Fast hardware-aware matrix-free algorithm for higher-order finite-element discretized matrix multivector products on distributed systems

Gourab Panigrahi[a,1], Nikhil Kodali[a,1], Debashis Panda[a,2], Phani Motamarri[a]

[a]*Department of Computational and Data Sciences, Indian Institute of Science, CV Raman Road, Bengaluru, 560012, Karnataka, India*

Abstract

Recent hardware-aware matrix-free algorithms for higher-order finite-element (FE) discretized matrix-vector multiplications reduce floating point operations and data access costs compared to traditional sparse matrix approaches. This work proposes efficient matrix-free algorithms for evaluating FE discretized matrix-multivector products on both multi-node CPU and GPU architectures. We address a critical gap in existing matrix-free implementations, which are well suited only for the action of FE discretized matrices on a single vector. We employ batched evaluation strategies, with the batchsize tailored to underlying hardware architectures, leading to better data locality and enabling further parallelization. On CPUs, we utilize even-odd decomposition, SIMD vectorization, and overlapping computation and communication strategies. On GPUs, we employ strategies to overlap compute and data movement in conjunction with GPU shared memory, constant memory, and kernel fusion to reduce data accesses. Our implementation outperforms the baselines for Helmholtz operator action, achieving up to 1.4x improvement on one CPU node and up to 2.8x on one GPU node, while reaching up to 4.4x and 1.5x improvement on multiple nodes for CPUs ($\sim$ 3000 cores) and GPUs ($\sim$ 25 GPUs), respectively. We further benchmark the performance of the proposed implementation for solving a model eigenvalue problem for 1024 smallest eigenvalue-eigenvector pairs by employing the Chebyshev Filtered Subspace Iteration method, achieving up to 1.5x improvement on one CPU node and up to 2.2x on one GPU node while reaching up to 3.0x and 1.4x improvement on multinode CPUs ($\sim$ 3000 cores) and GPUs ($\sim$ 25 GPUs), respectively.

Keywords: Matrix-free, Finite Element Method, Sum factorization, Scalable algorithms for heterogeneous architectures

Email address: phanim@iisc.ac.in (Phani Motamarri)

[1]Gourab Panigrahi and Nikhil Kodali contributed equally to this work.

[2]Currently at Department of Chemical Engineering in Imperial College London, U.K.

1. Introduction

Finite-element (FE) based computational methodologies are routinely employed to numerically solve partial differential equations (PDEs) arising in various domains of science and engineering. The underlying FE basis functions are usually the compactly supported piecewise-continuous Lagrange polynomials. The numerical solution of a partial differential equation employing the FE basis usually involves constructing an FE discretized operator, which is a sparse matrix due to the compact support of these FE basis functions. Consequently, the PDE reduces to a sparse system of linear equations or sparse matrix eigenvalue problems. These sparse matrix problems are traditionally solved using iterative solvers, which require computing the action of the sparse matrix on trial FE discretized fields for the solution of a linear system of equations or eigenvalue problems. Evaluation of the product of the sparse matrix and the vector (FE discretized field) is usually the computationally demanding step. It is traditionally computed using sparse-matrix vector multiplication algorithms [1, 2]. However, previous works [3–5] note that the evaluation of such sparse matrix-vector products for higher-order finite-elements can be performed more efficiently on multithreaded architectures using FE-cell level dense matrix-vector multiplications followed by the assembly of FE-cell level product vectors. Motamarri et al. [6], Das et al. [7] have recently employed this strategy on multi-node CPU and GPU architectures for evaluating the FE discretized matrix-multivector products involving a large number of vectors (>300). They have demonstrated a good throughput performance for the solution of FE discretized large-scale nonlinear eigenvalue problems arising in the field of quantum modeling of materials using density functional theory. However, recent hardware-aware algorithms for evaluating such matrix-vector multiplications suggest that computing on-the-fly matrix-vector products without storing the FE-cell level dense matrices reduces arithmetic complexity, data movement and memory footprint [8–11]. These algorithms, referred to as matrix-free approaches, exploit the tensor-structured nature of the FE basis functions and recast the 3D integrals involved in the matrix-vector products as a sequence of tensor contractions. The open-source implementations of the above matrix-free methods currently available to the community [12–15] are neither optimal nor directly applicable for the action of an FE discretized operator on a large number of vectors. Such situations are often encountered when solving FE discretized eigenvalue problems [2, 16] using iterative orthogonal projection approaches or solving linear systems of equations arising from FE discretizations with multiple RHS vectors. These problems arise in real-space quantum modeling of materials [7, 17, 18], electroelastics [19], modal analysis [20, 21], and scientific machine learning to train ML models with the solutions of FE discretized PDEs involving multiple forcing vectors [22]. Although some preliminary works, such as interpolation of mulitvectors to quadrature points [23] and evaluation of FE operator action on sparse multivectors [24] exist in this regard, no efficient algorithm exists for performing generic FE discretized matrix-multivector multiplication efficiently under the matrix-free paradigm. This work proposes an efficient hardware-aware matrix-free algorithm and implementation strategies to compute such FE discretized matrix-multivector products on multi-node CPU-only and multi-node GPU architectures.

