plasticity [30–33], generalized standard materials [18], metamaterials [34], electroelasticity [35], and thermoelasticity [36,37].

While NCMs have demonstrated remarkable flexibility and expressivity in replicating complex material behavior, their widespread adoption is hindered by the high computational cost incurred during integration into numerical solvers, especially FE programs. Unlike traditional constitutive models, NCMs require evaluating large computational graphs, making the calculation of stress and stiffness tensors significantly more expensive in terms of floating point operations [38], thereby causing the cost of the assembly process to overshadow that of linear solves. For example, in a standard FE calculation, comparing a Mooney-Rivlin model with an NCM trained to replicate it showed that evaluating stress and stiffness accounted for about 5% of the total computed time with the Mooney Rivlin model but rose to 54% with the NCM [39]. Even for elastic material models that require no updating of state variables, computing material behavior with an NCM can become the dominant computational bottleneck.

Without targeted improvements to how solvers evaluate NCM computations, the practical utility of these models will remain limited to small-scale offline analyses. Bridging this gap between model fidelity and computational performance is therefore critical to enabling the routine use of NCMs in large-scale simulations across engineering and scientific domains.

While many studies have focused on improving FE solver performance through better CPU cache utilization and single instruction multiple data (SIMD) based vectorization [40–45], these efforts do not address the cost of complex constitutive evaluations, assuming relatively inexpensive material models (which is not the case for NCMs). Nonetheless, the techniques developed in those works – particularly with regard to batched computations and vectorization – can inspire performance optimizations for solvers utilizing NCMs. On the other hand, several studies have employed vectorized constitutive updates for FE solves and inverse problems in solid mechanics [6,21,34,35,46,47]. However, these efforts have primarily been implemented in non-performant languages such as Python – due to the prevalence of Python-based NCM training environments – or using JAX [48], which lacks sufficient support for scaling across multiple compute nodes. Moreover, they have not undergone a quantitative and systematic investigation into their scaling behavior or comprehensive performance analysis. Lastly, some efforts [7,25,37,38,49] have focused on retrofitting NCMs into legacy FE solvers, including commercial software such as Abaqus [50] and Ansys [51]. While these solvers support user-defined material models and routines, their underlying architectures do not support vectorization strategies across multiple quadrature points and elements and therefore, suffer from computational bottlenecks. To fully leverage the representational power of NCMs in practical applications, it is essential that the surrounding FE solver architecture be re-imagined from the ground up.

To address these challenges, we introduce a novel FE assembly algorithm that enables batched and vectorized evaluation of the constitutive model. This vectorization strategy requires simultaneous access to state variables of multiple material points, which in turn necessitates a redesign of the conventional FE element-level assembly process. To allow fine-grained control over the performance trade-offs introduced by batching, the solver architecture includes a mechanism to explicitly manage the *batch size* – the number of constitutive updates processed in a single vectorized operation. Typically for NCMs, the stress and stiffness are calculated by automatic differentiation, which can be slow for NCMs with large computational graphs. Therefore, we further improve FE performance by replacing automatic differentiation (AD) with compute graph optimized (CGO) implementations of NCMs. In CGO, we demonstrate that a carefully designed analytical treatment of NCMs can outperform AD by enabling efficient analytical computation of first and second derivatives. This approach significantly reduces both memory usage and computation time. Finally, we show that batch-vectorization is compatible with distributed-memory parallelism using message-passing interface (MPI) [52], effectively marrying SIMD-based parallelism within each compute node with MPI-based parallelism across multiple compute nodes. This hierarchical approach to parallelization enables highly scalable and efficient large-scale simulations. For the scope of this work, we focus on hyperelastic material behavior, while noting that the overall framework is agnostic to NCM architecture and can also be applied similarly to path-dependent NCMs.

As a companion to this work, we introduce **COMMET** (*COmputational Mechanics and Machine learning Toolbox*) – an open-source software incorporating the technologies introduced in this work. Specifically, COMMET provides dedicated Python-based modules for implementing and training NCMs, as well as a C++-based FE solver which has been developed on top of the Deal.II library [53] with batch-vectorization, CGO, and MPI parallelism for scalable simulations using NCMs. We invite the research community to utilize and contribute to COMMET to enable broader adoption and further exploration of data-driven constitutive modeling in solid mechanics.

The remaineder of this contribution is organized as follows. Key background on finite elements and neural constitutive models is summarized in Section 2. We provide details on the global and batch-vectorized assembly algorithms as well as CGO and MPI parallelization in Section 3. The results of extensive benchmarks of the vectorization algorithms and developed solver are presented and discussed in Section 4. In Section 5, we present demonstration of the developed solver's capabilities by simulating the passive filling of a human heart while using an NCM to model the material behavior. Finally, we provide concluding remarks in Section 6.

2. Background and preliminaries

Before detailing the batch-vectorized FE assembly algorithm for efficient NCM-based large-scale modeling (see Section 3), we briefly outline key preliminary knowledge on FE and NCMs in Sections 2.1 and 2.2, respectively.

2.1. Finite elements for solid mechanics problems

Here, we concisely introduce the mathematical components of FE that are relevant to this work. For a more complete introduction to FE, readers are referred to standard textbooks, e.g., [54–56]. In the context of boundary value problems in nonlinear solid mechanics, the global displacement vector $\boldsymbol{d}$ is typically obtained by using the Newton-Raphson (NR) method (or similar gradient-based methods) to solve the weak form of the linear momentum balance discretized over a domain. In other words, $\boldsymbol{d}$ is iteratively updated by $\boldsymbol{d} \leftarrow \boldsymbol{d} + \Delta\boldsymbol{d}$, where

$$\boldsymbol{K}\Delta\boldsymbol{d} = -\boldsymbol{r}. \tag{1}$$

Here, $\boldsymbol{K}$ and $\boldsymbol{r}$ represent the global stiffness matrix and global residual vector, respectively, of the form

$$\boldsymbol{K} = \begin{bmatrix} K_{11}^{11} & K_{12}^{11} & K_{13}^{11} & K_{11}^{12} & K_{12}^{12} & K_{13}^{12} & \dots \\ K_{21}^{11} & K_{22}^{11} & K_{23}^{11} & K_{21}^{12} & K_{22}^{12} & K_{23}^{12} & \dots \\ K_{31}^{11} & K_{32}^{11} & K_{33}^{11} & K_{31}^{12} & K_{32}^{12} & K_{33}^{12} & \dots \\ K_{11}^{21} & K_{12}^{21} & K_{13}^{21} & K_{11}^{22} & K_{12}^{22} & K_{13}^{22} & \dots \\ K_{21}^{21} & K_{22}^{21} & K_{23}^{21} & K_{21}^{22} & K_{22}^{22} & K_{23}^{22} & \dots \\ K_{31}^{21} & K_{32}^{21} & K_{33}^{21} & K_{31}^{22} & K_{32}^{22} & K_{33}^{22} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}, \qquad \boldsymbol{r} = \begin{bmatrix} r_1^1 \\ r_2^1 \\ r_3^1 \\ r_1^2 \\ r_2^2 \\ r_3^2 \\ \vdots \end{bmatrix}, \tag{2}$$

where K_{ij}^{IJ} denotes the tangent stiffness for the pair of nodes $I, J \in \{1, 2, \dots, n_{\text{nodes}}\}$ in the spatial directions $i, j \in \{1, 2, 3\}$ and r_i^I denotes the force residual for node I in direction i. These entries are given by the following expressions, which result from the discretization of the weak form of the balance of equilibrium with negligible body forces in the current configuration (see, e.g. [56] Section 4.2.3):

$$K_{ij}^{IJ} = \int_{\Omega_0} \nabla_{\boldsymbol{x}}\phi_k^I \left[\delta_{ij}\tau_{kl} + \mathbb{c}_{ikjl}\right] \nabla_{\boldsymbol{x}}\phi_l^J \, \mathrm{d}\Omega_0, \tag{3}$$

$$r_i^I = \int_{\Omega_0} \tau_{ij} \nabla_{\boldsymbol{x}}\phi_j^I \, \mathrm{d}\Omega_0 - \int_{\Gamma_N} \phi^I \bar{t}_i \, \mathrm{d}\Gamma_N. \tag{4}$$

Here, Ω_0 denotes the reference material domain, $\phi^I : \mathbb{R}^3 \to \mathbb{R}$ is the shape function of node I, $\nabla_{\boldsymbol{x}}\phi^I$ represents the gradient of ϕ^I in the spatial configuration, δ is the Kronecker delta, $\boldsymbol{\tau}$ is the Kirchhoff stress, $\mathbb{c}$ is the spatial stiffness tensor, Γ_N is the portion of the boundary to which a traction condition is prescribed, and $\bar{\boldsymbol{t}}$ is a prescribed traction. Additionally, the Einstein summation convention is invoked for repeated indices.

Both $\boldsymbol{\tau}$ and $\mathbb{c}$ are constitutive quantities that are functions of the deformation gradient $\boldsymbol{F} = \boldsymbol{I} + \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{X}}$ and internal state variables, where $\boldsymbol{I}$ is the identity tensor, $\boldsymbol{u}$ is the displacement field, and $\boldsymbol{X}$ represents the position in the material domain.

The integrals in Eqs. (3) and (4) are evaluated using numerical quadrature, i.e., as the sum of the integrand evaluated at a finite number of n_{q} quadrature points, within a finite number of elements n_{e}, and weighted by the *volume* of the quadrature point within that element $w^{\text{e,q}}$

$$\int_{\Omega_0} \tau_{ij} \nabla_{\boldsymbol{x}}\phi_j^I \, \mathrm{d}\Omega_0 \approx \sum_{\text{e}}^{n_{\text{e}}} \sum_{\text{q}}^{n_{\text{q}}} w^{\text{e,q}} \tau_{ij}^{\text{e,q}} \nabla_{\boldsymbol{x}}\phi_j^{I,\text{e,q}}, \tag{5}$$

$$\int_{\Omega_0} \nabla_{\boldsymbol{x}}\phi_k^I \left[\delta_{ij}\tau_{kl} + \mathbb{c}_{ikjl}\right] \nabla_{\boldsymbol{x}}\phi_l^J \, \mathrm{d}\Omega_0 \approx \sum_{\text{e}}^{n_{\text{e}}} \sum_{\text{q}}^{n_{\text{q}}} w^{\text{e,q}} \nabla_{\boldsymbol{x}}\phi_k^{I,\text{e,q}} \left[\delta_{ij}\tau_{kl}^{\text{e,q}} + \mathbb{c}_{ikjl}^{\text{e,q}}\right] \nabla_{\boldsymbol{x}}\phi_l^{J,\text{e,q}}. \tag{6}$$

The superscript $(\cdot)^{\text{e,q}}$ denotes the evaluation at quadrature point q in element e.

Evaluating the summations in Eqs. (5) and (6) lies at the heart of the so-called *assembly process* in FE methods. Traditionally, the assembly is implemented using nested for-loops: iterating over each element, then over quadrature points, and finally over pairs of nodes within each element. In this approach, the number of constitutive calculations ($\boldsymbol{F} \to \boldsymbol{\tau}, \mathbb{c}$) performed at each NR iteration amounts to $n_{\text{q}} \times n_{\text{e}}$, which is often large. Hence, rapid computation of the constitutive map for many material points is crucial for the performant evaluation of the necessary integrals in Eqs. (5) and (6), and hence, is crucial for a fast assembly process.

For completeness, and for comparison with the new algorithms presented subsequently, the traditional algorithm for FE assembly is presented in Algorithm 1.

2.2. Neural constitutive models

For the scope of this work, we focus on hyperelastic material behavior, while noting that the overall framework is material-agnostic and can also be applied to path-dependent material behaviors. To model (anisotropic) hyperelastic material behavior, we postulate a strain energy density $\Psi(\boldsymbol{F}, \mathcal{A})$ that is a function of the deformation gradient $\boldsymbol{F}$ and a set of n_{sv} structural vectors $\mathcal{A} = \left\{\boldsymbol{A}^1, \dots, \boldsymbol{A}^{n_{\text{sv}}}\right\}$. The Kirchhoff stress and spatial stiffness tensor are then given by (see derivation in Appendix A.1)

$$\tau_{ij} = \frac{\partial \Psi}{\partial F_{iJ}} F_{jJ}, \qquad \mathbb{c}_{ijkl} = F_{jJ} \frac{\partial^2 \Psi}{\partial F_{iJ} \partial F_{kL}} F_{lL} - \delta_{ik}\tau_{jl}. \tag{7}$$

Algorithm 1 Traditional algorithm for finite element system assembly.

for $\mathrm{e} = 1, \dots, n_\mathrm{e}$ **do** ▷ Loop over elements
  for $\mathrm{q} = 1, \dots, n_\mathrm{q}$ **do** ▷ Loop over quadrature points for element
    $\boldsymbol{F} \leftarrow \boldsymbol{I} + \sum_I \boldsymbol{u}^I \otimes \nabla_{\boldsymbol{X}} \phi^{I,\mathrm{e,q}}$ ▷ Evaluate trial $\boldsymbol{F}$ at quadrature point
    $\boldsymbol{\tau}, \mathbb{c} \leftarrow$ constitutive_model($\boldsymbol{F}$) ▷ Evaluate stress and stiffness at quadrature point
    for $I \in$ {Nodes on element e} **do** ▷ Loop over nodes for element
      $r_i^I \leftarrow r_i^I + w^{\mathrm{e,q}} \tau_{ij}^{\mathrm{e,q}} \nabla_{\boldsymbol{x}} \phi_j^{I,\mathrm{e,q}}$ ▷ Add contribution to residual Eq. (5)
      for $J \in$ {Nodes on element e} **do** ▷ Inner loop over nodes for element
        $K_{ij}^{IJ} \leftarrow K_{ij}^{IJ} + w^{\mathrm{e,q}} \nabla_{\boldsymbol{x}} \phi_k^{I,\mathrm{e,q}} \left[\delta_{ij} \tau_{kl}^{\mathrm{e,q}} + \mathbb{c}_{ikjl}^{\mathrm{e,q}} \right] \nabla_{\boldsymbol{x}} \phi_l^{J,\mathrm{e,q}}$ ▷ Add stiffness contribution Eq. (6)
      end for
    end for
  end for
end for

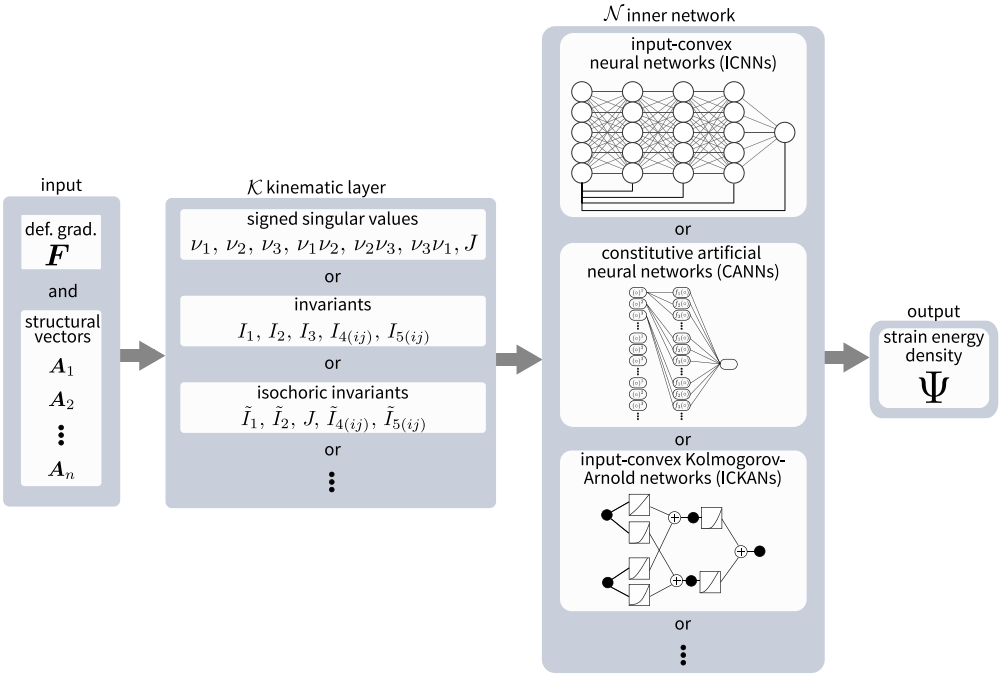

Fig. 1. High-level architecture of a neural constitutive model (NCM). The hyperelastic strain energy density formulated as a composition of two functions $\mathcal{K}$ and $\mathcal{N}$. The kinematic layer $\mathcal{K}$ maps the deformation gradient and structural vectors to a set of invariant kinematic scalars, ensuring objectivity and material symmetry. These scalars then serve as input to the inner network $\mathcal{N}$, typically a neural network architecture designed to satisfy convexity conditions required for polyconvexity. The inner network outputs the final strain energy density, which is used to derive the stress and stiffness needed in finite element simulations.

In order to satisfy the axioms of objectivity and material symmetry, the strain energy density is typically not postulated in terms of the deformation gradient $\boldsymbol{F}$ directly, but instead postulated in terms of a set of kinematic scalar values that are invariant to the choice of basis for the reference or current configuration. Examples of these scalars include the signed singular values of $\boldsymbol{F}$, ν_i, $i = 1, 2, 3$ [3,57].[1] Alternatively, invariants of the right Cauchy-Green tensor $\boldsymbol{C} = \boldsymbol{F}^T \boldsymbol{F}$, e.g.,

$$I_1 = \mathrm{tr}(\boldsymbol{C}), \qquad I_2 = \frac{1}{2}\left[\mathrm{tr}(\boldsymbol{C})^2 - \mathrm{tr}\left(\boldsymbol{C}^2\right)\right], \qquad I_3 = \det \boldsymbol{C}, \qquad I_{4,ij} = \boldsymbol{A}^i \cdot \boldsymbol{C}\boldsymbol{A}^j, \tag{8}$$

where $i, j \in \{1, \dots, n_{\mathrm{sv}}\}$, can also be used as inputs for the strain energy density. Hence, in an abstract sense, the strain energy density can be thought of as a composition of two functions (see Fig. 1): $\mathcal{K}$ which maps the deformation gradient and structural vectors to a set of kinematic scalars, and $\mathcal{N}$ which maps those scalars to the final strain energy density, i.e.,

$$\Psi(\boldsymbol{F}, \mathcal{A}) = \mathcal{N}(\mathcal{K}(\boldsymbol{F}, \mathcal{A})). \tag{9}$$

[1] Additional requirements of Π invariance are required for objectivity and material symmetry in this case; see [3,57] for details.

In the traditional constitutive modeling paradigm, $\mathcal{N}$ is typically a simple analytical function that is postulated by mechanicians. However, in the NCM paradigm, $\mathcal{N}$ takes the form of a neural network architecture that is agnostic to its use as a constitutive model. In this sense, $\mathcal{N}$ is simply a highly expressive ansatz for a constitutive equation that is inspired by machine learning. In accordance with machine learning (ML) terminology, we term $\mathcal{K}$ as the *kinematic layer* and $\mathcal{N}$ as the *inner network*. To ensure polyconvexity of the NCM in $\boldsymbol{F}$, certain convexity conditions are required for the inner network. These conditions are dependent on the choice of the kinematic scalars. For example, if the set of kinematic scalar inputs is $\{\nu_1, \nu_2, \nu_3, \nu_1\nu_2, \nu_2\nu_3, \nu_3\nu_1, J\}$, then input-convexity[2] of $\mathcal{N}$ is sufficient to guarantee polyconvexity in $\boldsymbol{F}$ [3,12,57]. However, if the scalar inputs are themselves polyconvex functions in $\boldsymbol{F}$ (e.g., I_1, I_2, and I_3), then one requires that $\mathcal{N}$ is convex and monotonically increasing in its inputs [6] for the NCM to be polyconvex in $\boldsymbol{F}$ overall. To this end, some inner networks that have been used to date include input convex neural networks (ICNNs) [9,21,58], monotonically non-decreasing input convext neural networks (MICNNs) [59], constitutive artificial neural networks (CANNs) [5,8,38,60], and input convex Kolmogorov-Arnold networks (ICKANs) [6,61]. Further details on these specific NCM-based architectures are provided in Appendix B for completeness.