To this end, as traditionally done in the finite-element literature, we partition the physical domain into non-overlapping subdomains, each assigned to an `MPI` task, and use the `MPI` paradigm to communicate the boundary data across multiple nodes. The tensor contractions involved in the matrix-free approach are recast as small dense matrix-matrix multiplications involving the FE shape function matrices. On CPU architectures, to compute these small dense matrix-matrix multiplications, we utilize the SIMD vectorization capabilities of modern CPUs along with optimal implementation strategies that exploit the symmetry of the FE shape function matrices (such as the *even-odd* decomposition [25, 26]) to minimize the computation time and use non-blocking `MPI` communications to overlap computation and communication, allowing for higher scaling efficiencies. Our proposed implementation utilizes a batched layout for the storage of the multivector, which improves data locality and allows for efficient use of the SIMD capabilities on modern CPUs, and the *even-odd* decomposition strategy reduces the floating point operations required to compute the matrix multivector products by half at the cost of increased data movement. On GPU architectures, the proposed matrix-free implementation efficiently utilizes the GPU shared memory and registers to pipeline data access and computation in conjunction with the proposed batched layout. The small matrix-matrix multiplications arising in the matrix-free approach are performed as a linear combination of columns of FE shape function matrices, which are stored in constant memory, to overlap computation with data movement from device memory. Furthermore, constant memory is utilized to broadcast accesses of the FE shape function matrices and reduces shared memory usage and bank conflicts. The proposed implementation also utilizes the concept of kernel fusion to minimize data access by combining various implementation steps in a single kernel. This has the added benefit of reducing the memory footprint further. We also employ CUDA-Aware MPI to optimize communications and a mixed precision strategy to communicate data on the shared subdomain boundary to reduce the amount of data that needs to be communicated.

In Section 2, we provide a concise account of the mathematical formulation of the problem that we intend to solve using a finite-element based discretization technique. Subsequently, we delve deeper into the mathematical underpinnings of the cell-matrix and matrix-free methods as applied to multivectors, specifically utilizing adaptively refined hexahedral meshes. Furthermore, we describe the various steps involved in evaluating matrix-multivector products within these frameworks, such as subdomain partitioning, the imposition of constraints to ensure continuity, extraction of FE-cell level representations and the assembly of subdomain level representations.

In Section 3, we first describe the mathematical aspects of our proposed algorithm and subsequently delve into the numerical implementation strategies employed to evaluate matrix-multivector products in the matrix-free paradigm. A key consideration in this context is the adaptation of the algorithm to the specific characteristics of the underlying hardware architecture.

To this end, we propose a batched algorithm in which we concurrently process a limited subset of vectors, known as a batch. The dimension of this batch is chosen based on the properties of the underlying hardware architecture. We also propose a batched layout for storing the multivectors, which significantly improves the data locality for the implementation of our proposed batched algorithm. Furthermore, we briefly describe the methods used for imposing constraints, the extraction of FE-cell level representations, and other relevant operations. We further describe the strategy employed for the implementation of the tensor contractions on both CPU-only and GPU-based architectures in Section 3.2.4.