3. COMMET: vectorized and batched FE solver enabling efficient NCM implementation

Central to our approach is a novel element assembly algorithm that enables batched and vectorized evaluation of NCMs (see Section 3.1). This allows the solver to fully utilize the capabilities of modern CPUs and memory hierarchies, improving cache behavior. To enable large-scale simulations, we parallelize the solver using MPI, supporting distributed-memory computation across multiple compute nodes while maintaining consistent batched execution of NCM evaluations (see Section 3.3). Finally, we further reduce the cost of constitutive evaluations by replacing automatic differentiation-based computations of the stress and tangent stiffness with compute graph optimization (see Section 3.2).

3.1. Assembly vectorization and batching

In the traditional assembly algorithm (see Algorithm 1), one loops over each element and quadrature point and evaluates the constitutive updates sequentially, as shown in Fig. 2 (a).

In contrast, we propose to bundle, i.e., *batch* the constitutive update calculations across multiple quadrature points – both within and across multiple elements – and evaluate them in parallel through a single NCM constitutive update instance (i.e., *vectorization*).

The traditional FE assembly procedure (Algorithm 1) does not readily allow vectorization of the constitutive updates since the state variables are overwritten from one quadrature point to the next throughout the assembly procedure. Hence, we alter this procedure by creating tables for the relevant state variables across all elements and quadrature points. Each table is stored in a structured and contiguous block in the memory. We introduce this assembly process as the **globally vectorized** algorithm schematically shown in Fig. 2 (b). In this algorithm, we perform the constitutive update across the entire mesh in a single vectorized operation as presented in Algorithm 2.

While the globally vectorized algorithm is arguably the simplest way to leverage vectorization, memory constraints only render it practical for small mesh sizes. The RAM on a compute node is inherently limited, therefore a contiguous block of memory may not be available to store a large table of state variables for a high-resolution mesh. We address this issue by dividing the state variables table into multiple batches of smaller but equal sizes, with each batch still stored in a contiguous memory block. Fig. 2 (c) introduces the second **batch-vectorized** algorithm which processes the constitutive update in user-chosen batch sizes as detailed in Algorithm 3. Practically, we recommend using the batch-vectorized algorithm and determining the optimal batch size subject to RAM usage constraints for each machine through computational experiments, as demonstrated in Section 4.1. Note that the globally vectorized algorithm is equivalent to the batch-vectorized algorithm with batch size equal to the total number of material/quadrature points (subject to memory constraints).

Overall, the batch-vectorization approach leverages the Single Instruction Multiple Data (SIMD) [62] paradigm to accelerate the FE assembly. SIMD refers to data-level parallelism where the same instructions are executed simultaneously on multiple data points, given that there are no interdependencies between data points or their corresponding instructions. As a simple example, the addition of two arrays can be performed in two ways: using a non-SIMD approach, where each element is processed sequentially in a loop, e.g.,

```
for(unsigned int i = 0; i < array_size; i++)
    a[i] = b[i] + c[i];
```

or using the faster SIMD approach, where a single instruction adds multiple elements in parallel, e.g.,

```
a[0:array_size] = b[0:array_size] + c[0:array_size];
```

Within the FE context and the proposed algorithm, the computational benefit of this SIMD approach emerges from multiple aspects, the most important ones being as follows.

Data prefetching As illustrated in Fig. 3, modern computers typically employ two types of RAM: dynamic random access memory (DRAM) and static random access memory (SRAM) [63].

[2] The architecture of an input-convex neural network [58] is designed to ensure that the output is identically convex with respect to the inputs, regardless of the network's weights.

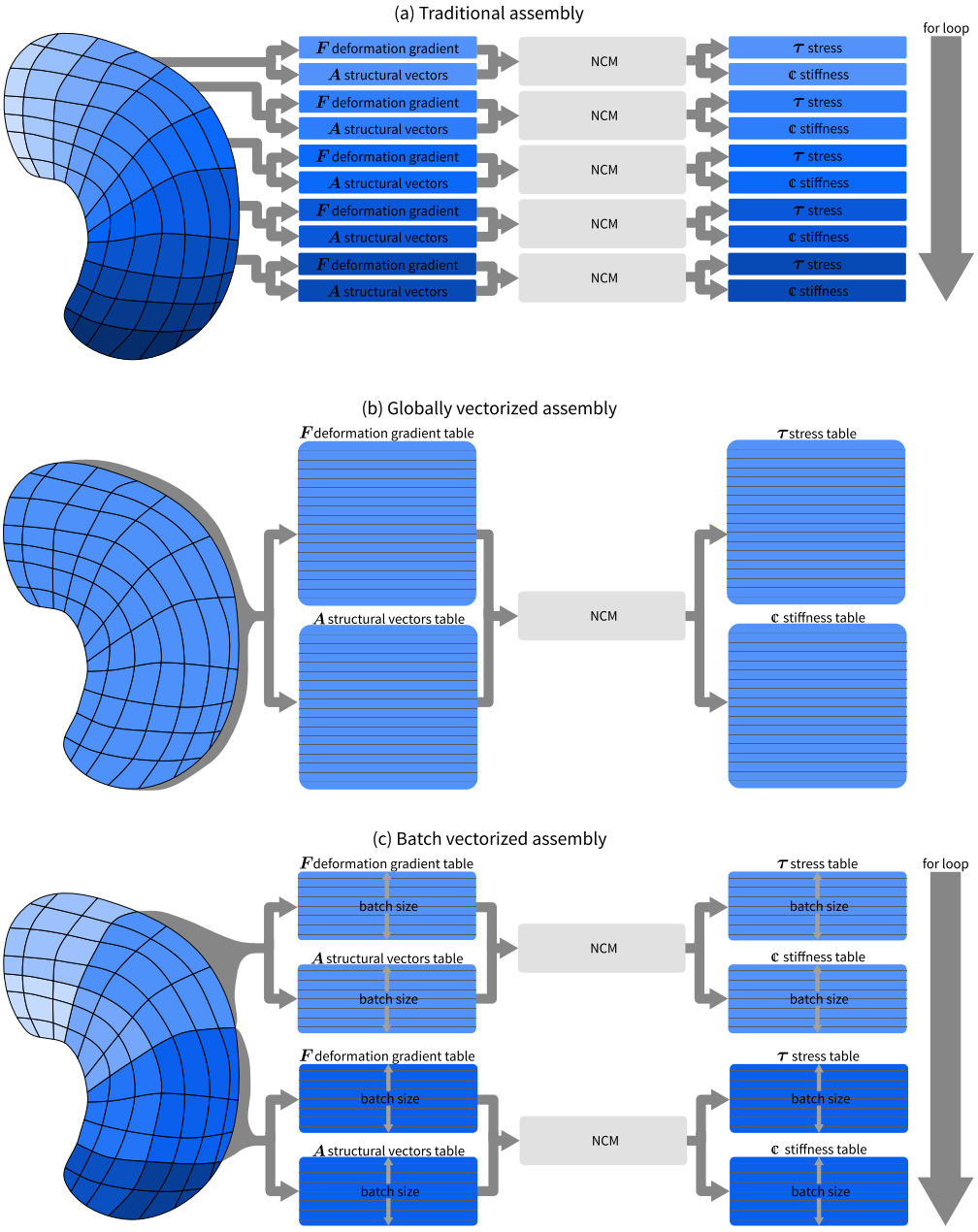

Fig. 2. Schematic comparison of constitutive update strategies in finite element assembly: (a) the traditional approach whereby the stress and stiffness are calculated for one quadrature point at a time, (b) the globally vectorized approach where the state variables (i.e. deformation gradient and structural vectors in the case of hyperelasticity) for all quadrature points are collected in tables from which associated stress and stiffness tables are calculated in a single vectorized computation, and (c) the batch-vectorized approach where batches of quadrature points are processed at a time.

In general, accessing data from SRAM is roughly two orders of magnitude faster than reading data from DRAM. However, SRAM requires more physical space per byte and is significantly more expensive to manufacture. As a result, modern systems use a relatively small amount of SRAM – ranging from a few kilobytes to several megabytes – integrated directly into the CPU as a *cache*. In contrast, DRAM is used for *main memory*, typically on the order of gigabytes, and is connected to the CPU via a memory controller and memory bus located on the motherboard.

The cache itself is divided into three levels, L1, L2, and L3 illustrated in Fig. 3. These L1, L2, and L3 SRAM caches have increasing sizes and decreasing speeds. For example, a 24-core Intel(R) Xeon(R) Gold 6248R chip has L1, L2, and L3 has cache level sizes of 3 MiB, 48 MiB, and 71.5 MiB, respectively, and data-retrieval latencies amounting to 4 cycles, 12 cycles, and 44 cycles, respectively

Algorithm 2 Algorithm for **globally vectorized** finite element system assembly.

count ← 0
for e = 1, … , n_{e} **do** ▷ Loop over elements
  for q = 1, … , n_{q} **do** ▷ Loop over quadrature points
    $\{\boldsymbol{F}\}$[count] ← $\boldsymbol{I} + \sum_I \boldsymbol{u}^I \otimes \nabla_{\boldsymbol{X}} \phi^{I,\mathrm{e,q}}$ ▷ Evaluate trial $\boldsymbol{F}$ at quadrature point
    count + +
  end for
end for
$\{\Psi\}, \{\boldsymbol{\tau}\}, \{\mathbb{c}\}$ ← vectorized_constitutive_model($\{\boldsymbol{F}\}$)
count ← 0
for e = 1, … , n_{e} **do** ▷ Loop over elements
  for q = 1, … , n_{q} **do** ▷ Loop over quadrature points
    $\boldsymbol{\tau}, \mathbb{c}$ ← $\{\boldsymbol{\tau}\}$[count], $\{\mathbb{c}\}$[count] ▷ Look up quadrature point values
    for $I \in$ {Nodes on element e} **do** ▷ Loop over nodes for element
      $r_i^I \leftarrow r_i^I + w^{\mathrm{e,q}} \tau_{ij}^{\mathrm{e,q}} \nabla_{\boldsymbol{x}} \phi_j^{I,\mathrm{e,q}}$ ▷ Add contribution to residual Eq. (5)
      for $J \in$ {Nodes on element e} **do** ▷ Inner loop over nodes for element
        $K_{ij}^{IJ} \leftarrow K_{ij}^{IJ} + w^{\mathrm{e,q}} \nabla_{\boldsymbol{x}} \phi_k^{I,\mathrm{e,q}} \left[\delta_{ij} \tau_{kl}^{\mathrm{e,q}} + \mathbb{c}_{ikjl}^{\mathrm{e,q}} \right] \nabla_{\boldsymbol{x}} \phi_l^{J,\mathrm{e,q}}$ ▷ Add stiffness contribution Eq. (6)
      end for
    end for
  end for
end for

Algorithm 3 Algorithm for **batch-vectorized** assembly.

accumulated_count, count ← 0
while accumulated_count < n_{e} **do**
  count ← 0
  while count < n_{b} ς count + accumulated_count < n_{e} **do** ▷ Loop over element batch
    $\{\boldsymbol{F}\}$[count] ← $\boldsymbol{I} + \sum_I \boldsymbol{u}^I \otimes \nabla_{\boldsymbol{X}} \phi^{I,\mathrm{e,q}}$ ▷ Evaluate trial $\boldsymbol{F}$ at quadrature point
    count + +
  end while
  $\{\Psi\}, \{\boldsymbol{\tau}\}, \{\mathbb{c}\}$ ← vectorized_constitutive_model($\{\boldsymbol{F}\}$)
  count ← 0
  while count < n_{b} ς count + accumulated_count < n_{e} **do** ▷ Loop over element batch
    for q = 1, … , n_{q} **do** ▷ Loop over quadrature points
      $\boldsymbol{\tau}, \mathbb{c}$ ← $\{\boldsymbol{\tau}\}$[count], $\{\mathbb{c}\}$[count] ▷ Look up quadrature point values
      for $I \in$ {Nodes on element e} **do** ▷ Loop over nodes for element
        $r_i^I \leftarrow r_i^I + w^{\mathrm{e,q}} \tau_{ij}^{\mathrm{e,q}} \nabla_{\boldsymbol{x}} \phi_j^{I,\mathrm{e,q}}$ ▷ Add contribution to residual Eq. (5)
        for $J \in$ {Nodes on element e} **do** ▷ Inner loop over nodes for element
          $K_{ij}^{IJ} \leftarrow K_{ij}^{IJ} + w^{\mathrm{e,q}} \nabla_{\boldsymbol{x}} \phi_k^{I,\mathrm{e,q}} \left[\delta_{ij} \tau_{kl}^{\mathrm{e,q}} + \mathbb{c}_{ikjl}^{\mathrm{e,q}} \right] \nabla_{\boldsymbol{x}} \phi_l^{J,\mathrm{e,q}}$ ▷ Add stiffness contribution Eq. (6)
        end for
      end for
      count + +
    end for
  end while
  accumulated_count ← accumulated_count + count
end while

[64,65]. The L1 SRAM cache is further subdivided into L1D and L1I for data and instruction caching, respectively. Main memory, although much larger than cache, has a latency on the order of 200 cycles [64,65]. Hence, memory is arranged in a hierarchical sense, with decreasing size and increasing speed from main memory to L3, L2, and L1 cache, as illustrated by the hierarchical arrows shown in Fig. 3.

When a CPU executes an operation on data, it must first load that data into its *registers*, which are extremely small and fast memory locations (typically 8 - 512 bits in size) embedded directly in the CPU. When loading these data into registers, the hierarchy is traversed; the L1 cache is checked first, then L2, L3, and finally, if the data is not contained in the cache, it will be retrieved from main memory. Obtaining the desired data from main memory as opposed to cache is termed a *cache miss* and is costly as the CPU often remains idle while it waits for the necessary data to arrive.

Modern CPUs can perform arithmetic operations far faster than they can fetch data from main memory [63]. For instance, a floating-point addition typically completes in about one clock cycle [66], whereas retrieving a single double-precision value from

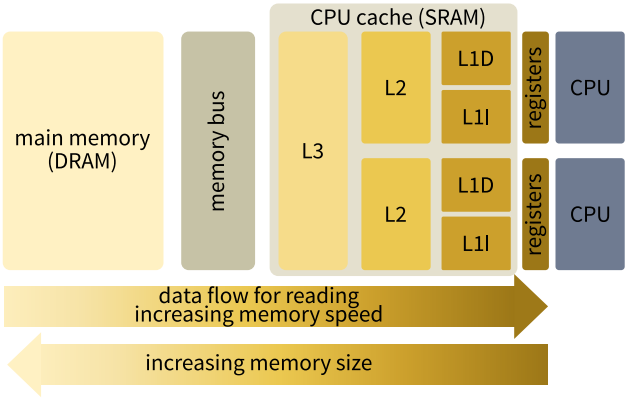

Fig. 3. Schematic computer memory hierarchy with decreasing latency and increasing speed from left to right. The hierarchy consists of main memory (consisting of DRAM); CPU cache (consisting of SRAM) which is further divided into L3, L2, and L1 cache; and registers. The L1 cache is further divided into an L1D cache for storing data and an L1I cache for storing instructions, in contrast to the L3 and L2 cache which store both data and instructions. Typically, each CPU has a dedicated L1 and L2 cache while the L3 cache is often shared between multiple CPUs.

main memory can take around 200 cycles. This stark imbalance means that performance is often limited not by compute speed but by memory latency. To sustain efficiency, data must therefore be available in the CPU cache at the moment it is needed.

To mitigate this performance gap between computation speed and memory latency, modern CPUs employ a technique known as *prefetching*. Prefetching involves predicting which memory locations will be accessed in the near future and pre-emptively loading that data into cache before it is explicitly requested by the CPU. When data is structured contiguously in memory, such as in arrays, it enables the hardware prefetcher to recognize regular patterns and preemptively load upcoming cache lines. In contrast, if data is scattered across memory in a non-contiguous fashion – as can occur with data structures like linked-lists or trees – prefetching becomes much less effective, which leads to more cache misses and stalls as the CPU waits on data from main memory.

In the context of FE assembly – often implemented using object-oriented programming, though not exclusively – state variables are typically passed to or stored within individual element instances. This results in a non-contiguous memory layout for the state variables. When such a layout is used with NCMs, it can lead to frequent cache misses and processor stalling due to repeated accesses to main memory. In contrast, our approach constructs a contiguous data structure for the state variables, enabling hardware prefetching to minimize cache misses and significantly improve performance.

Vector registers Modern CPUs are equipped with *wide vector registers* that support SIMD operations. Common examples include 128-bit SSE2, 256-bit AVX, and 512-bit AVX-512 registers. Since double-precision floating-point numbers occupy 64 bits, these registers can hold 2, 4, and 8 such values, respectively.

These CPUs also support vectorized instructions that operate on all elements within a vector register simultaneously. For instance, the assembly instruction `VADDPD zmm0, zmm1, zmm2` performs element-wise addition of the 512-bit registers `zmm1` and `zmm2`, storing the result in `zmm0`. This enables 8 double-precision additions to be performed in a single instruction.

To use these vectorized operations, the data must be properly aligned and structured in memory to map cleanly onto the registers. As with prefetching, contiguous and well-aligned memory layouts are essential for maximizing the performance benefits of SIMD execution.

Overhead of launching compute kernels Beyond hardware-related efficiencies, software implementation choices can also introduce performance bottlenecks. Traditional FE methods – without the use of neural constitutive models (NCMs) – have typically been implemented in high-performance languages such as Fortran and C/C++. However, with the rise of ML in computational mechanics, NCMs are now almost exclusively implemented and trained in high-level scripting environments like Python [67], where frameworks such as PyTorch [68] and TensorFlow [69] operate.

To bridge this software gap for integration of NCMs in FE, recent works [21,34] have developed FE and similar numerical methods around NCMs in Python-like environments. While the NCM component benefits from the highly optimized C/C++-like backends of these ML libraries, the remaining FE components suffer from well-known performance limitations of Python-like environments, thereby becoming computational bottlenecks.

An underexplored alternative is exporting trained NCM models – specifically their weights and computational graphs – from Python-based ML frameworks, and importing them into high-performance languages like C/C++, where the rest of the FE code is executed efficiently. There are two ways that this can be done: reimplement the NCM model in the C/C++ codebase by hand, or export the NCM model via a graph-compilation tool such as TorchScript [70]. Current software environments for ML and high-performance languages like C/C++ only support the latter as hard-coding neural networks with complex architectures and large numbers of parameters would be too labor-intensive.

Each call that is made to the graph-compiled model requires several overhead operations, e.g. the data type, layout, and contiguity of the input tensors is checked, reference counters are updated, memory is allocated, etc. The cost of these operations is known as the "compute kernel launch overhead". This cost is distinct from the cost of instantiating an NCM and loading its weights from memory,

i.e. even for a single NCM instance the compute kernel launch overhead is incurred for each call made to that instance. In traditional FE assembly algorithms (see Algorithm 1), the compute kernel is launched for every material point [38], and so the incurred compute kernel launch overhead is multiplied by the number of material points.

To mitigate this, we leverage vectorization by processing multiple material points simultaneously. This approach reduces the incurred kernel launch overhead significantly, requiring only one invocation per batch, and thus achieving a speed-up proportional to the batch size.

Through low-level monitoring tools and CPU performance counters, one can quantify the benefits of individual optimization aspects using micro-benchmarks. However, in the context of FE methods, these aspects are often tightly intertwined, making it difficult to isolate their individual contributions. As a result, our work focuses on the overall speed-up achieved from all contributing factors combined, while acknowledging that a detailed breakdown of each low-level contribution is beyond the scope.