In Section 4, we benchmark the performance of our implementation using a representative FE discretized matrix on multi-node CPU architectures (NSM[3] Param Pravega) and multi-node GPU architectures (ORNL[4] Summit supercomputer). Specifically, as a model problem, we compute the action of the Helmholtz problem on multivectors of various sizes. We use a cell-matrix implementation and the existing matrix-free implementation from the `deal.II` library as our baselines. We begin our comparative study with the `deal.II` matrix-free approach for a single vector. Our GPU implementation outperforms the `deal.II` matrix-free baseline on a single GPU (∼120k DoFs/GPU) with a speedup of 16x – 17.5x for the single vector case with polynomial orders 6, 7 and 8. Hence, we do not consider the `deal.II` method as a baseline for the evaluation of matrix-multivector products on GPU architectures. We subsequently benchmark our matrix-free multivector implementation against the chosen baselines. Our results indicate the superior performance of our proposed implementation, demonstrating computational gains of 2x – 2.8x on one Summit node (6 GPUs, ∼200k DoFs/GPU), 16% – 30% on 16 Summit nodes (96 GPUs, ∼12k DoFs/GPU), and 2.4x – 4.4x on 64 nodes of Param Pravega (3072 CPU `MPI` tasks, ∼700 DoFs per `MPI` task) for matrix-multivector products (1024 vectors) compared to the best baseline implementation for polynomial orders 6, 7 and 8. Additionally, we present the strong scaling studies of our proposed implementation on both multi-node CPU and GPU architectures.

We further benchmark our implementation strategy by solving the eigenvalue problem involving the differential operator $-\mu\nabla^2 + \kappa(\mathbf{x})$, we show speedups of 1.6x – 2.2x on a uniform mesh for 1 Summit node (6 GPUs, ∼200k DoFs/GPU), 14% – 41% for 4 Summit nodes (24 GPUs, ∼50k DoFs/GPU), and 2x – 3x on 64 nodes of Param Pravega (3072 CPU `MPI` tasks, ∼700 DoFs per `MPI` task) for matrix-multivector products (1024 vectors) compared to the best baseline implementation for polynomial order 6, 7 and 8. In addition, we report benchmarks on adaptively refined meshes for our matrix-free implementation against the baselines. Finally in Section 5, we present brief concluding remarks with a future outlook.

[3]National Supercomputing Mission, India
[4]Oak Ridge National Laboratory, USA

2. Methodology

2.1. Mathematical background

Consider a partial differential equation (PDE) defined on a bounded domain $\Omega \subset \mathbb{R}^3$ involving the differential operator $\mathcal{F} = -\mu\nabla^2 + \kappa(\mathbf{x})$ with $\mu \in \mathbb{R}$ and $\kappa(\mathbf{x}) : \Omega \to \mathbb{R}$. Note that the operator $\mathcal{F}$ is reduced to the Laplace operator if $\mu = 1,\ \kappa(\mathbf{x}) = 0 \quad \forall \mathbf{x} \in \Omega$ and to the Helmholtz operator if $\mu = 1,\ \kappa(\mathbf{x}) = k^2 \quad \forall \mathbf{x} \in \Omega$, where $k \in \mathbb{R}$ is a constant.

To elucidate our matrix-free multivector algorithmic strategies developed in the current work, we introduce the following problem of finding $u^\beta(\mathbf{x}) \in \mathcal{V}$ with $\beta = 1, 2, \ldots, n_v$ such that

$$\begin{aligned} \mathcal{F}u^\beta(\mathbf{x}) = -\mu\nabla^2 u^\beta(\mathbf{x}) + \kappa(\mathbf{x})u^\beta(\mathbf{x}) &= \begin{cases} f^\beta(\mathbf{x}) \\ \lambda^\beta u^\beta(\mathbf{x}) \end{cases} \qquad \forall \mathbf{x} \in \Omega \\ u^\beta(\mathbf{x}) = u_D(\mathbf{x}) \quad \forall \mathbf{x} \in \partial\Omega_D & \end{aligned} \tag{1}$$

where $\mathcal{V}$ denotes a suitable function space in which the solution of the problem in Eq. (1) lies, and $u_D(\mathbf{x})$ in the above equation corresponds to the Dirichlet boundary condition applied on $\partial\Omega_D \subseteq \partial\Omega$ and the boundary of Ω. If the choice of the RHS is a set of forcing functions $f^\beta(\mathbf{x}) : \Omega \to \mathbb{R}$, for $\beta = 1, 2, \ldots, n_v$, the above problem represents a set of linear PDEs. If the choice of the RHS is $\lambda^\beta u^\beta(\mathbf{x})$, then Eq. (1) represents an eigenvalue problem with $(\lambda_\beta, u^\beta(\mathbf{x}))$ as the eigenvalue and eigenfunction pair corresponding to the operator $\mathcal{F}$. Eigenvalue problems of this nature with large n_v are very similar to those arising in quantum-modeling of materials using Kohn-Sham density functional theory (DFT) [7, 27], electroelastics [19] and modal analysis [20, 21]