Remark 1. The necessity of instructions being identical across all data poses a challenge for constitutive updates that may have control flow which differs from one material point to the next, e.g. updates that require local Newton-Raphson iterations for the so-called return mapping algorithm in plasticity. A possible solution in this case is to continue iterating across the whole batch until the state variables for all material points have converged. We note, however, that many path dependent NCMs that have been presented in the literature do not require solving of nonlinear equations for the evolution state variables through such iterative schemes [28,30,31,71], and so the proposed global and batch-vectorized algorithms can be applied as is.

3.2. Compute graph optimization

When using NCMs in FE solvers, it is necessary to compute the first and second derivatives of the strain energy density function to obtain the stress and tangent stiffness in accordance with Eq. (7). While automatic differentiation (AD) is a widely adopted approach for this purpose due to its accuracy and ease of integration [47,72–74], it incurs significant performance overhead - particularly in reverse-mode implementations [75,76] required for computing second-order derivatives (Hessians). This overhead stems from the need to construct and traverse computational graphs multiple times, along with the storage of intermediate states during evaluation, which together lead to increased memory consumption and compute time. Numerical differentiation via input perturbation offers no clear advantage: it typically requires multiple evaluations of the NCM, and does not provide accurate gradients, potentially leading to unstable simulations. Analytically derived gradients, in contrast, can significantly reduce both memory usage and computational cost. However, unlike traditional constitutive models, they are not straightforward to obtain for NCMs with large computational graphs.

To further reduce execution time of the constitutive updates, we introduce a general framework to obtain analytical derivatives of NCMs with just one forward pass (i.e., a single evaluation of the NCM). This is significantly faster than AD while providing exact gradients unlike numerical differentiation. We term implementations that make use of such analytical derivations as *compute graph optimization* (CGO). Without loss of generality, this work presents the formulation for hyperelasticity, with the extension to path-dependent materials identified as a direction for future research.

The core idea is to compute the intermediate derivatives of the network layers in a modular fashion using chain rule of differentiation. We refer back to Section 2.2 and Fig. 1, where the strain energy density is described as a composition of a kinematic layer $\mathcal{K}$ and inner network $\mathcal{N}$ Eq. (9). Accordingly, we obtain the following expressions:

$$\frac{\partial \Psi}{\partial F_{iJ}} = \frac{\partial \mathcal{N}}{\partial \mathcal{K}_m}\frac{\partial \mathcal{K}_m}{\partial F_{iJ}}, \qquad \frac{\partial^2 \Psi}{\partial F_{iJ}\partial F_{kL}} = \frac{\partial^2 \mathcal{N}}{\partial \mathcal{K}_m \partial \mathcal{K}_n}\frac{\partial \mathcal{K}_m}{\partial F_{iJ}}\frac{\partial \mathcal{K}_n}{\partial F_{kL}} + \frac{\partial \mathcal{N}}{\partial \mathcal{K}_m}\frac{\partial^2 \mathcal{K}_m}{\partial F_{iJ}\partial F_{kL}}. \tag{10}$$

Hence, we can implement the first and second derivatives for different kinematic layer types (principal strains, invariants, etc.) and inner network types (different architectures) individually and then combine them through Eq. (10) to obtain the associated computational graph for the gradients of the strain energy density, thus providing modularity.

We provide associated derivatives for various types of kinematic layers in Appendix A.2. As didactic illustrations, we present the analytical first and second derivatives for two notable hyperelastic NCMs found in the literature, MICNNs [9,21,58,59] and CANNs [5,8,38,60] in B. These derivations can be similarly extended for other NCMs.

3.3. Compatibility with distributed-memory parallelization for scalable simulations

Parallelization is the primary strategy for accelerating scientific computing applications. On shared-memory machines (workstations), where all CPUs have access to a common memory space, parallelization is typically achieved by employing multiple threads within a single process. This approach is effective as long as the entire problem fits within the memory of a single compute node.

However, computational requirements for large-scale simulations, such as finite element analyses on highly refined meshes, often exceed the memory and processing capacity of a single motherboard. In such cases, computations are performed on distributed-memory systems, or supercomputers, which consist of multiple compute nodes connected by high-speed interconnects. Each node has its own private memory, and processes on one node cannot directly access the memory of another. As a result, thread-based parallelization within a single process is insufficient. Instead, parallel execution requires launching at least one process per compute node, with data exchange between processes handled explicitly. This communication is conventionally managed through the Message Passing Interface (MPI) [53,77–79], a standardized and portable message-passing system that has become the de facto approach for distributed scientific computing.

Our proposed algorithms for vectorized assembly with NCMs are fully compatible with MPI-based distributed computing. As illustrated in Fig. 4, the computational mesh is partitioned into subdomains, each assigned to a specific MPI rank (i.e., process).

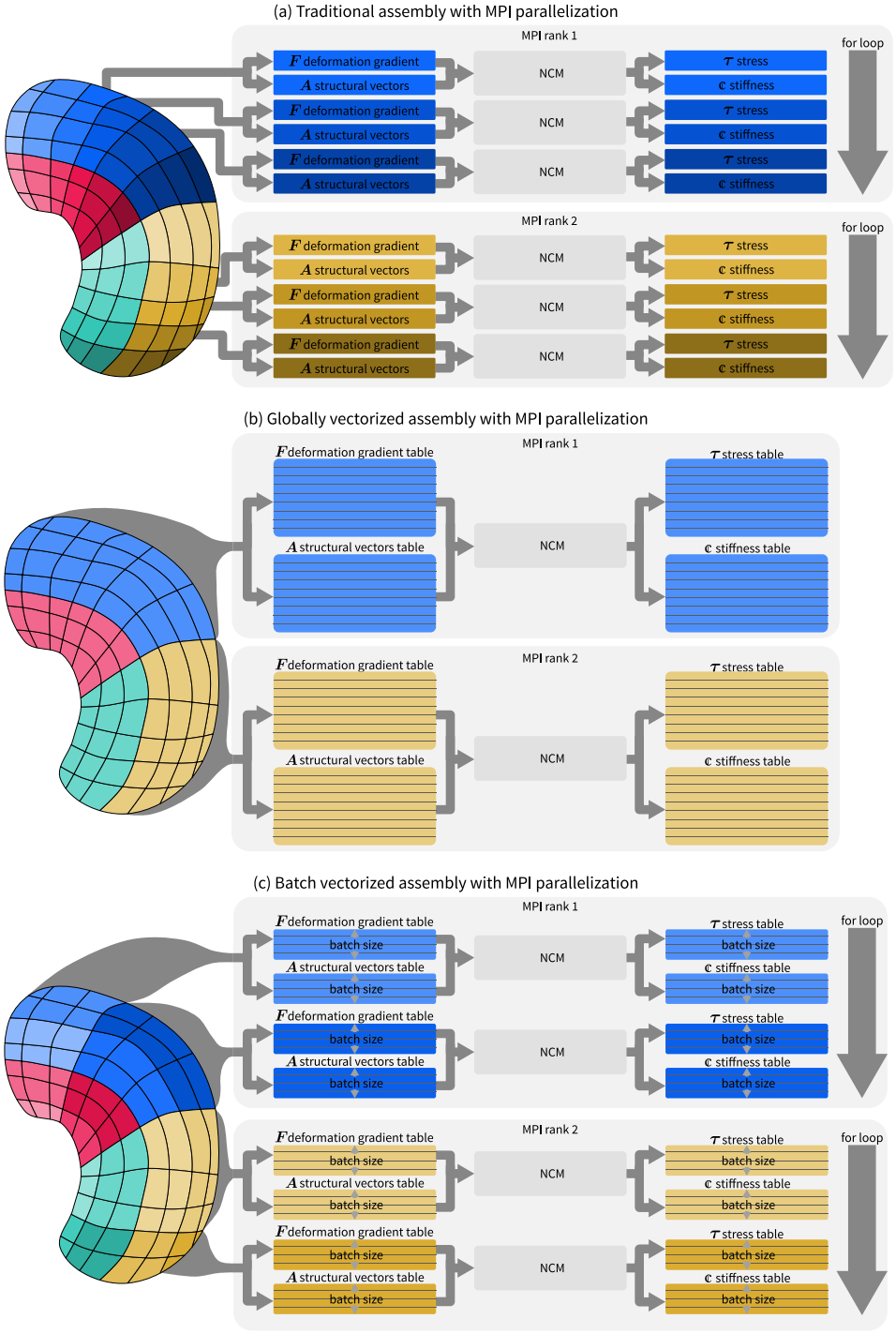

Fig. 4. Schematic comparison of constitutive update procedures in finite element assembly under MPI-based distributed parallelization: (a) traditional, (b) globally vectorized, and (c) batch-vectorized algorithms apply similarly to the single process case shown in Fig. 2. However, each MPI rank is only responsible for assembly on its associated subdomain of the mesh. Accordingly, for the globally vectorized algorithm (b), the table sizes correspond to the subdomain owned by the rank as opposed to the entire mesh.

The globally and batch-vectorized assembly algorithms are then executed independently on each rank for its local subdomain. While this strategy increases the number of kernel launches, these launches occur concurrently across ranks and therefore add little to the overall overhead.

4. Results

For the purposes of benchmarking, we investigate three NCM architectures for which architecture and implementation details were discussed Section 2.2: MICNNs [9,21,58,59], CANNs [5,8,38,60], and ICKANs [6,61]. These architectures are each trained on several (simulated) hyperelastic material data in an *unsupervised* manner – as presented in the NN-EUCLID framework of Thakolkaran et al. [6,21]. Specifically, the training data contains full-field displacements and global reaction forces for test coupons containing heterogeneous strain fields. The NCMs are then trained to satisfy the weak form of the linear momentum balance. For demonstration purposes and without loss of generality, we assume the ground-truth material model to be a Gent-Thomas hyperelastic material. Further details on the synthetic data generation and training of the NCMs are provided in Appendix C.

We emphasize that the framework is agnostic to the specific NCM architecture and, more importantly, to the source of the training data. Moreover, the accuracy of the NCMs is not the focus of this work; prior and contemporary studies (e.g., [3,80]) compare the performance of various NCMs and explore strategies and architectural enhancements aimed at improving NCM fidelity.

To investigate the effects of vectorization, batching, and CGO, computational experiments are first run only at the material point (i.e., quadrature point) level and in the absence of a FE assembler and solver to isolate the mechanisms for any potential speed-ups. We term these experiments *material point benchmarks* (Section 4.1). We then investigate these effects within the context of a FE solver (Section 4.2), followed by MPI scaling (Section 4.3) to verify that we can utilize these mechanisms to dramatically speed up the FE solver as a whole, while maintaining scalability on large-scale distributed computing.

4.1. Material point benchmarks

4.1.1. Material point benchmarks: Effect of vectorization and batching, without CGO

To evaluate the performance gains from vectorization and batching, we conduct computational experiments where constitutive updates are computed for a fixed number of material points, corresponding to quadrature points in the FE context. These updates are computed in batches of varying sizes, referred to as “batch size”. Each constitutive update includes computing the strain energy density, stress tensor, and stiffness tensor. In this benchmark, the stress and stiffness tensors are computed using AD, though these will later be replaced by CGO.

For each experiment, we report the absolute number of cache misses, cache misses relative to the non-vectorized baseline, absolute wall time, and speed-up in wall time compared to the non-vectorized baseline in Fig. 5.

Here, the non-vectorized baseline refers to the sequential evaluation of constitutive updates for each material point in a loop, which is equivalent to a batch-vectorized computation with a batch size of 1. All material point benchmarks are performed on a workstation equipped with an AMD Ryzen 9 7950X CPU, the details of which are presented in Table D.2 in Appendix D.

Increasing the batch size from 1 to approximately 10^3 reduces the number of cache misses by two orders of magnitude (Fig. 5(a,c)). The effect of reduction in cache misses is directly reflected in reduction of wall time and increase in **speed-up by two orders of magnitude** (Fig. 5(b,d)) over the non-vectorized baseline. The similarity between reduction in cache misses and wall times—i.e. the similarity between Fig. 5(a) and (b) and between Fig. 5(c) and (d)—are indicative of hardware level optimizations, such as prefetching, as discussed in Section 3.1. We emphasize that these results are obtained on a single processor, without the use of GPUs, multi-threading, or multiprocessing.

Moreover, we observe that there is an optimal batch size around $\sim 10^3 - 10^4$ for which the wall time per material point evaluation is minimized. In the context of an FE solver, this motivates for functionality allowing for control of the batch size, so that a user may choose the optimal batch size for a given machine. Note that the optimal batch size may vary across different machines.

The computational complexity for non-vectorized constitutive updates is simply $\mathcal{O}(n_{\text{pts}})$ where n_{pts} is the number of material points. To determine how batch size affects computational complexity, we perform computational experiments where we fix batch size and track wall time for increasing n_{pts} in Fig. 6.

The batch size reduces the wall time without affecting its linear scaling with respect to n_{pts}. Therefore, we estimate computational complexity of batch-vectorized constitutive updates as $\mathcal{O}(mn_{\text{pts}})$ where $m < 1$ is a factor that depends on the choice of batch size and computer architecture.

We note that cache misses, speed-ups, and scaling are comparable across diverse NCM architectures—namely, MICNNs [9,21,58,59], CANNs [5,8,38,60], and ICKANs [6,61]—supporting a reasonable conclusion that the observed performance gains are largely agnostic to the choice of NCM architecture. This suggests that, for accelerating FE simulations integrated with NCMs, implementation-level optimizations—such as vectorization—can have a greater impact on inference speed than architectural enhancements like pruning or the choice of NCM architecture itself. In light of this result and for the sake of brevity, we present the performance analyses from Section 4.1.2 onward using only the MICNN architecture, while noting that similar performance gains are observed for CANNs and ICKANs.

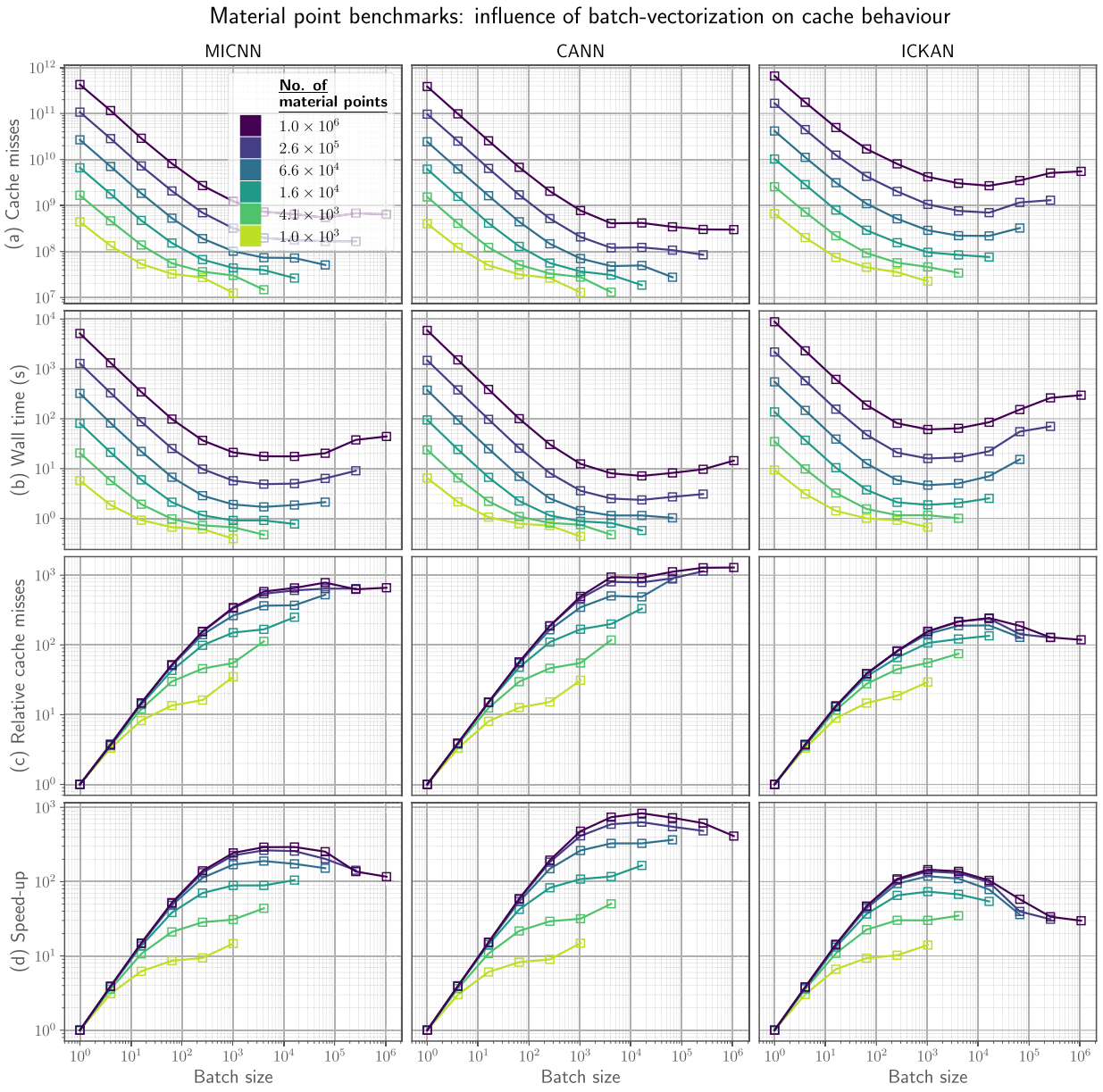

Fig. 5. Effect of batch size on cache efficiency and computational performance of NCMs. Results are shown for various fixed numbers of materials points (indicated in the legend). Metrics include (a) cache misses, (b) wall time, (c) relative cache misses (non-vectorized divided by vectorized), and (d) relative speed-up (non-vectorized wall time divided by vectorized wall time).

4.1.2. Material point benchmarks: Effect of compute graph optimization

As an alternative to AD, we use CGO (see Section 3.2) to obtain stress and stiffness tensors from a pre-trained NCM. To demonstrate the performance improvement due to CGO, the computational experiments at the material point-level presented in Section 4.1.1 and Fig. 5 are repeated for CGO implementations and compared against AD implementations in Fig. 7.

For fair comparison, the reverse-mode AD framework is implemented using LibTorch [81], which we assume to be highly optimized and state-of-the-art at the time of writing.

Combined batch-vectorization and CGO yields **speed-ups of nearly three orders of magnitude** over the equivalent non-vectorized non-CGO implementation (Fig. 7 (left)), e.g. 665 times faster at batch size of 2^{10}. To isolate the effect of CGO alone, we benchmark the speed-up for batch-vectorized and CGO relative to batch-vectorized and non-CGO implementation (Fig. 7 (right)) and observe speed-up between 2–10 times depending on the batch size. The remaining speed-up (up to 400 times depending on batch size) can be attributed to batch-vectorization – as shown in benchmark of batch-vectorized and non-CGO relative to non-batch-vectorized and non-CGO implementation (Fig. 7 (middle)).

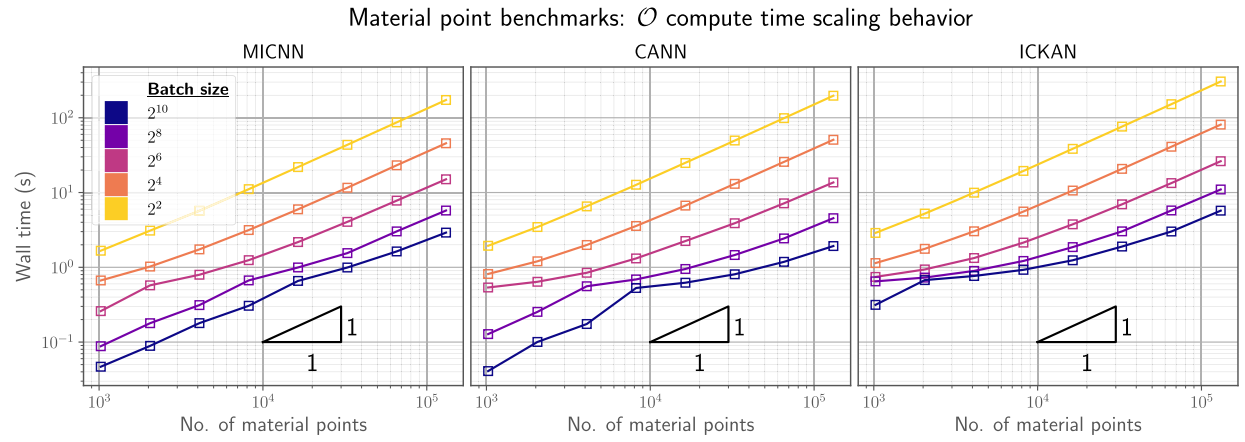

Fig. 6. Scaling of compute time for various batch sizes and NCMs. Wall time grows linearly with the number of material points. Increasing the batch size reduces the wall time but maintains the linear scaling with the number of material points.

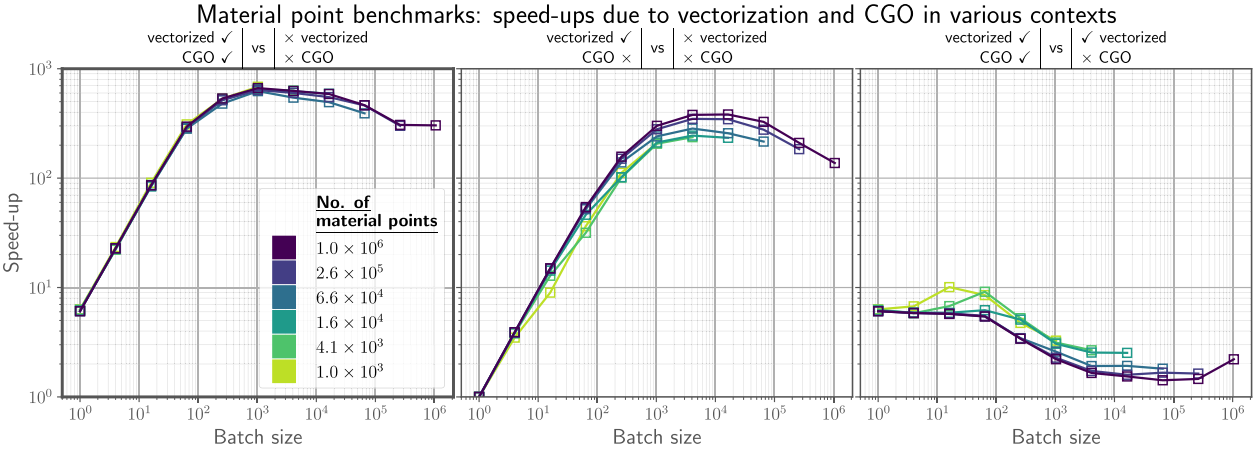

Fig. 7. Performance improvements from compute graph optimization (CGO): (left) combined speed-ups from CGO and vectorization relative to a non-vectorized, non-CGO baseline; (middle) speed-ups due to vectorization only (middle); and (right) Additional speed-ups due to CGO only at different batch sizes. Combining CGO and vectorization yields a speed-up of up to three orders of magnitude, whereas vectorization alone yields two orders of magnitude speed-up.

In addition to reducing wall time, CGO and batching allows for a reduction in RAM usage. To demonstrate this, we compare the RAM usage of batch-vectorized computations with and without CGO in Fig. 8.

The combination of batch-vectorization and CGO dramatically reduces RAM usage relative to global vectorization with non-CGO computations (Fig. 8 (left)). For example, the RAM usage is reduced by over 90%, when using batch-vectorization and CGO in comparison to globally vectorized non-CGO implementations for $\sim 10^6$ material points and batch sizes of less than 10^4 (Fig. 8 (left, middle)). Additionally, we note that the optimal batch size for speed-up is in the range of 10^3–10^4 (Fig. 7). CGO alone accounts for ~18–38 % of the reduction in RAM usage (Fig. 8 (right)). The majority of the reduction in RAM usage is typically due to batching (Fig. 8 (middle)), however, this largely depends on the total number of material points.

4.2. FE benchmarks

The material point benchmarks displayed thus far elucidate the effects of vectorization, batching, and CGO on the computational cost of constitutive updates of material points when using NCMs. We now translate this cost and speed-up in the context of FE simulations. We propose a benchmark simulation utilizing NCMs showcased in Fig. 9.

We simulate a unit cube, fixed at one end, and subjected to a unit axial tensile displacement and a half-rotation at the opposite end. We then proceed to apply vectorization, batching, and CGO, as detailed in Section 3, and systematically observe the effects on the computational cost. Specifically, we compare the cost of assembly with that of solving the resulting linear system, as these are the two most computationally intensive tasks in FE simulations.

For a fair assessment, the linear solver (as part of the nonlinear Newton-Raphson solver) used in these experiments is provided by PETSc [77], a highly mature, optimized, and state-of-the-art library (at the time of writing) for solving linear systems of equations.

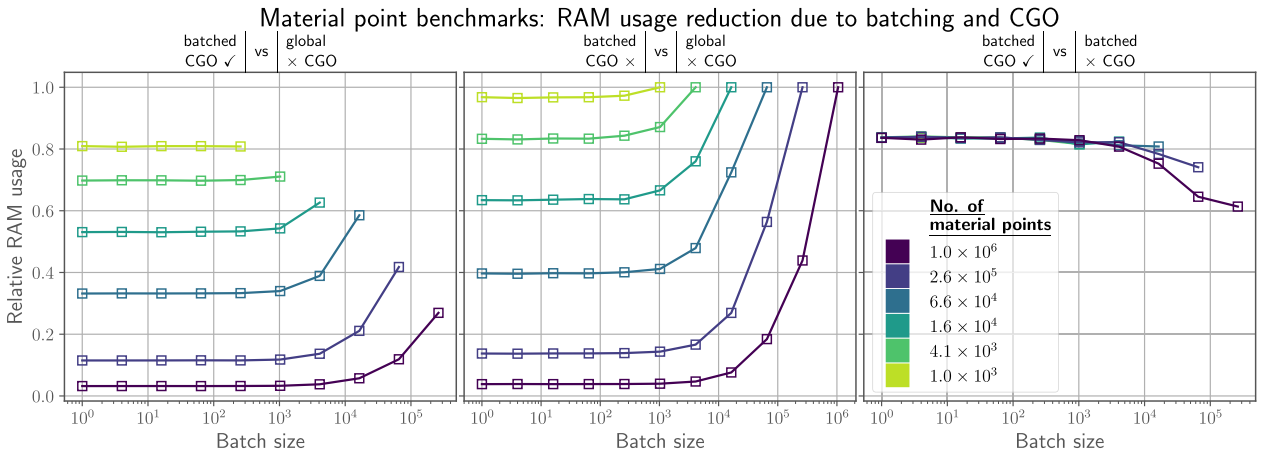

Fig. 8. Reduction of RAM usage by batch-vectorization and CGO: (left) RAM usage of batch-vectorized CGO computations relative to globally vectorized non-CGO computations; (middle) batch-vectorized non-CGO computations relative to globally vectorized non-CGO computations, and (right) batch-vectorized CGO computations relative to batch-vectorized non-CGO computations. Batch-vectorization and CGO reduce RAM usage by more than 90% at $\sim 10^6$ material points.

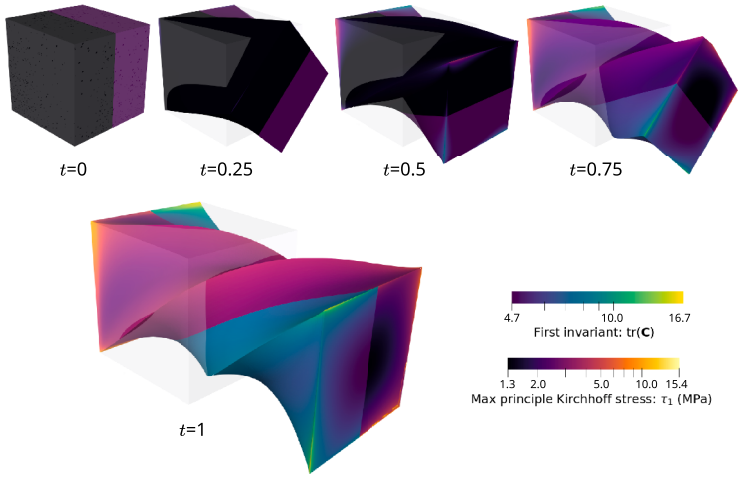

Fig. 9. Finite element benchmark problem for COMMET performance testing. A unit cube is fixed at one end and subjected to a unit displacement and half-rotation at the other. The reference configuration is shown in grey. Large values of tr($\boldsymbol{C}$) (greater than 16) demonstrate the robustness of the underlying FE solver.

Specifically, we use a conjugate gradient linear solver with Jacobi relaxation as the preconditioner. We emphasize that the proposed framework for accelerating NCMs in finite elements is agnostic to the choice of linear solver and preconditioner; the selections here are intended merely as representative examples.

4.2.1. FE benchmarks: Speed-up due vectorization, batching, and CGO

We benchmark the computational performance of the globally-vectorized (Algorithm 2) and batch-vectorized (Algorithm 3) assembly algorithms in Fig. 10, for both single-core and 16-core multiprocess (via MPI on a single node) simulations.

Combining batch-vectorization and CGO yields a three orders of magnitude speed-up in the constitutive update (Fig. 10 (a), left). The speed-ups are slightly diluted by the time taken up by other tasks in assembly (Fig. 10 (b), left), and then diluted further in the overall simulation time (Fig. 10 (c), left). However, the overall speed-up is still upwards of two orders of magnitude. (Fig. 10 (c), left).

The majority of the speed-up, i.e. two of the three orders of magnitude, results from assembly vectorization (Fig. 10 (a), middle). This is determined by repeating the computational experiments without the use of CGO and determining the speed-up relative to non-vectorized computations, thus isolating the effect of vectorization (Fig. 10 (a), middle). Comparing vectorized CGO computations with vectorized non-CGO computations (Fig. 10 (a), right) isolates the speed-ups due to CGO in different vectorization contexts, and shows that CGO accounts for a speed-up of 2–10 times depending on the batch size.

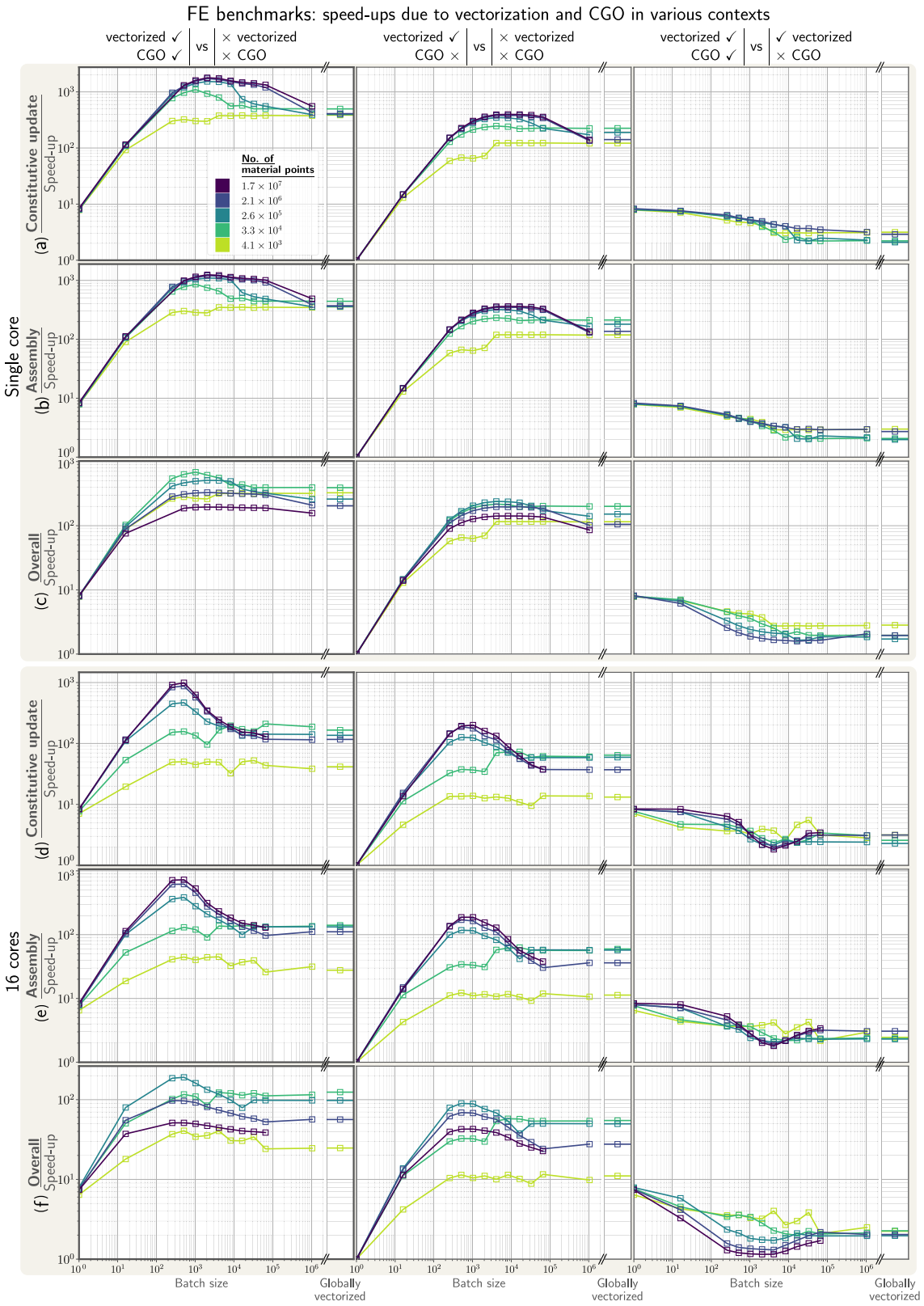

Fig. 10. Speed-up of FE simulations due to vectorization and CGO: (left column) the combined effects of vectorization and CGO, (middle column) the effects of vectorization only, and (right column) the effects of CGO only at different vectorization contexts are shown in the (top group) single core and (bottom group) multicore contexts. Additionally, the effects are shown for different sections of the program: (top) constitutive update, (middle) assembly, and (bottom) overall.

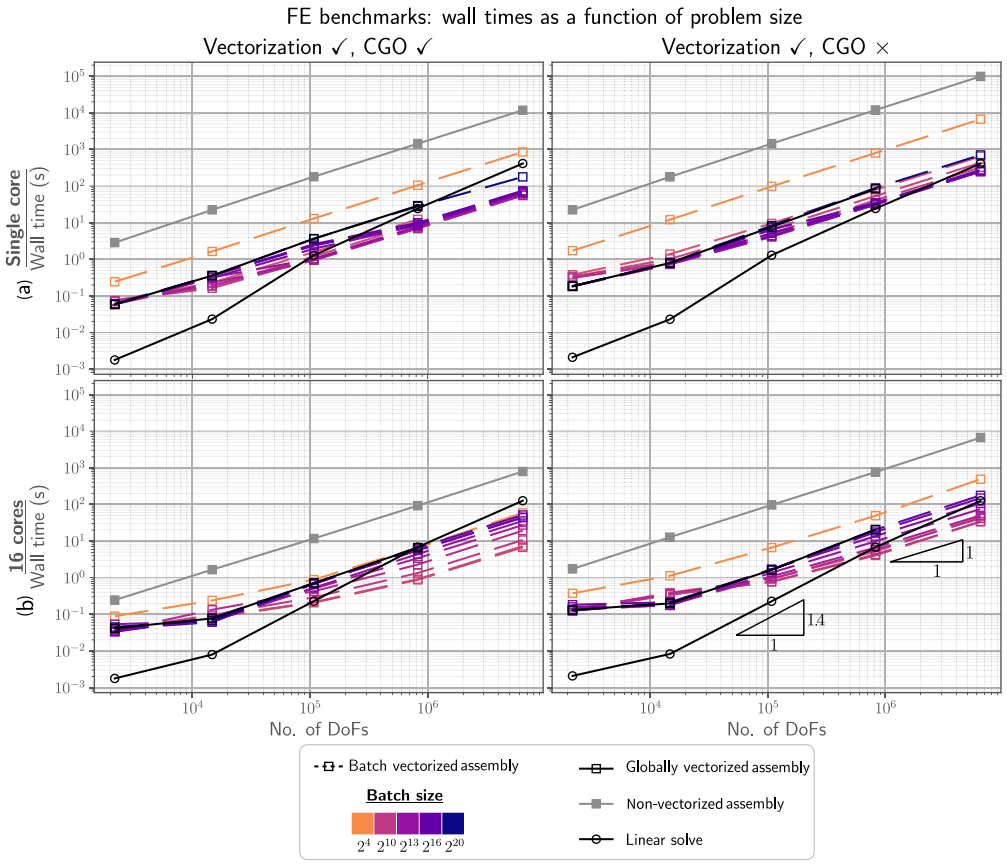

Fig. 11. Scaling of assembly and solver wall times with problem size: (left column) the combined effects of vectorization and CGO and (right column) vectorization without CGO are displayed in (a) the single core context and (b) multicore context. Assembly retains linear scaling with increasing degrees of freedom, while solver complexity grows superlinearly.

Batch-vectorization at the optimal batch size ($\sim 10^3$–10^4) consistently outperforms global-vectorization by a factor of 1–10 (Fig. 10 (a-c) left). This motivates the usage of the batch-vectorized algorithm for not only alleviating memory constraints, but also for the reduction of compute time.

Speed-up is maintained in the multiprocessing context (Fig. 10 (d-f)). However, the optimal batch size is shifted to be in the range of 10^2–10^3 in contrast to single core case i.e. 10^3–10^4. We suspect that this results from the cores sharing the L3 cache; since multiple processes are utilizing the cache, the batch size per process must be smaller for the data to fit in the cache than in the single core/process case.

The performance improvements due to vectorization, batching, and CGO are further contextualized in Fig. 11 by comparing the wall time for solving the linear system of equations with that of assembling the same linear system when using global-vectorization and batch-vectorization for various batch sizes as a function of the number of degrees of freedom (DoFs).

Linear (optimal) scaling of assembly wall time with the number of DoFs is maintained when assembly is globally- or batch-vectorized (Fig. 11 all subfigures). Furthermore, this scaling is maintained in the single core and multicore contexts, both for the CGO and non-CGO cases. By contrast, solving the linear system of equations using the conjugate gradient method with a Jacobi preconditioner has an empirically derived computational complexity of approximately $\mathcal{O}\left({n_{\text{dofs}}}^{1.4}\right)$, where n_{dofs} is the number of DoFs. Hence, with our vectorized FE assembly approach, there will be some problem size for which solving the linear system of equations becomes the dominant computational bottle-neck. However, in these experiments, even for problem sizes of 6,440,067 DoFs, which is larger than is used in many practical applications, the wall time for non-vectorized (batch size of 1) non-CGO assembly is more than 100 times larger than that of the solving the linear system of equations (Fig. 11 (a) right).