We now consider the discretization of the eigenvalue problem in Eq. (1) using finite-elements, a strictly local piecewise polynomial basis comprising of C^0 continuous Lagrange polynomials generated using Gauss Lobatto Legendre (GLL) nodal points [28]. To this end, we consider the finite-dimensional space $\mathbb{V}_m^h \subset \mathcal{V}$ with a 3D tensor-structured finite-element (FE) basis $N_J^h(\mathbf{x}) : 1 \leq J \leq m$ constructed from strictly local 1D Lagrange interpolating polynomials of order $n_p - 1$, generated using the nodes of the FE triangulation $\mathcal{T}^h$, with the characteristic mesh size denoted by h. Consequently, the discretization of the solution fields in Eq. (1) using the FE basis is given by $u^{\beta,h}(\mathbf{x}) = \sum_{J=1}^m u_J^\beta N_J^h(\mathbf{x})$, where u_J^β denotes the coefficient of the β^{th} discretized field for $\beta = 1, 2, \ldots, n_v$.

Finally, the finite-element discretization of the eigenvalue problem in Eq. (1) results in the following:

$$\begin{aligned} \mathbf{K}\boldsymbol{u}^\beta + \mathbf{M}^\kappa \boldsymbol{u}^\beta &= \lambda_\beta \mathbf{M}\boldsymbol{u}^\beta \\ u_J^\beta = \Pi u_D(\mathbf{x}_J) \qquad & \forall \mathbf{x}_J \in \partial\Omega_D \end{aligned} \tag{2}$$

to be solved for the eigenvalues $\lambda^\beta \in \mathbb{R}$ and eigenvectors $\boldsymbol{u}^\beta \in \mathbb{R}^m \quad \forall \beta = 1, \ldots, n_v$ comprising of the FE nodal degrees of freedom (DoFs), where $\Pi u_D(\mathbf{x}_J)$ is the interpolant of $u_D(\mathbf{x})$ in $\mathbb{V}_m^h$ and $\mathbf{K}$, $\mathbf{M}$ and $\mathbf{M}^\kappa$ denote the stiffness matrix, mass matrix (FE basis overlap matrix) and the weighted mass matrix respectively,

and are given by:

$$K_{IJ} = \int_\Omega \mu \nabla N_I^h(\mathbf{x}) \cdot \nabla N_J^h(\mathbf{x})\, d\mathbf{x} \tag{3a}$$

$$M_{IJ} = \int_\Omega N_I^h(\mathbf{x})\, N_J^h(\mathbf{x})\, d\mathbf{x} \tag{3b}$$

$$M_{IJ}^\kappa = \int_\Omega \kappa(\mathbf{x})\, N_I^h(\mathbf{x})\, N_J^h(\mathbf{x})\, d\mathbf{x} \tag{3c}$$

Defining the multivector matrix $\mathbf{U} = \left[\boldsymbol{u}^1\ \boldsymbol{u}^2\ \ldots\ \boldsymbol{u}^{n_v}\right]$, we can now rewrite Eq. (2) as

$$\begin{aligned} &\mathbf{KU} + \mathbf{M}^\kappa\mathbf{U} = \mathbf{MU}\boldsymbol{\Lambda} \\ &U_{J\beta} = \Pi u_D(\mathbf{x}_J) \qquad \forall \mathbf{x}_J \in \partial\Omega_D \end{aligned} \tag{4}$$

where $\Lambda_{\alpha\beta} = \delta_{\alpha\beta}\lambda_\alpha$.

The computational efficiency of an iterative solution strategy for solving the eigenvalue problem in Eq. (4) relies on the efficient evaluation of matrix multivector products $\mathbf{KU}$, $\mathbf{M}^\kappa\mathbf{U}$ and $\mathbf{MU}$ on distributed heterogeneous architectures, which will be the primary focus of this work.

2.2. Matrix multivector product

According to the standard prescription of finite-element (FE) discretization, we decompose Ω into non-overlapping volumes called finite-element cells $\Omega^{(e)}$ i.e., $\Omega = \bigcup_{e=1}^{E} \Omega^{(e)}$ where E is the number of finite-element cells. We refer to these elements $\Omega^{(e)}$ as FE-cells, and in this work, we choose them to be hexahedra. Furthermore, we assume that a linear map exists from each FE-cell to a reference domain $\widehat{\Omega} = [-1, 1]^3$ with $\boldsymbol{\xi} = [\xi_1, \xi_2, \xi_3]$ as the reference coordinate system. In this framework, the FE discretized field $u^{\beta,h}(\mathbf{x})$ for a given FE-cell (e) can be defined as follows:

$$u^{\beta,h\,(e)}(\mathbf{x}(\boldsymbol{\xi})) = \sum_{J=1}^{n_p^3} u^{\beta,h\,(e)}\left(\mathbf{x}_J^{(e)}\right) \widehat{N}_J(\boldsymbol{\xi}) \tag{5}$$

where $\widehat{N}_J$ is the 3D finite-element (FE) cell level basis function of polynomial order $(n_p - 1)^3$ corresponding to the FE node J.