Batched-vectorization and CGO alleviates assembly as the computational bottle-neck for problems with more than 100,000 DoFs (Fig. 11 (a) left). Although vectorization does not alter the linear (ideal) scaling behavior of assembly, it reduces the assembly time by a machine- and batch size dependent coefficient (as discussed in Section 4.1.1 and Fig. 6), thus resulting in the transition of the computational bottle-neck from being assembly to being the solving of the linear system of equations at a much

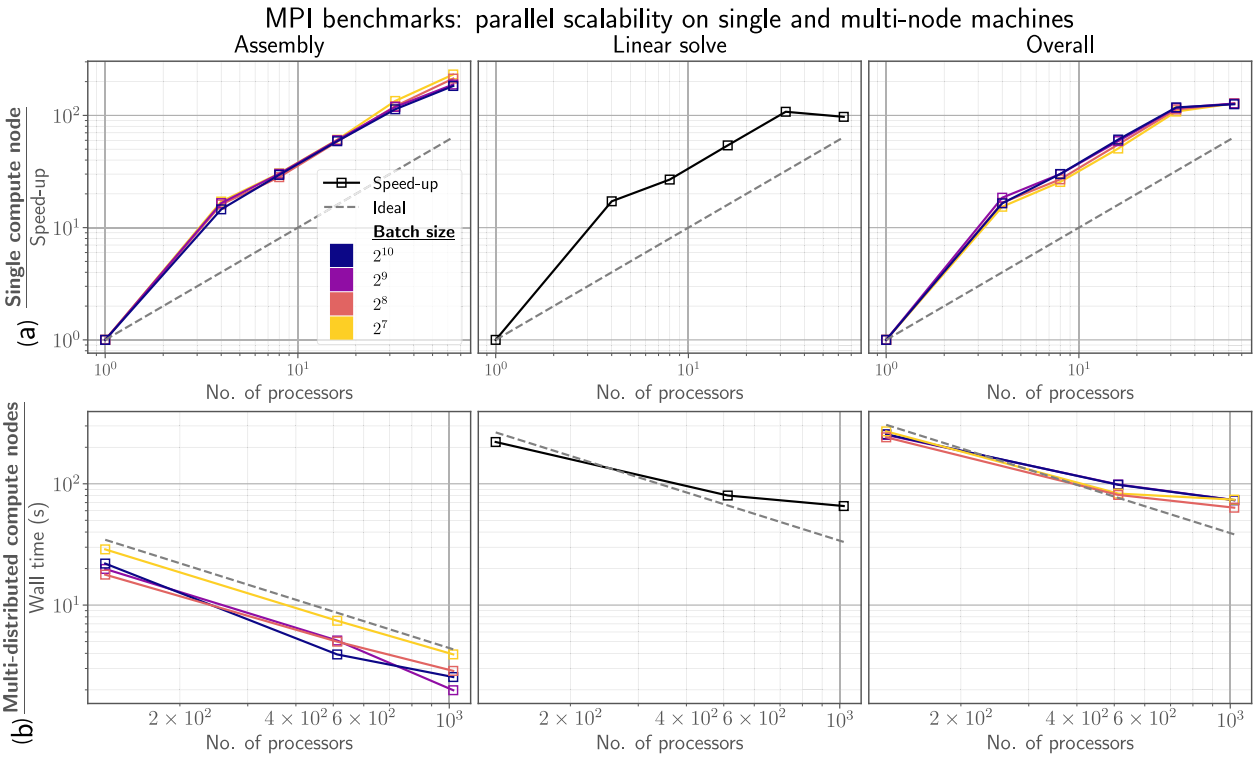

Fig. 12. Strong scaling behavior of our developed FE solver under MPI parallelization. For (left) assembly, (middle), linear solve, and (right) the overall simulation we report as a function of the number of processors used in the computation (a) the speed-up for a fixed problem size of 6,440,067 DoFs on a single compute node and (b) the wall time for a fixed problem size of 50,923,779 DoFs on a machine with distributed compute nodes. Moreover, values are reported for a range of batch sizes.

smaller problem size. Hence, to speed up the FE simulation overall further, it would be more effective to put effort into speeding up the linear solver as it is now the dominant computational bottle-neck for the majority of problem sizes of practical relevance.

4.3. Distributed memory parallelization benchmarks

To evaluate parallelization performance of the batch-vectorized assembly algorithm (Algorithm 3) and surrounding FE solver, we evaluate the *strong scaling* in Fig. 12, i.e. the speed-up of the wall time when using multiple processors relative to the single processor wall time, for both the single compute node and multi-compute nodes.[3]

In a single compute node, data exchange between multiple processors is faster because all processors share the same memory. In contrast, in a multi-node setup, data must be communicated over interconnects, which can introduce latency and slow down performance.

We focus only on the batch-vectorized case here since, for larger problems, the globally-vectorized algorithm introduces significant RAM requirements as discussed in Section 3.1. Moreover, we note that our results in Section 4.2.1 show that batch-vectorized assembly outperforms globally vectorized assembly with respect to compute time as well as RAM requirements.

The batch-vectorized assembly algorithm and FE solver displays super-linear strong scaling in the single node case (Fig. 12 (a)); i.e., the observed speed-up is greater than the ideal linear upper-bound proposed by Amdahl's law [83]. Super-linear scaling typically results from the fact that each CPU has its own L1 and L2 cache [84]. As more CPUs are made available to the program, a larger total amount of CPU cache is also made available ultimately reducing the amount of latency that is introduced due to main memory reads.

The batch-vectorized assembly algorithm and FE solver displays good strong scaling behavior on as many as 1024 cores on a multi-node setup (Fig. 12 (b)). In this case, we no longer observe super-linear scaling, which can be attributed to communication latency between multiple compute nodes via interconnects. We note, however, that this is not a feature of the program but rather the system on which the program is run. At the same time, **the scaling behavior of the batch-vectorized assembly algorithm outperforms that of the state-of-the-art linear solver implementation provided by PETSc** [77] (Fig. 12 (a) and (b), left and

[3] These computational experiments were run on the Delft Blue Supercomputer [82]. See Table D.2 in D for details of the CPUs used on these compute nodes.

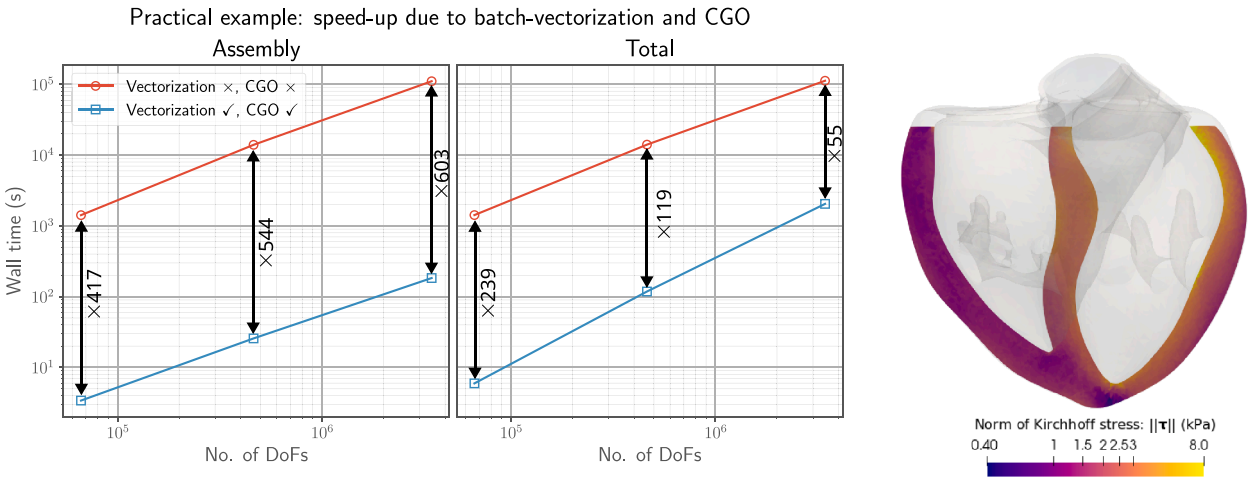

Fig. 13. COMMET functionality showcase. Using NCMs to simulate the diastolic filling of a patient-specific heart. We compute the end-diastolic deformation and stretch in the human heart, for which the myocardium was modelled using a MICNN-based NCM. The wall times for vectorized CGO implementation are compared to that of non-vectorized non-CGO implementations for assembly (left) and the simulation overall (middle) at three different mesh refinement levels. The resulting Kirchhoff stress distribution is shown on the right.

middle). This ensures that assembly remains at least comparable to, if not more scalable than, the linear solve and is therefore unlikely to become the scaling bottleneck in large-scale FE simulations.

5. COMMET: Demonstration of performance on large-scale practical FE simulations

COMMET is not merely an academic exercise to showcase the effects of vectorization, batching, and CGO on simple structured-mesh benchmarks. Instead, it delivers full finite element functionality for a wide range of solid mechanics problems, including support for unstructured meshes, three-dimensional field definitions, and diverse 3D boundary conditions.

To illustrate the capability of COMMET beyond canonical benchmarks, we perform a patient-specific simulation of human heart inflation under physiological loading conditions as an example problem. The geometry was reconstructed from high-resolution magnetic resonance images of a healthy 44-year-old male subject (178 cm, 70 kg) [85]. Diastolic filling was simulated by applying endocardial pressures of 8 mmHg and 4 mmHg in the left and right ventricles, respectively, representing physiologic end-diastolic states [86]. A pericardial constraint was imposed through Robin-type boundary conditions [87]. Myocardial tissue was modeled with a MICNN-based NCM, parameterized to reflect average biaxial stiffness of human myocardium [88].

Fig. 13 (right) depicts the resulting end-diastolic stress distributions.

To once again demonstrate the speed-up due batch-vectorization and CGO, the same problem was solved using a non-vectorized non-CGO implementation and batch-vectorized CGO (batch size of 512) implementation. Moreover, the problem was solved for three levels of mesh refinement resulting in problem sizes of 66 234, 462 474, and 3 436 362 degrees of freedom. Fig. 13 (left) and (right) show that batch-vectorization and CGO result in reducing the time for assembly by a factor 417–603 times and reducing the time for the total simulation by a factor of 239–55. Hence, it is clear that the computational gains of vectorization, batching, and CGO extend beyond synthetic benchmarks to complex, real-world geometries. Such efficiency enables high-fidelity simulations in solid mechanics at scales that were previously impractical, and creates opportunities for large-scale studies requiring repeated solves, parameter sweeps, or uncertainty quantification.

6. Conclusion

In this work, we have presented COMMET, a scalable and performant finite element solver designed to accelerate computationally intensive constitutive updates. Neural constitutive models (NCMs) represent an extreme but illustrative case, as their large computational graphs make repeated evaluations of stress and stiffness particularly costly, yet the same bottlenecks arise for many advanced material models in nonlinear solid mechanics. Our contributions are threefold: (i) globally and batch-vectorized assembly algorithms that restructure the traditional update loop to allow simultaneous evaluation of many material points, (ii) compute-graph-optimized derivatives that replace automatic differentiation and provide exact gradients at a fraction of the runtime and memory cost, and (iii) full compatibility with distributed-memory parallelism via MPI to ensure scalability across multiple compute nodes.

Extensive computational experiments demonstrated speed-ups exceeding three orders of magnitude in constitutive evaluations relative to traditional non-vectorized AD-based implementations, with roughly two orders of magnitude attributable to batch-vectorization and an additional 2–10× improvement from CGO depending on batch size. Batch-vectorization consistently outperformed global vectorization, exhibited an optimal batch size balancing cache efficiency with memory usage, and reduced RAM requirements compared to global vectorization. Parallel benchmarks showed superlinear scaling on single nodes and robust scaling

to thousands of cores across distributed nodes, ensuring that assembly no longer constitutes the limiting factor in large-scale FE analyses.

Although our demonstrations focused on NCMs, the framework is not restricted to them: the same strategies apply wherever loop-based constitutive updates dominate runtime, from sophisticated anisotropic plasticity to multiscale homogenization. COMMET therefore lays a strong foundation not only for the practical deployment of NCMs but also for accelerating high-fidelity FE simulations more broadly across solid mechanics. Future work will target automatic batch-size tuning, support for history-dependent materials, multiphysics extensions, and GPU implementation. Through the open-source release of COMMET, we invite the community to adopt, extend, and accelerate both neural and conventional constitutive models in computational mechanics.

CRediT authorship contribution statement

Benjamin Alheit: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Mathias Peirlinck:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization; **Siddhant Kumar:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Data availability

The COMMET codebase is publicly available at https://doi.org/10.5281/zenodo.17310683.

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.

Appendix A. Hyperelasticity formulations

A.1. Material and spatial stiffnesses

To allow for a generic interface in our code, we elect for defining the strain energy density Ψ as a function of the deformation gradient $\boldsymbol{F}$ instead of e.g. the left or right Cauchy-Green tenors, $\boldsymbol{C}$ or $\boldsymbol{B}$, respectively. However, taking first and second derivatives of Ψ with respect to $\boldsymbol{F}$ yields the non-symmetric first Piola-Kirchhoff stress $\boldsymbol{P}$ and associated fourth order stiffness tensor $\mathbb{C}^{\boldsymbol{P}}$, respectively,

$$P_{iJ} := \frac{\partial \Psi}{\partial F_{iJ}}, \qquad \mathbb{C}^{\boldsymbol{P}}_{iJkL} := \frac{\partial^2 \Psi}{\partial F_{iJ} \partial F_{kL}}. \tag{A.1}$$

The lack of symmetry in these tensors preclude the usage of Voigt notation representations in code which would allow for significantly more performant tensor operations, particular in the case of the fourth order stiffness tensor. Hence, to allow for the performance gains provided by Voigt notation, we transform these stress and stiffness tensors into the symmetric spatial counterparts. We use the well-known push-forward operation to obtain the symmetric Kirchhoff stress,

$$\tau_{ij} = \frac{\partial \Psi}{\partial F_{iJ}} F_{jJ}. \tag{A.2}$$

Obtaining the transformation for the stiffness tensor is less trivial. We start by noting

$$\mathbb{C}^{\boldsymbol{P}}_{iJkL} = \frac{\partial}{\partial F_{kL}} \left[F_{iI} S_{IJ} \right] = \delta_{ik} S_{JL} + F_{iI} \mathbb{C}_{IJKL} F_{kK}, \tag{A.3}$$

where $\boldsymbol{S} = \boldsymbol{F}^{-1}\boldsymbol{P}$ is the second Piola-Kirchhoff stress, $\mathbb{C} = 2\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{C}}$ is the material stiffness tensor, and we have used the identity

$$\frac{\partial \bullet}{\partial F_{kL}} = \frac{\partial \bullet}{\partial C_{IJ}} \left[\delta_{LI} F_{kJ} + F_{kI} \delta_{JL} \right]. \tag{A.4}$$

The spatial stiffness tensor is related to the material stiffness tensor by the well-known push forward operation

$$\mathbb{c}_{ijkl} = \mathbb{C}_{IJKL} F_{iI} F_{jJ} F_{kK} F_{lL}. \tag{A.5}$$

By rearranging Eq. (A.3) and substituting into Eq. (A.5) we obtain

$$\mathbb{c}_{ijkl} = F_{jJ} \left[\mathbb{C}^{\boldsymbol{P}}_{iJkL} - \delta_{ik} S_{JL} \right] F_{lL} = F_{jJ} \mathbb{C}^{\boldsymbol{P}}_{iJkL} F_{lL} - \delta_{ik} \tau_{jl}, \tag{A.6}$$

where we have used $\boldsymbol{\tau} = \boldsymbol{F}\boldsymbol{S}\boldsymbol{F}^T$. Hence, the necessary transformations for obtaining the Kirchhoff stress and spatial stiffness when defining Ψ in terms of $\boldsymbol{F}$ are

$$\tau_{ij} = \frac{\partial \Psi}{\partial F_{iJ}} F_{jJ}, \qquad \mathbb{c}_{ijkl} = F_{jJ} \frac{\partial^2 \Psi}{\partial F_{iJ} \partial F_{kL}} F_{lL} - \delta_{ik} \tau_{jl}. \tag{A.7}$$

A.2. Kinematic layers and derivatives for compute graph optimization

In most cases, the kinematic scalars used as inputs to the inner layer can be obtained from the right Cauchy-Green tensor. Hence, using the chain-rule as discussed in Section 3.2 and applying the relevant push forward operations yields the following expressions for the Kirchhoff stress and the spatial stiffness tensor:

$$\boldsymbol{\tau} = 2\sum_m \frac{\partial \mathcal{N}}{\partial \mathcal{K}_m} \underbrace{\frac{\partial \mathcal{K}_m}{\partial C_{IJ}} F_{iI} F_{jJ}}_{G^m_{ij}} \tag{A.8}$$

$$\mathfrak{c}_{ijkl} = 4\sum_{m,n} \frac{\partial^2 \mathcal{N}}{\partial \mathcal{K}_m \partial \mathcal{K}_m} \underbrace{\frac{\partial \mathcal{K}_m}{\partial C_{IJ}} F_{iI} F_{jJ}}_{G^m_{ij}} \underbrace{\frac{\partial \mathcal{K}_n}{\partial C_{KL}} F_{kK} F_{lL}}_{G^n_{kl}} + 4\sum_m \frac{\partial \mathcal{N}}{\partial \mathcal{K}_m} \underbrace{\frac{\partial^2 \mathcal{K}_m}{\partial C_{IJ} \partial C_{KL}} F_{iI} F_{jJ} F_{kK} F_{lL}}_{\mathbb{G}^m_{ijkl}}. \tag{A.9}$$

Here, we have grouped the derivatives of the kinematic layer along with the deformation gradients resulting from the push-forward operations and define these as

$$\boldsymbol{G}^m := \boldsymbol{F} \frac{\partial \mathcal{K}_m}{\partial \boldsymbol{C}} \boldsymbol{F}^T, \qquad \mathbb{G}^m_{ijkl} := \frac{\partial^2 \mathcal{K}_m}{\partial C_{IJ} \partial C_{KL}} F_{iI} F_{jJ} F_{kK} F_{lL}. \tag{A.10}$$

Hence, we can determine the tensors $\boldsymbol{G}^m$ and $\mathbb{G}^m$ for each kinematic scalar independently of the inner network used in the NCM. Once, the first and second derivatives of the inner network are known, they can be combined with the corresponding $\boldsymbol{G}^m$ and $\mathbb{G}^m$ to obtain the stress and stiffness tensors according to Eqs. (A.8) and (A.9), respectively. We elect for using the tensors defined in Eq. (A.10) as opposed to, say $\frac{\partial \mathcal{K}_m}{\partial \boldsymbol{F}}$ and $\frac{\partial^2 \mathcal{K}_m}{\partial C_{iJ} \partial F_{kL}}$, as they are symmetric by construction and conveniently allow for the use of Voigt notation. We now proceed to present expressions for this second- and fourth-order for the case of standard and isochoric invariants, while noting that this approach can be applied similarly to the case of principal stretches.

A.2.1. Invariants

The standard invariants, and the corresponding second and fourth order tensors as defined in A.10 are given by

$$I_1 = \mathrm{tr}(\boldsymbol{C}), \qquad \boldsymbol{G}^1 := \boldsymbol{B}, \qquad \mathbb{G}^1 := \mathbb{O}, \tag{A.11}$$

$$I_2 = \frac{1}{2}\left[\mathrm{tr}(\boldsymbol{C})^2 - \mathrm{tr}\left(\boldsymbol{C}^2\right)\right], \qquad \boldsymbol{G}^2 := \boldsymbol{B}\mathrm{tr}(\boldsymbol{B}) - \boldsymbol{B}^2, \qquad \mathbb{G}^2 := \boldsymbol{B} \otimes \boldsymbol{B} - \boldsymbol{B} \underline{\overline{\otimes}} \boldsymbol{B}, \tag{A.12}$$

$$I_3 = \det \boldsymbol{C}, \qquad \boldsymbol{G}^3 := \det \boldsymbol{B}\, \boldsymbol{I}, \qquad \mathbb{G}^3 := \det \boldsymbol{B} \left[\boldsymbol{I} \otimes \boldsymbol{I} - \boldsymbol{I} \underline{\overline{\otimes}} \boldsymbol{I}\right], \tag{A.13}$$

$$I_{4,ij} = \boldsymbol{A}^i \cdot \boldsymbol{C} \boldsymbol{A}^j, \qquad \boldsymbol{G}^{4,ij} := \mathrm{sym}\left(\boldsymbol{a}^i \otimes \boldsymbol{a}^j\right), \qquad \mathbb{G}^{4,ij} := \mathbb{O}, \tag{A.14}$$

$$I_{5,ij} = \boldsymbol{A}^i \cdot \boldsymbol{C}^2 \boldsymbol{A}^j, \qquad \boldsymbol{G}^{5,ij} := 2\mathrm{sym}\left(\boldsymbol{a}^i \otimes \boldsymbol{B}\boldsymbol{a}^j\right), \qquad \mathbb{G}^{5,ij} := \boldsymbol{B} \underline{\overline{\otimes}} \mathrm{sym}\left(\boldsymbol{a}^i \otimes \boldsymbol{a}^j\right) + \mathrm{sym}\left(\boldsymbol{a}^i \otimes \boldsymbol{a}^j\right) \underline{\overline{\otimes}} \boldsymbol{B}. \tag{A.15}$$

Here, $\boldsymbol{A}^i$ is the i^{th} structural vector, $\boldsymbol{a}^i := \boldsymbol{F}\boldsymbol{A}^i$ is the current configuration counterpart of $\boldsymbol{A}^i$, $\mathrm{sym}(\bullet) := \frac{1}{2}\left[\bullet + \bullet^T\right]$ is the symmetric part of a tensor, and the various tensor products are defined as follows:

$$[\boldsymbol{a} \otimes \boldsymbol{b}]\boldsymbol{c} = a_i b_j c_j \tag{A.16}$$

$$\boldsymbol{A} \otimes \boldsymbol{B} = A_{ij} B_{kl} \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k \otimes \boldsymbol{e}_l, \tag{A.17}$$

$$\boldsymbol{A} \underline{\overline{\otimes}} \boldsymbol{B} = \frac{1}{2}\left[A_{ik} B_{jl} + A_{il} B_{kj}\right] \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k \otimes \boldsymbol{e}_l, \tag{A.18}$$

where $\boldsymbol{e}$ denotes a basis vector.

A.2.2. Isochoric invariants

Many hyperelastic materials exhibit behavior that is far stiffer in volumetric deformation than in isochoric (volume-preserving) deformation. For this reason, it is common to multiplicatively decompose the deformation gradient into an isochoric $\tilde{\boldsymbol{F}}$ and volumetric part $\bar{\boldsymbol{F}}$; that is,

$$\boldsymbol{F} = \tilde{\boldsymbol{F}}\bar{\boldsymbol{F}}, \qquad \tilde{\boldsymbol{F}} := J^{-1/3}\boldsymbol{F}, \qquad \bar{\boldsymbol{F}} := J^{1/3}\boldsymbol{I}, \qquad J = \det \boldsymbol{F}. \tag{A.19}$$

Here, J is the (volumetric) Jacobian and $\tilde{\bullet}$ and $\bar{\bullet}$ denote the isochoric and volumetric parts of $\bullet$, respectively. The strain energy density function is then postulated in terms of the isochoric invariants,

$$\tilde{I}_m := I_m I_3^{\alpha_m} \qquad \text{for } m \in \tilde{\mathcal{I}}, \tag{A.20}$$

$$\tilde{\mathcal{I}} := \left\{1, 2, (4, ij), (5, ij), \,|\, i, j \in \left[1, n_{\mathrm{sv}}\right]\right\}, \tag{A.21}$$

$$\alpha_m := \begin{cases} -2/3 & \text{if } m \in \left\{2, (5, ij) \,|\, i, j \in \left[1, n_{\mathrm{sv}}\right]\right\}, \\ -1/3 & \text{otherwise.} \end{cases} \tag{A.22}$$

The corresponding second- and fourth-order tensors defined in Eq. (A.10) are then given by

$$\boldsymbol{G}^m = \tilde{\boldsymbol{G}}^m + \alpha_m \tilde{I}_m \boldsymbol{I}, \tag{A.23}$$

$$\mathbb{G}^m = \tilde{\mathbb{G}}^m + \alpha_m \Big[\boldsymbol{I} \otimes \tilde{\boldsymbol{G}}^m + \tilde{\boldsymbol{G}}^m \otimes \boldsymbol{I} + \tilde{I}_m \Big[\alpha_m \boldsymbol{I} \otimes \boldsymbol{I} - \boldsymbol{I} \overline{\underline{\otimes}} \boldsymbol{I} \Big] \Big], \tag{A.24}$$

where $\tilde{\boldsymbol{G}}^m$ and $\tilde{\mathbb{G}}^m$ are the isochoric versions of the corresponding terms in Eqs. (A.15)–(A.15), i.e.

$$\tilde{\boldsymbol{G}}^1 := \tilde{\boldsymbol{B}}, \qquad \tilde{\mathbb{G}}^1 := \mathbb{O}, \tag{A.25}$$

$$\tilde{\boldsymbol{G}}^2 := \tilde{\boldsymbol{B}} \mathrm{tr}(\tilde{\boldsymbol{B}}) - \tilde{\boldsymbol{B}}^2, \qquad \tilde{\mathbb{G}}^2 := \tilde{\boldsymbol{B}} \otimes \tilde{\boldsymbol{B}} - \tilde{\boldsymbol{B}} \overline{\underline{\otimes}} \tilde{\boldsymbol{B}}, \tag{A.26}$$

$$\tilde{\boldsymbol{G}}^{4,ij} := \mathrm{sym}\big(\tilde{\boldsymbol{a}}^i \otimes \tilde{\boldsymbol{a}}^j\big), \qquad \tilde{\mathbb{G}}^{4,ij} := \mathbb{O}, \tag{A.27}$$

$$\tilde{\boldsymbol{G}}^{5,ij} := 2\mathrm{sym}\big(\tilde{\boldsymbol{a}}^i \otimes \boldsymbol{B}\tilde{\boldsymbol{a}}^j\big), \qquad \tilde{\mathbb{G}}^{5,ij} := \tilde{\boldsymbol{B}} \overline{\underline{\otimes}} \mathrm{sym}\big(\tilde{\boldsymbol{a}}^i \otimes \tilde{\boldsymbol{a}}^j\big) + \mathrm{sym}\big(\tilde{\boldsymbol{a}}^i \otimes \tilde{\boldsymbol{a}}^j\big) \overline{\underline{\otimes}} \tilde{\boldsymbol{B}}. \tag{A.28}$$

At the same time, strain energy due to volumetric changes are modelled using J, for which the corresponding second- and fourth-order tensors are

$$\boldsymbol{G} = \frac{J}{2} \boldsymbol{I}, \qquad \mathbb{G} = \frac{J}{4} \Big[I \otimes \boldsymbol{I} - 2 \boldsymbol{I} \overline{\underline{\otimes}} \boldsymbol{I} \Big]. \tag{A.29}$$

Appendix B. Inner neural constitutive networks

Here we briefly present the architectures for several NCMs from literature including CANNs [5,8,38,60], MICNNs [9,21,58,59], and ICKANs [6,61]. The presentations here are kept brief and readers are referred to the original publications for detailed treatments. Additionally, we provide analytical expressions for the first and second derivatives of CANNs and MICNNs as didactic examples for usage in CGO. These expressions can be similarly derived for ICKANs and other NCM inner networks.

B.1. Constitutive artificial neural networks (CANNs)

CANNs [5,8,38,60] have a tree-like architecture that is expressed mathematically as

$$\mathcal{N}(\boldsymbol{\mathcal{K}}) = f_2 \circ f_1 \circ f_0(\boldsymbol{\mathcal{K}}) = \sum_{m \in \mathcal{I}} \sum_{k=1}^{n} w_{2,k,m} f_2\big(f_1\big(f_0\big(\mathcal{K}_m; w_{0,k,m}\big); w_{1,k,m}\big)\big), \tag{B.1}$$

where, $\mathcal{I}$ is an enumeration of the kinematic scalars used as input to the network, $w_{i,k,m}$ $i = 1, 2, 3$, $k = 1, \ldots, n$ $k \in \mathcal{I}$ are trainable weights, and

$$f_0 = \begin{cases} (\circ) \\ \langle \circ \rangle \\ |\circ| \\ \vdots \end{cases} \qquad f_1 = \begin{cases} (\circ)^1 \\ (\circ)^2 \\ (\circ)^3 \\ \vdots \end{cases} \qquad f_2 = \begin{cases} w_1(\circ) \\ \exp w_1(\circ) - 1 \\ -\ln\big(1 - w_1(\circ)\big) \\ \vdots \end{cases}. \tag{B.2}$$

Following [38], we obtain the first and second derivatives of $\mathcal{N}$ using the chain-rule; these are,

$$\frac{\partial \mathcal{N}}{\partial \mathcal{K}_m} = \sum_{k=1}^{n} w_{2,k,m} \frac{\partial f_2}{\partial \circ} \frac{\partial f_1}{\partial \circ} \frac{\partial f_0}{\partial \mathcal{K}_m}, \tag{B.3}$$

$$\frac{\partial^2 \mathcal{N}}{\partial \mathcal{K}_m \partial \mathcal{K}_m} = \sum_{k=1}^{n} w_{2,k,m} \left[\left[\frac{\partial^2 f_2}{\partial \circ \partial \circ} \left[\frac{\partial f_1}{\partial \circ} \right]^2 + \frac{\partial f_2}{\partial \circ} \frac{\partial^2 f_1}{\partial \circ \partial \circ} \right] \left[\frac{\partial f_0}{\partial \mathcal{K}_m} \right]^2 + \frac{\partial f_2}{\partial \circ} \frac{\partial f_1}{\partial \circ} \frac{\partial^2 f_0}{\partial \mathcal{K}_m \partial \mathcal{K}_m} \right]. \tag{B.4}$$

Note that, due to the form of Eq. (B.1), $\frac{\partial^2 \mathcal{N}}{\partial \mathcal{K}_m \partial \mathcal{K}_n} = 0, \quad \forall m \neq n$. In order to evaluate the derivatives in Eq. (B.4) the first and second derivatives of the expressions in Eq. (B.2) are required; these are,

$$\frac{\partial f_0}{\partial \circ} = \begin{cases} 1 \\ \frac{1}{2}(1 + \mathrm{sgn}(\circ)) \\ \mathrm{sgn}(\circ) \\ \vdots \end{cases} \qquad \frac{\partial f_1}{\partial \circ} = \begin{cases} 1 \\ 2(\circ)^1 \\ 3(\circ)^2 \\ \vdots \end{cases} \qquad \frac{\partial f_2}{\partial \circ} = \begin{cases} w_1 \\ w_1 \exp w_1(\circ) \\ \frac{w_1}{1 - w_1(\circ)} \\ \vdots \end{cases} \tag{B.5}$$

$$\frac{\partial^2 f_0}{\partial \circ \partial \circ} = \begin{cases} 0 \\ 0 \\ 0 \\ \vdots \end{cases} \qquad \frac{\partial^2 f_1}{\partial \circ \partial \circ} = \begin{cases} 0 \\ 2 \\ 6(\circ)^1 \\ \vdots \end{cases} \qquad \frac{\partial^2 f_2}{\partial \circ \partial \circ} = \begin{cases} 0 \\ w_1^2 \exp w_1(\circ) \\ -\frac{w_1^2}{\left[1 - w_1(\circ)\right]^2} \\ \vdots \end{cases} \tag{B.6}$$

B.2. Monotonic input convex neural networks (MICNNs)

In short, input convext neural networks (ICNNs) [9,58] are described by the following equations:

$$\boldsymbol{z}^{(0)} = \boldsymbol{\mathcal{K}}, \tag{B.7a}$$

$$\boldsymbol{y}^{(k)} = \boldsymbol{A}^{(k)}\boldsymbol{z}^{(k-1)} + \boldsymbol{B}^{(k)}\boldsymbol{z}^{(0)} + \boldsymbol{c}^{(k)}, \tag{B.7b}$$

$$\boldsymbol{z}^{(k)} = \mathcal{F}\left(\boldsymbol{y}^{(k)}\right), \tag{B.7c}$$

$$\mathcal{N} = \boldsymbol{A}^{(n)}\boldsymbol{z}^{(n-1)} + \boldsymbol{B}^{(n)}\boldsymbol{z}^{(0)}. \tag{B.7d}$$

Here, $\boldsymbol{\mathcal{K}}$ is the input to the network, $\boldsymbol{z}^{(n)}$ is the output of the network, $\mathcal{F}$ is an activation function that is applied elementwise, $\boldsymbol{c}^{(k)}$ are learnable bias vectors, and $\boldsymbol{A}^{(k)}$ and $\boldsymbol{B}^{(k)}$ are learnable weight matrices. Additionally, Eqs. (B.7b) and (B.7c) are applied iteratively for $k = 1, \dots, n-1$; that is, for each hidden layer in the network. Convexity of Eq. (B.7) in $\boldsymbol{\mathcal{K}}$ is gauranteed if all values in $\boldsymbol{A}^{(k)}$ are non-negative for $k > 0$ and $\mathcal{F}$ is convex and monotonically non-decreasing. Furthermore, convexity and non-decreasing monotonicity of Eq. (B.7) in $\boldsymbol{\mathcal{K}}$ is gauranteed if all values in $\boldsymbol{A}^{(k)}$ and $\boldsymbol{B}^{(k)}$ are non-negative and $\mathcal{F}$ is convex and monotonically non-decreasing.

The relative first derivatives of Eqs. (B.7a)–(B.7d), determined via use of the chain-rule, are as follows:

$$\frac{\partial \mathcal{N}}{\partial \mathcal{K}_m} = A_j^{(n)} \frac{\partial z_j^{(n-1)}}{\partial \mathcal{K}_m} + B_m^{(n)} \tag{B.8}$$

$$\text{(no sum on j)}\quad \frac{\partial z_j^{(n-1)}}{\partial \mathcal{K}_m} = \frac{\partial \mathcal{F}}{\partial y_j^{(n-1)}} \frac{\partial y_j^{(n-1)}}{\partial \mathcal{K}_m} \tag{B.9}$$

$$\frac{\partial y_j^{(n-1)}}{\partial \mathcal{K}_m} = A_{jk}^{(n-1)} \frac{\partial z_k^{(n-2)}}{\partial \mathcal{K}_m} + B_{jm}^{(n-1)} \tag{B.10}$$

Note that the expression for $\frac{\partial z_j^{(n-2)}}{\partial \mathcal{K}_m}$ will be identical to that in Eq. (B.9), however "$n-1$" will be replaced with "$n-2$". Hence, Eqs. (B.9) and (B.10) can be applied recursively from $k = n$ to $k = 1$, at which point the necessary derivatives are given by

$$\text{(no sum on j)}\quad \frac{\partial z_j^{(1)}}{\partial \mathcal{K}_m} = \frac{\partial \mathcal{F}}{\partial y_j^{(1)}} \frac{\partial y_j^{(1)}}{\partial \mathcal{K}_m}, \tag{B.11}$$

$$\frac{\partial y_j^{(1)}}{\partial \mathcal{K}_m} = A_{jm}^{(1)} + B_{jm}^{(1)}. \tag{B.12}$$

The second derivatives of Eqs. (B.7a)–(B.7d), determined via use of the chain-rule on Eqs. (B.8)–(B.10), are as follows:

$$\frac{\partial^2 \mathcal{N}}{\partial K_m \partial K_n} = A_j^{(n)} \frac{\partial^2 z_j^{(n-1)}}{\partial K_m \partial K_n} \tag{B.13}$$

$$\text{(no sum on j)}\quad \frac{\partial^2 z_j^{(n-1)}}{\partial K_m \partial K_n} = \frac{\partial \mathcal{F}}{\partial y_j^{(n-1)}} \frac{\partial^2 y_j^{(n-1)}}{\partial K_m \partial K_n} + \frac{\partial^2 \mathcal{F}}{\partial y_j^{(n-1)} \partial y_j^{(n-1)}} \frac{\partial y_j^{(n-1)}}{\partial K_m} \frac{\partial y_j^{(n-1)}}{\partial K_n} \tag{B.14}$$

$$\frac{\partial^2 y_j^{(n-1)}}{\partial K_m \partial K_n} = A_{jk}^{(n-1)} \frac{\partial^2 z_k^{(n-2)}}{\partial K_m \partial K_n}. \tag{B.15}$$

Note that the recursive logic applies to Eqs. (B.14) and (B.15) in a similar manner to that applied to Eqs. (B.9) and (B.10). Hence, the (M)ICNN, along with its first and second derivatives can be evaluated in one pass, without the use of automatic differentiation, as detailed in Algorithm 4. There, $\odot$ denotes the Hadamard product.

B.3. Input-convex Kolmogorov-Arnold networks (ICKANs)

We first briefly introduce Kolmogorov-Arnold networks (KANs) [61] and we then discuss how this architecture is altered to ensure input-convexity in-line with [6]. We note again that, the presentation brief and only included for completeness – readers are referred to [6,61] for more detailed treatments. The architecture for KAN of R layers is defined as follows:

$$\boldsymbol{z}^{(0)} = \boldsymbol{\mathcal{K}}, \tag{B.16}$$

$$\boldsymbol{z}^{(r)} = \left[\sum_{j=1}^{n_{r-1}} s_{r-1,1,j}\left(z_j^{(r-1)}\right),\ \dots,\ \sum_{j=1}^{n_{r-1}} s_{r-1,n_r,j}\left(z_j^{(r-1)}\right)\right]^T, \tag{B.17}$$

$$\mathcal{N} = \sum_{j=1}^{n_{R-1}} s_{R-1,1,j}\left(z_j^{(R-1)}\right). \tag{B.18}$$

Here, $s_{i,j,k}$ are weighted trainable univariate splines, i.e.

$$s(x) = w_s \psi(x) \tag{B.19}$$

Algorithm 4 Evaluating an M(ICNN) along with its first and second derivatives with one pass.

$\boldsymbol{z} \leftarrow \boldsymbol{\mathcal{K}}$
$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}}} \leftarrow \boldsymbol{I}$
$\frac{\partial^2 \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}} \leftarrow \mathbb{O}$
for $k = 1, \dots, n-1$ **do**
  $\boldsymbol{y} \leftarrow \boldsymbol{A}^{(k)} \boldsymbol{z} + \boldsymbol{B}^{(k)} \boldsymbol{\mathcal{K}} + \boldsymbol{c}^{(k)}$
  $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\mathcal{K}}} \leftarrow \boldsymbol{A}^{(k)} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}}} + \boldsymbol{B}$
  $\frac{\partial^2 \boldsymbol{y}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}} \leftarrow \boldsymbol{A}^{(k)} \frac{\partial^2 \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}}$
  $\frac{\partial^2 \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}} \leftarrow \frac{\partial \mathcal{F}}{\partial \boldsymbol{y}} \odot \frac{\partial^2 \boldsymbol{y}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}} + \frac{\partial^2 \mathcal{F}}{\partial \boldsymbol{y} \partial \boldsymbol{y}} \odot \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\mathcal{K}}} \otimes \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\mathcal{K}}}$
  $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}}} \leftarrow \frac{\partial \mathcal{F}}{\partial \boldsymbol{y}} \odot \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{\mathcal{K}}}$
  $\boldsymbol{z} \leftarrow \mathcal{F}(\boldsymbol{y})$
end for
$\mathcal{N} \leftarrow \boldsymbol{A}^{(n)} \boldsymbol{z} + \boldsymbol{B}^{(n)} \boldsymbol{\mathcal{K}}$
$\frac{\partial \mathcal{N}}{\partial \boldsymbol{\mathcal{K}}} \leftarrow \boldsymbol{A}^{(n)} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}}} + \boldsymbol{B}^{(n)}$
$\frac{\partial^2 \mathcal{N}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}} \leftarrow \boldsymbol{A}^{(n)} \frac{\partial^2 \boldsymbol{z}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}}$
Output: $\mathcal{N}$, $\frac{\partial \mathcal{N}}{\partial \boldsymbol{\mathcal{K}}}$, $\frac{\partial^2 \mathcal{N}}{\partial \boldsymbol{\mathcal{K}} \partial \boldsymbol{\mathcal{K}}}$

where w_s is a trainable weight and ψ is k^{th}-order a B-spline consisting of n_b basis functions $B_{i,k}$ with control points c_i, i.e.

$$\psi(x) = \sum_{i=1}^{n_b} c_i B_{i,k}(x), \qquad \text{with} \quad \sum_{i=1}^{n_b} B_{i,k}(x) = 1 \quad \text{for} \quad x \in [x_{\min}, x_{\max}]. \tag{B.20}$$

To define the k^{th}-order B-spline basis functions, we consider a set of $m_b = (k + n_b + 1)$ knots $\{t_i\}_{i=1}^{m_b}$ and apply De Boor's recursive algorithm [89] as follows:

Zero-order basis function ($k = 0$):

$$B_{i,0}(x) = \begin{cases} 1, & \text{if } t_i \le x < t_{i+1}, \\ 0, & \text{otherwise.} \end{cases} \tag{B.21}$$

Recursive definition for higher orders ($k > 0$):

$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x). \tag{B.22}$$

For the special case of a uniform B-spline, the knots are equally spaced, i.e.,

$$t_{i+2} - t_{i+1} = t_{i+1} - t_i, \quad \forall\, i \in [1, m_b - 2]. \tag{B.23}$$

For a KAN to be input-convex, i.e. for a KAN to be an ICKAN, we require that the weights w_s are positive and that the splines are convex and monotonically non-decreasing [6]. This is satisfied so long as the control points satisfy the following condition:

$$c_{i+2} - c_{i+1} \ge c_{i+1} - c_i \ge 0, \quad \forall\, i \in [1, n_b - 2]. \tag{B.24}$$

Appendix C. Data generation and NCM training

When training all NCMs used in this work, we follow the EUCLID paradigm for unsupervised discovery of material behavior. More specifically, we use the NN-EUCLID framework of Thakolkaran et al. [6,21]. In brief, this allows for the training of NCMs using full-field displacements and global reaction forces, both of which are physically obtainable from real experiments by using a combination of digital image correlation (DIC) and a load cell, i.e. stress measurements are not required. Given the known displacements and reaction forces $R^{\beta,t}$ on $\beta = 1, \dots, n_\beta$ constrained boundaries at $t = 1, \dots, n_t$ time steps, the parameters Q for a given NCM are obtained using

$$Q = \arg\min_Q \sum_{t=1}^{n_t} \left[\sum_{(I,i) \in D^{\text{free}}} \left(r_i^{I,t} \right)^2 + \sum_{\beta=1}^{n_\beta} \left(R^{\beta,t} - \sum_{(I,i) \in D_\beta^{\text{fix}}} r_i^{I,t} \right)^2 \right] \tag{C.1}$$

where D^{free} is the set of tuples of nodes that are unconstrained in given direction and D_β^{fix} is the set of nodes that are constrained in a given direction on boundary β. Readers are referred to [6,21] for the derivation of Eq. (C.1), however, in essence the sum over D^{free} enforces that the discovered values for Q result in the balance of linear momentum being satisfied for the given data and the sum over the boundaries $\beta = 1, \dots, n_\beta$ enforces that the discovered values for Q results in the observed reaction forces.

For the purposes of this work, and without loss of generality, we generate synthetic data using a FE simulation. We choose a Gent-Thomas material model [90], defined by

$$\Psi(\boldsymbol{F}) = 0.5(\tilde{I}_1 - 3) + \log\left(\tilde{I}_2/3\right) + (J-1)^2, \tag{C.2}$$

and specimen geometry and boundary conditions illustrated in Fig. C.14. The specimen consists of a 1×1 square with of hole of radius 0.1 in the bottom left corner that has been extruded by 0.1. Slider boundary conditions are applied on the left, bottom, and back of the specimen, while a unit of upwards displacement is applied to the top of the specimen.

All NCMs made use of a kinematic layer that maps the deformation gradient to polyconvex terms as follows:

$$\mathcal{K}(\boldsymbol{F}) = \left[\tilde{I}_1 - 3 \quad \tilde{I}_2^{3/2} - 3^{3/2} \quad (J-1)^2\right]. \tag{C.3}$$

The hyperparameters for the inner networks are provided in Table C.1 and are comparable to those found in literature, e.g. [5,6,21].

The accuracy of the trained NCMs is evaluated by comparing the predicted strain energy density against that of the ground truth for six different loading paths, namely uniaxial tension (UT), uniaxial compression (UC), biaxial tension (BT), biaxial compression

Table C.1
Hyperparameters used in NCMs.

Parameter	Value
MICNN [9,21,58,59]:	
Number of hidden layers	3
Number of neurons in each hidden layer	16
Total number of learnable parameters	675
CANN [5,8,38,60]:	
Exponents used in power layer	$\{1, 2\}$
Terms used in function layer	$\{f(x) = x, \quad f(x) = \exp x\}$
Total number of learnable parameters	33
ICKAN [6,61]:	
Number of hidden layers	2
Order of splines	4
Number of grid points	30
Total number of learnable parameters	508

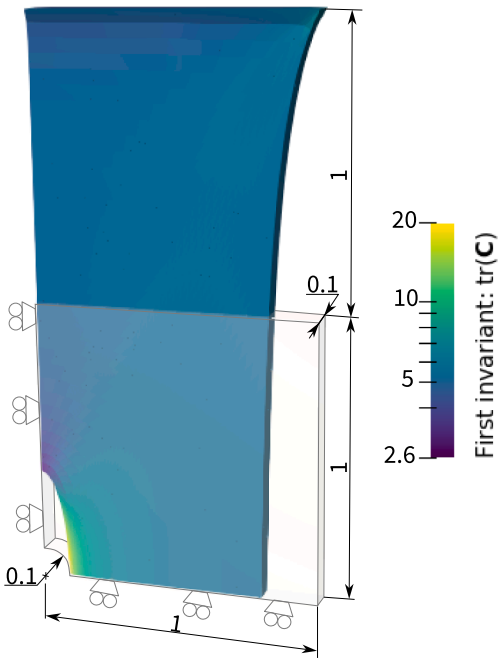

Fig. C.14. Specimen used for NCM training data generation. Both reference and deformed configurations are shown.

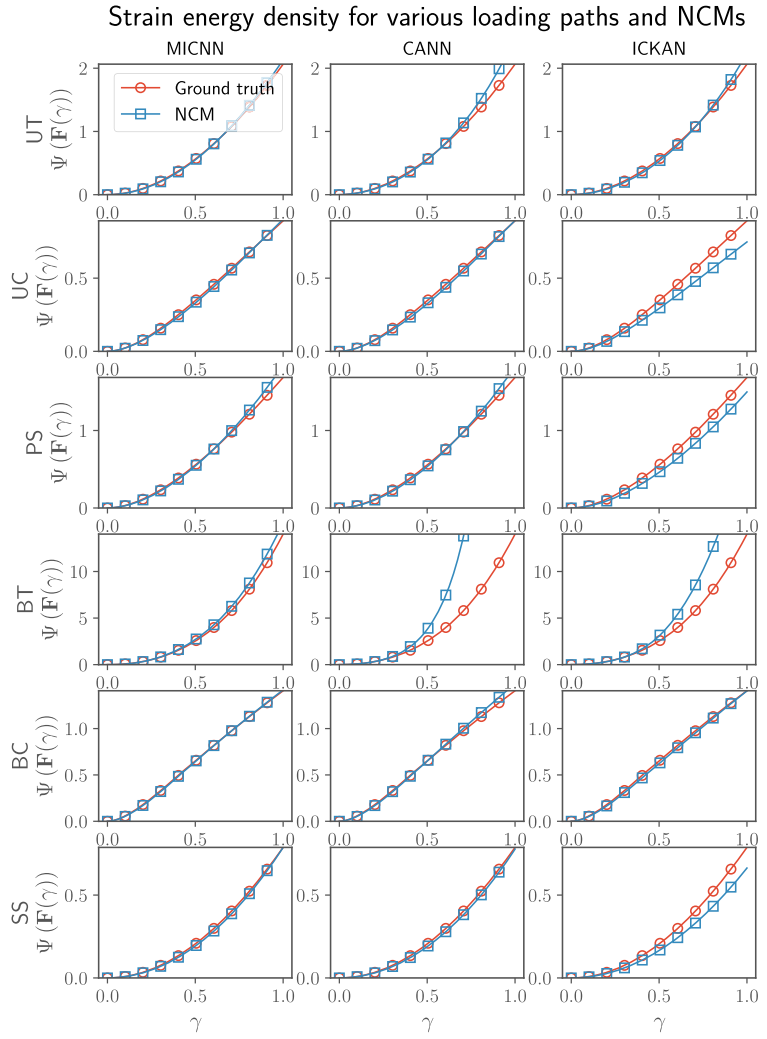

Fig. C.15. Comparison of trained NCM behavior with ground truth. The NCMs are able to discover the ground truth behavior accurately in all cases apart from biaxial tension.

(BC), simple shear (SS), and pure shear (PS), defined, respectively, as follows:

$$\boldsymbol{F}^{\mathrm{UT}}(\gamma)=\begin{bmatrix}1+\gamma & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix},\quad \boldsymbol{F}^{\mathrm{UC}}(\gamma)=\begin{bmatrix}\frac{1}{1+\gamma} & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix},\quad \boldsymbol{F}^{\mathrm{BT}}(\gamma)=\begin{bmatrix}1+\gamma & 0 & 0\\ 0 & 1+\gamma & 0\\ 0 & 0 & 1\end{bmatrix},$$
$$\boldsymbol{F}^{\mathrm{BC}}(\gamma)=\begin{bmatrix}\frac{1}{1+\gamma} & 0 & 0\\ 0 & \frac{1}{1+\gamma} & 0\\ 0 & 0 & 1\end{bmatrix},\quad \boldsymbol{F}^{\mathrm{SS}}(\gamma)=\begin{bmatrix}1 & \gamma & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix},\quad \boldsymbol{F}^{\mathrm{PS}}(\gamma)=\begin{bmatrix}1+\gamma & 0 & 0\\ 0 & \frac{1}{1+\gamma} & 0\\ 0 & 0 & 1\end{bmatrix}. \tag{C.4}$$

We note that these loading paths do not produce e.g. uniaxial tension in the typical sense as the τ_{22} and τ_{33} components of the resulting stress tensor will not in general be zero. However, this is immaterial as the purpose is simply to compare the behavior of the trained NCMs to the ground truth for a small number of interpretable loading paths. The resulting behavior for these loading paths is presented in Fig. C.15.

The NCMs are able to discover the ground truth behavior accurately in all cases apart from biaxial tension, the loading for which is outside of the training data.

Appendix D. Machine details

For completeness, we provide the details of the CPUs used in the computational experiments conducted in this work. The details of the CPU in the workstation and on the HPC nodes are provided in Table D.2.

Table D.2
Details of CPUs used in computational experiments.

	Workstation	HPC node
Model name	AMD Ryzen 9 7950X	Intel(R) Xeon(R) Gold 6248R
Core(s) per socket	16	24
Socket(s)	1	2
CPU max MHz	5881	4000
CPU min MHz	400	1200
L1d cache	512 KiB (16 instances)	32 KiB (48 instances)
L1i cache	512 KiB (16 instances)	32 KiB (48 instances)
L2 cache	16 MiB (16 instances)	1 MiB (48 instances)
L3 cache	64 MiB (2 instances)	35.75 MiB (2 instances)

Appendix E. The effect of model size

The hyperparameter choices and model sizes considered in this work are comparable to those commonly reported in the literature [5,6,21]. However, applications may require models with significantly more or fewer parameters than are typically employed. Consequently, it is also of interest to assess how the performance gains arising from vectorization, batching, and CGO depend on the size of the NCM.

To this end, we repeat the material point benchmarks presented in Section 4.1.2 for MICNNs with varying numbers of hidden layers and layer widths, thereby spanning a wide range of model sizes. In particular, MICNNs are constructed with 2, 3, and 4, hidden layers and layer widths of 2, 4, 8, 16, 32, and 64. In addition, to evaluate whether similar performance gains can be achieved for traditional constitutive laws, we repeat the material point benchmarks for a Gent–Thomas material model.

The resulting speed-ups are shown in Fig. E.16 as a function of batch size for different NCM sizes. Overall, speed-ups due to vectorization, batching, and CGO are larger for smaller NCMs. We speculate that this is because the parameters for the NCM are held in cache during these computations. Hence, for bigger NCMs there is less cache memory available for the data on which the NCM operates. However, since the contents of the CPU cache cannot be directly inspected, this hypothesis cannot be conclusively verified. Unsurprisingly, the speed-up for the Gent-Thomas model is similar to that observed for NCMs with a small number of parameters, i.e. over three orders of magnitude. Hence, batching and vectorization also leads to significant performance gains for traditional constitutive models.

Despite the observed decrease in speed-up with increasing model size, the performance gains remain substantial. In particular, for NCMs with approximately 13,000 parameters, speed-ups exceeding two orders of magnitude are still obtained. This is notable given that typical NCMs reported in the literature contain on the order of 1,000 parameters [21].

In addition to computational performance, memory requirements are an important practical consideration. The RAM usage as a function of batch size and NCM size is therefore shown in Fig. E.17. For the largest NCM considered, containing approximately 13,000 parameters, the total RAM usage remains below 1 GB for batch sizes smaller than 10^4. Notably, this range also encompasses the batch sizes that yield near-optimal speed-ups (Fig. E.16). Such memory requirements are well within the capabilities of modern computing

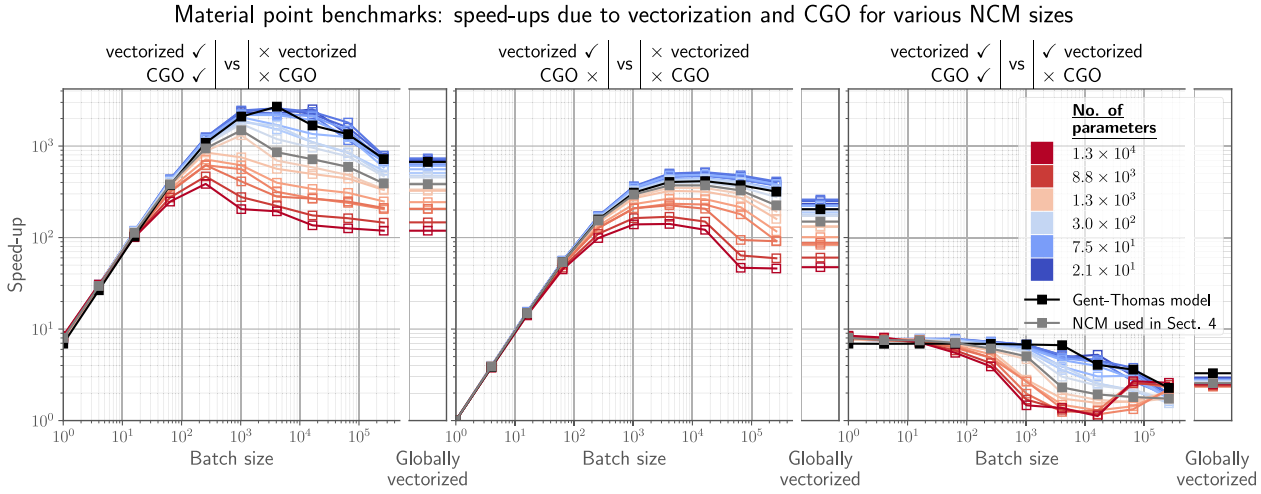

Fig. E.16. Effect of NCM size on speed-up due to batching, vectorization, and CGO: (left) combined speed-ups from CGO and vectorization relative to a non-vectorized, non-CGO baseline; (middle) speed-ups due to vectorization only; and (right) additional speed-ups due to CGO only at different batch sizes. While increasing model size reduces the attainable speed-up, improvements of more than two orders of magnitude are still achieved for models substantially larger than those typically reported in the literature.

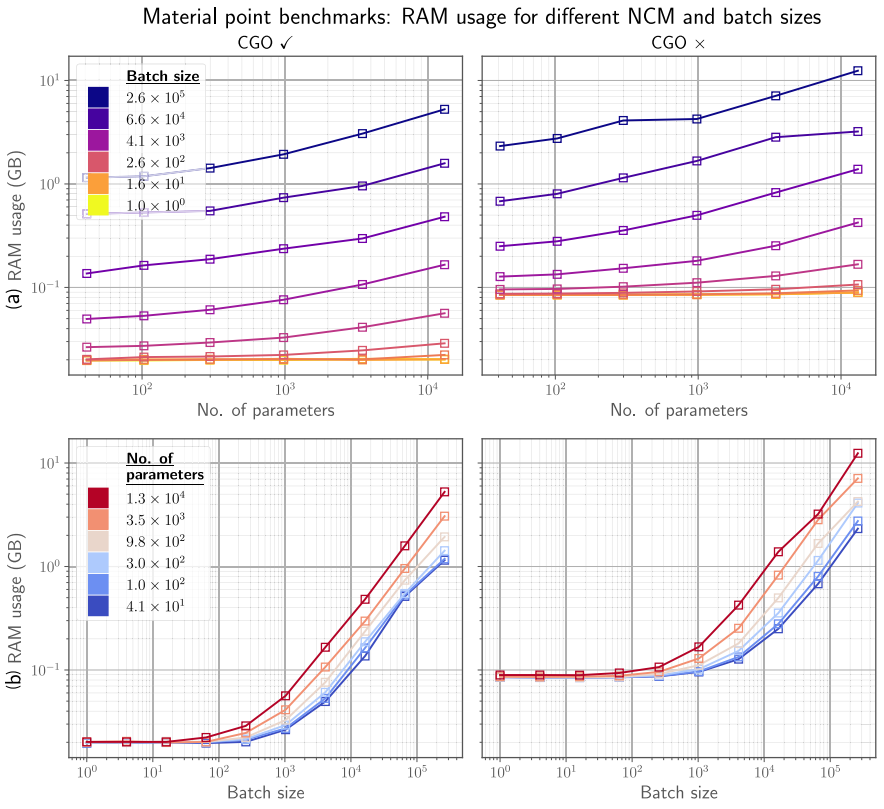

Fig. E.17. Effect of NCM size on RAM usage: RAM consumption (left) with CGO and (right) without CGO, shown (a) as a function of the number of parameters and (b) as a function of batch size.

hardware. Furthermore, a comparison of the left and right columns of Fig. E.17 clearly indicates that the use of CGO reduces overall RAM consumption.

References

[1] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals Syst. 2 (4) (1989) 303–314.

[2] L. Linden, D.K. Klein, K.A. Kalina, O. Brummund, Jörg an d Weeger, M. Kästner, Neural networks meet hyperelasticity: a guide to enforcing physics, J. Mech. Phys. Solids 179 (2023) 105363. https://doi.org/10.1016/j.jmps.2023.105363

[3] G.-L. Geuken, P. Kurzeja, D. Wiedemann, J. Mosler, Input convex neural networks: universal approximation theorem and implementation for isotropic polyconvex hyperelastic energies, 2025. 2502.08534https://doi.org/10.48550/arXiv.2502.08534

[4] D.K. Klein, M. Hossain, K. Kikinov, M. Kannapinn, S. Rudykh, A.J. Gil, Neural networks meet hyperelasticity: A monotonic approach, 2025. 2501.02670https://doi.org/10.48550/arXiv.2501.02670

[5] M. Peirlinck, K. Linka, J.A. Hurtado, E. Kuhl, On automated model discovery and a universal material subroutine for hyperelastic materials, Comput. Methods Appl. Mech. Eng. 418 (2024) 116534. https://doi.org/10.1016/j.cma.2023.116534

[6] P. Thakolkaran, Y. Guo, S. Saini, M. Peirlinck, B. Alheit, S. Kumar, Can kan cans? input-convex kolmogorov-arnold networks (kans) as hyperelastic constitutive artificial neural networks (cans), Comput. Methods Appl. Mech. Eng. 443 (2025) 118089.

[7] K. Linka, M. Hillgärtner, K.P. Abdolazizi, R.C. Aydin, M. Itskov, C.J. Cyron, Constitutive artificial neural networks: a fast and general approach to predictive data-driven constitutive modeling by deep learning, J. Comput. Phys. 429 (2021) 110010. https://doi.org/10.1016/j.jcp.2020.110010

[8] K. Linka, E. Kuhl, A new family of constitutive artificial neural networks towards automated model discovery, Comput. Methods Appl. Mech. Eng. 403 (2023) 115731. https://doi.org/10.1016/j.cma.2022.115731

[9] D.K. Klein, M. Fernández, R.J. Martin, P.z. Neff, O. Weeger, Polyconvex anisotropic hyperelasticity with neural networks, J. Mech. Phys. Solids 159 (2022) 104703. https://doi.org/10.1016/j.jmps.2021.104703

[10] P. Weber, J. Geiger, W. Wagner, Constrained neural network training and its application to hyperelastic material modeling, Comput. Mech. 68 (2021) 1179–1204. https://doi.org/10.1007/s00466-021-02064-8

[11] J.N. Fuhg, A. Jadoon, O. Weeger, D.T. Seidl, R.E. Jones, Polyconvex neural network models of thermoelasticity, J. Mech. Phys. Solids 192 (2024) 105837. https://www.sciencedirect.com/science/article/pii/S002250962400303X.

[12] A.B. Tepole, A.A. Jadoon, M. Rausch, J.N. Fuhg, Polyconvex physics-augmented neural network constitutive models in principal stretches, Int. J. Solids Struct. (2025) 113469. https://www.sciencedirect.com/science/article/pii/S0020768325002550.

[13] J.N. Fuhg, G. Anantha Padmanabha, N. Bouklas, B. Bahmani, W. Sun, N.N. Vlassis, M. Flaschel, P. Carrara, L. De Lorenzis, A review on data-driven constitutive laws for solids, Arch. Comput. Methods Eng. (2024) 1–43.

[14] S. Yang, M. Levin, G.A. Padmanabha, M. Borshevsky, O. Cohen, D.T. Seidl, R.E. Jones, N. Bouklas, N. Cohen, Physics Augmented Machine Learning Discovery of Composition-Dependent Constitutive Laws for 3D Printed Digital Materials, 2025. 2507.02991https://doi.org/10.48550/arXiv.2507.02991

[15] K. Upadhyay, J.N. Fuhg, N. Bouklas, K.T. Ramesh, Physics-informed data-driven discovery of constitutive models with application to strain-rate-sensitive soft materials, Comput. Mech. (2024) 1–30.

[16] M. Flaschel, S. Kumar, L. De Lorenzis, Automated discovery of generalized standard material models with EUCLID, Comput. Methods Appl. Mech. Eng. 405 (2023) 115867. https://doi.org/10.1016/j.cma.2022.115867
[17] M. Flaschel, S. Kumar, L. De Lorenzis, Unsupervised discovery of interpretable hyperelastic constitutive laws, Comput. Methods Appl. Mech. Eng. 381 (2021) 113852. https://doi.org/10.1016/j.cma.2021.113852
[18] M. Flaschel, P. Steinmann, L. De Lorenzis, E. Kuhl, Convex neural networks learn generalized standard material models, J. Mech. Phys. Solids 200 (2025) 106103. https://doi.org/10.1016/j.jmps.2025.106103
[19] M. Flaschel, S. Kumar, L. De Lorenzis, Discovering plasticity models without stress data, npj Comput. Mater. 8 (2022) 1–10. https://doi.org/10.1038/s41524-022-00752-4
[20] A. Dekhovich, O.T. Turan, J. Yi, M.A. Bessa, Cooperative data-driven modeling, Comput. Methods Appl. Mech. Eng. 417 (2023) 116432. https://doi.org/10.1016/j.cma.2023.116432
[21] P. Thakolkaran, A. Joshi, Y. Zheng, M. Flaschel, L. De Lorenzis, S. Kumar, NN-EUCLID: Deep-learning hyperelasticity without stress data, J. Mech. Phys. Solids 169 (2022) 105076. https://doi.org/10.1016/j.jmps.2022.105076
[22] E. Marino, M. Flaschel, S. Kumar, L. De Lorenzis, Automated identification of linear viscoelastic constitutive laws with EUCLID, Mech. Mater. 181 (2023) 104643. https://doi.org/10.1016/j.mechmat.2023.104643
[23] L.F. Li, C.Q. Chen, Equilibrium-based convolution neural networks for constitutive modeling of hyperelastic materials, J. Mech. Phys. Solids 164 (2022) 104931. https://doi.org/10.1016/j.jmps.2022.104931
[24] S. Meng, A.A.K. Yousefi, S. Avril, Machine-learning-based virtual fields method: application to anisotropic hyperelasticity, Comput. Methods Appl. Mech. Eng. 434 (2025) 117580. https://doi.org/10.1016/j.cma.2024.117580
[25] V. Tac, V.D. Sree, M.K. Rausch, A.B. Tepole, Data-driven modeling of the mechanical behavior of anisotropic soft biological tissue, Eng. Comput. 38 (2022) 4167–4182. https://doi.org/10.1007/s00366-022-01733-3
[26] F. As' ad, P. Avery, C. Farhat, A mechanics-informed artificial neural network approach in data-driven constitutive modeling, Int. J. Numer. Methods Eng. 123 (2022) 2738–2759. https://doi.org/10.1002/nme.6957
[27] N.N. Vlassis, R. Ma, W. Sun, Geometric deep learning for computational mechanics part i: anisotropic hyperelasticity, Comput. Methods Appl. Mech. Eng. 371 (2020) 113299. https://doi.org/10.1016/j.cma.2020.113299
[28] M. Rosenkranz, K.A. Kalina, J. Brummund, W. Sun, M. Kästner, Viscoelasticty with physics-augmented neural networks: model formulation and training methods without prescribed internal variables, Comput. Mech. 74 (6) (2024) 1279–1301.
[29] H. Holthusen, L. Lamm, T. Brepols, S. Reese, E. Kuhl, Theory and implementation of inelastic constitutive artificial neural networks, Comput. Methods Appl. Mech. Eng. 428 (2024) 117063. https://doi.org/10.1016/j.cma.2024.117063
[30] M. Mozaffar, R. Bostanabad, W. Chen, K. Ehmann, J. Cao, M.A. Bessa, Deep learning predicts path-dependent plasticity, Proceed. National Acad. Sci. 116 (2019) 26414–26420. https://doi.org/10.1073/pnas.1911815116
[31] P. Weber, W. Wagner, S. Freitag, Physically enhanced training for modeling rate-independent plasticity with feedforward neural networks, Comput. Mech. 72 (2023) 827–857. https://doi.org/10.1007/s00466-023-02316-9
[32] H. Xu, M. Flaschel, L. De Lorenzis, Discovering non-associated pressure-sensitive plasticity models with EUCLID, Adv. Model. Simulat. Eng. Sci. 12 (2025) 1. https://doi.org/10.1186/s40323-024-00281-3
[33] N.N. Vlassis, W. Sun, Sobolev training of thermodynamic-informed neural networks for interpretable elasto-plasticity models with level set hardening, Comput. Methods Appl. Mech. Eng. 377 (2021) 113695. https://doi.org/10.1016/j.cma.2021.113695
[34] L. Zheng, D.M. Kochmann, S. Kumar, HyperCAN: hypernetwork-Driven deep parameterized constitutive models for metamateria ls, Extreme Mech. Lett. 72 (2024) 102243. https://doi.org/10.1016/j.eml.2024.102243
[35] D.K. Klein, R. Ortigosa, J. Martínez-Frutos, O. Weeger, Nonlinear electro-elastic finite element analysis with neural network constitutive models, Comput. Methods Appl. Mech. Eng. 425 (2024) 116910. https://doi.org/10.1016/j.cma.2024.116910
[36] J.N. Fuhg, A. Jadoon, O. Weeger, D.T. Seidl, R.E. Jones, Polyconvex neural network models of thermoelasticity, J. Mech. Phys. Solids https://doi.org/10.1016/j.jmps.2024.105837
[37] M. Zlatić, M. Canajija, Incompressible rubber thermoelasticity: a neural network approach, Comput. Mech. 71 (2023) 895–916. https://doi.org/10.1007/s00466-023-02278-y
[38] M. Peirlinck, J.A. Hurtado, M.K. Rausch, A.B. Tepole, E. Kuhl, A universal material model subroutine for soft matter systems, Eng. Comput. (2024). https://doi.org/10.1007/s00366-024-02031-w
[39] M. Franke, D.K. Klein, O. Weeger, P. Betsch, Advanced discretization techniques for hyperelastic physics-augmented neural networks, Comput. Methods Appl. Mech. Eng. 416 (2023) 116333. https://doi.org/10.1016/j.cma.2023.116333
[40] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D.A. Ham, P.H.J. Kelly, A study of vectorization for matrix-free finite element methods, Int. J. High Perform. Comput. Appl. 34 (2020) 629–644. https://doi.org/10.1177/1094342020945005
[41] M. Kronbichler, K. Kormann, A generic interface for parallel cell-based finite element operator application, Comput. Fluids 63 (2012) 135–147. https://doi.org/10.1016/j.compfluid.2012.04.012
[42] M. Kronbichler, K. Kormann, Fast matrix-Free evaluation of discontinuous galerkin finite element operators, ACM Trans. Math. Software 45 (2019) 1–40. https://doi.org/10.1145/3325864
[43] G.F. Castelli, Numerical Investigation of Cahn-Hilliard-Type Phase-Field Models for Battery Active Particles, Ph.D. thesis, Karlsruhe Institute of Technology (KIT), 2021. https://doi.org/10.5445/IR/1000141249
[44] K. Ljungkvist, Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes, HPC '17, Society for Computer Simulation International, p. 1–12.
[45] P. Munch, K. Kormann, M. Kronbichler, Hyper.deal: an efficient, matrix-free finite-element library for high-dimensional partial differential equations, ACM Trans. Math. Software 47 (2021) 1–34. https://doi.org/10.1145/3469720
[46] T. Xue, S. Liao, Z. Gan, C. Park, X. Xie, W.K. Liu, J. Cao, JAX-FEM: A differentiable GPU-accelerated 3D finite element solver for automatic inverse design and mechanistic data science, Comput. Phys. Commun. 291 (2023) 108802. https://doi.org/10.1016/j.cpc.2023.108802
[47] B.P. Ferreira, M.A. Bessa, Automatically Differentiable Model Updating (ADiMU): conventional, hybrid, and neural network material model discovery including history-dependency, 2025. 2505.07801https://doi.org/10.48550/arXiv.2505.07801
[48] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, Q. Zhang, JAX: composable transformations of Python + NumPy programs, 2018, http://github.com/jax-ml/jax.
[49] S.M. Rosenbusch, P. Diercks, V. Kindrachuk, J.F. Unger, Integrating custom constitutive models into FEniCSx: a versatile approach and case studies, Adv. Eng. Software 206 (2025) 103922. https://doi.org/10.1016/j.advengsoft.2025.103922
[50] Dassault Systèmes Simulia Corp., SIMULIA Abaqus FEA, 2024, (Software). Version 2025; available at https://www.3ds.com/products-services/simulia/products/abaqus/.
[51] Ansys, Inc., ANSYS 2025 R1, 2025, (Software). Version 2025 R1; available at https://www.ansys.com.
[52] M. P. I. Forum, MPI: A Message-Passing Interface Standard Version 5.0, 2025, https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf.
[53] D. Arndt, W. Bangerth, M. Bergbauer, B. Blais, M. Fehling, R. Gassmöller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, S. Scheuerman, B. Turcksin, S. Uzunbajakau, D. Wells, M. Wichrowski, The deal.II library, version 9.7, J. Numer. Math. 33 (4) (2025) 403–415. https://doi.org/10.1515/jnma-2025-0115
[54] T. Belytschko, W.K. Liu, B. Moran, K. Elkhodary, Nonlinear Finite Elements for Continua and Structures, John Wiley & Sons, 2014.
[55] J. Bonet, R.D. Wood, Nonlinear Continuum Mechanics for Finite Element Analysis, Cambridge University Press, 1997.
[56] P. Wriggers, Nonlinear Finite Element Methods, Springer Science & Business Media, 2008.

[57] D. Wiedemann, M.A. Peter, Characterization of polyconvex isotropic functions, 2023. 2304.08385https://doi.org/10.48550/arXiv.2304.08385
[58] B. Amos, L. Xu, J.Z. Kolter, Input convex neural networks, in: Proceedings of the 34Th International Conference on Machine Learning, PMLR, 2017, pp. 146–155. https://proceedings.mlr.press/v70/amos17b.html.
[59] A. Jadoon, D. Seidl, R. Jones, J. Fuhg, Input specific neural networks (2025).
[60] M. Peirlinck, K. Linka, J.A. Hurtado, G.A. Holzapfel, E. Kuhl, Democratizing biomedical simulation through automated model discovery and a universal material subroutine, Comput. Mech. (2024) 1–21.
[61] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljacic, T.Y. Hou, M. Tegmark, KAN: Kolmogorov-Arnold Networks, 2025. 2404.19756https://doi.org/10.48550/arXiv.2404.19756
[62] J.L. Hennessy, D.A. Patterson, Computer architecture: a quantitative approach (2011).
[63] U. Drepper, What every programmer should know about memory, Red Hat., Inc. (2007). https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.
[64] Skylake Processors - HECC Knowledge Base, https://www.nas.nasa.gov/hecc/support/kb/skylake-processors_550.HTML.
[65] Intel Skylake, https://www.7-cpu.com/cpu/Skylake.HTML.
[66] A. Fog, Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, 2022, https://www.agner.org/optimize/instruction_tables.pdf.
[67] The Python Language Reference, https://docs.python.org/3/reference/index.HTML.
[68] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019. 1912.01703https://doi.org/10.48550/arXiv.1912.01703
[69] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015, Software available from tensorflow.org, https://www.tensorflow.org/.
[70] TorchScript —PyTorch 2.8 documentation, https://docs.pytorch.org/docs/stable/jit.HTML,/jit.HTML.
[71] B. Guo, Z. Lin, Q. He, History-aware neural operator: robust data-driven constitutive modeling of path-dependent materials, Comput. Methods Appl. Mech. Eng. 447 (2025) 118358. https://doi.org/10.1016/j.cma.2025.118358
[72] D.T. Seidl, B.N. Granzow, Calibration of elastoplastic constitutive model parameters from full-field data with automatic differentiation-based sensitivities, Int. J. Numer. Methods Eng. 123 (2022) 69–100. https://doi.org/10.1002/nme.6843
[73] S. Al Hassanieh, W.F. Reinhart, A.M. Beese, Efficient material model parameter optimization in finite element analysis with differentiable physics, Comput. Mater. Sci 253 (2025) 113828. https://doi.org/10.1016/j.commatsci.2025.113828
[74] T. Chen, M.C. Messner, Training material models using gradient descent algorithms, Int. J. Plast. 165 (2023) 103605. https://doi.org/10.1016/j.ijplas.2023.103605
[75] A. Griewank, A. Walther, 3. Fundamentals of Forward and Reverse, Society for Industrial and Applied Mathematics, 2008, pp. 31–59. https://doi.org/10.1137/1.9780898717761.ch3
[76] A. Griewank, A. Walther, 10. Jacobian and Hessian Accumulation, Society for Industrial and Applied Mathematics, 2008, pp. 211–243. https://doi.org/10.1137/1.9780898717761.ch10
[77] S. Balay, S. Abhyankar, M.F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E.M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W.D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M.G. Knepley, F. Kong, S. Kruger, D.A. May, L.C. McInnes, R.T. Mills, L. Mitchell, T. Munson, J.E. Roman, K. Rupp, P. Sanan, J. Sarich, B.F. Smith, S. Zampini, H. Zhang, Hongand Zhang, J. Zhang, PETSc Web page, 2025, https://petsc.org/.
[78] L. Kaczmarczyk, Z. Ullah, K. Lewandowski, X. Meng, X.-Y. Zhou, I. Athanasiadis, H. Nguyen, C.-A. Chalons-Mouriesse, E. Richardson, E. Miur, A. Shvarts, M. Wakeni, C. Pearce, MoFEM: an open source, parallel finite element library, J. Open Source Software (2020). https://doi.org/10.21105/joss.01441
[79] I.A. Baratta, J.P. Dean, J.S. Dokken, M. Habera, J.S. Hale, C.N. Richardson, M.E. Rognes, M.W. Scroggs, N. Sime, G.N. Wells, DOLFINx: the next generation FEniCS problem solving environment, 2023, (preprint). https://doi.org/10.5281/zenodo.10447666
[80] V. Taç, K. Linka, F. Sahli-Costabal, E. Kuhl, A.B. Tepole, Benchmarking physics-informed frameworks for data-driven hyperelasticity, Comput. Mech. 73 (2024) 49–65. https://doi.org/10.1007/s00466-023-02355-2
[81] C++ –pytorch 2.7 documentation C++ —PyTorch 2.7 documentation, https://docs.pytorch.org/docs/stable/cpp_index.HTML.
[82] Delft High Performance Computing Centre (DHPC)], DelftBlue Supercomputer (Phase 2), 2024, (https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2).
[83] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: The April 18–20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), ACM Press, 1967, p. 483. https://doi.org/10.1145/1465482.1465560
[84] G. Hager, G. Wellein, Introduction to high performance computing for scientists and engineers (2011). http://www.crcnetbase.com/isbn/9781439811931.
[85] M. Peirlinck, F.S. Costabal, J. Yao, J.M. Guccione, S. Tripathy, Y. Wang, D. Ozturk, P. Segars, T.M. Morrison, S. Levine, E. Kuhl, Precision medicine in human heart modeling, Biomech. Model Mechanobiol. 20 (2021) 803–831. https://doi.org/10.1007/s10237-021-01421-z
[86] M. Peirlinck, K.L. Sack, P. De Backer, P. Morais, P. Se gers, T. Franz, M. De Beule, Kinematic boundary conditions substantially impact in silico ventricular function, Int. J. Numer. Method Biomed. Eng. 35 (1) (2019) e3151.
[87] R. Aróstica, D. Nolte, A. Brown, A. Gebauer, E. Karabelas, J. Jilberto, M. Salvador, M. Bucelli, R. Piersanti, K. Osouli, C. Augustin, H. Finsberg, L. Shi, M. Hirschvogel, M. Pfaller, P.C. Africa, M. Gsell, A. Marsden, D. Nordsletten, F. Regazzoni, G. Plank, J. Sundnes, L. Dede', M. Peirlinck, V. Vedula, W. Wall, C. Bertoglio, A software benchmark for cardiac elastodynamics, Comput. Methods Appl. Mech. Eng. 435 (2025) 117485. https://doi.org/10.1016/j.cma.2024.117485
[88] G. Sommer, A.J. Schriefl, M. Andrä, M.a. Sacherer, C. Viertler, H. Wolinski, G.A. Holzapfel, Biomechanical properties and microstructure of human ventricular myocardium, Acta. Biomater. 24 (2015) 172–192. https://doi.org/10.1016/j.actbio.2015.06.031
[89] C. de Boor, On calculating with b-splines, J. Approxim. Theory 6 (1972) 50–62. https://doi.org/10.1016/0021-9045(72)90080-9
[90] A.N. Gent, A.G. Thomas, Forms for the stored (strain) energy function for vulcanized rubber, J. Polymer Sci. 28 (1958) 625–628. https://doi.org/10.1002/pol.1958.1202811814