To make the problem more amenable to distributed parallelism, we partition the domain Ω into subdomains $\Omega^{(t)}\ \forall t = 1, 2, \ldots, n_t$, where n_t is the number of subdomains, and assign each subdomain $\Omega^{(t)}$ to an MPI task t. Let E_t be the number of FE-cells, and m_t be the number of basis functions in each subdomain $\Omega^{(t)}$ such that $\Omega^{(t)} = \bigcup_{e=1}^{E_t} \Omega^{(e)}$. Consequently, the matrix-multivector product $\mathbf{AU}$, where $\mathbf{A}$ denotes the FE discretized matrix (such as $\mathbf{K}$, $\mathbf{M}^\kappa$, $\mathbf{M}$ or $\mathbf{K} + \mathbf{M}^\kappa$), can be written as follows:

$$\mathbf{V} = \mathbf{AU} = \left[\sum_t^{n_t} \mathbf{P}^{(t)^T}\mathbf{C}^{(t)^T}\left(\sum_e^{E_t} \mathbf{Q}^{(e,t)^T}\mathbf{A}^{(e)}\mathbf{Q}^{(e,t)}\right)\mathbf{C}^{(t)}\mathbf{P}^{(t)}\right]\mathbf{U} \tag{6}$$

where the multi-index (e, t) denotes the FE-cell index (e) associated with an MPI task (t) and the Boolean sparse matrix $\mathbf{P}^{(t)}$ denotes the partitioner matrix whose action on $\mathbf{U}$ gives the subdomain level multivector $\mathbf{U}^{(t)}$. The matrix $\mathbf{P}^{(t)}$ imposes the continuity of the field $u^{\beta,h}(\mathbf{x})$ across the partitioned subdomains. Further, the Boolean sparse matrix $\mathbf{C}^{(t)}$ in Eq. (6) denotes an $m_t \times m_t$ constraint matrix employed to constrain the values of the $m_t \times n_v$ matrix $\mathbf{U}^{(t)}$ at certain nodal points. These constraints are used either to satisfy the necessary boundary conditions imposed on $u^{\beta,h}(\mathbf{x})$ or to deal with constraints arising from non-conforming meshes [29]. Furthermore, the imposition of the continuity condition associated with $u^{\beta,h}(\mathbf{x})$ across FE-cells within a partitioned subdomain $\Omega^{(t)}$ is accomplished by the action of $n_p^3 \times m_t$ Boolean sparse matrix $\mathbf{Q}^{(e,t)}$ on the constrained subdomain level multivector $\mathbf{C}^{(t)}\mathbf{P}^{(t)}\mathbf{U}$, with $\mathbf{Q}^{(e,t)}$ representing the subdomain level to FE-cell level map on the subdomain $\Omega^{(t)}$. Finally, the FE-cell level matrix $\mathbf{A}^{(e)}$ arising from the finite-element discretization of the underlying PDE can be evaluated as an integral over the reference domain $\widehat{\Omega}$. For example, the $n_p^3 \times n_p^3$ FE-cell level matrix $\mathbf{K}^{(e)}$ associated with the matrix $\mathbf{K}$ in Eq. (4) can be evaluated as

$$K_{IJ}^{(e)} = \int_{\Omega^{(e)}} \mu \nabla N_I \cdot \nabla N_J d\mathbf{x} \tag{7a}$$

$$= \int_{\widehat{\Omega}} \mu \left(\mathbf{J}^{(e)^{-T}} \nabla_{\boldsymbol{\xi}} \widehat{N}_I\right) \cdot \left(\mathbf{J}^{(e)^{-T}} \nabla_{\boldsymbol{\xi}} \widehat{N}_J\right) \det \mathbf{J}^{(e)} d\widehat{\mathbf{x}} \tag{7b}$$

$$= \sum_{Q=1}^{n_q^3} \left(\nabla_{\boldsymbol{\xi}} \widehat{N}_I\right)^T \mathbf{J}^{(e)^{-1}} \mathbf{J}^{(e)^{-T}} \left(\nabla_{\boldsymbol{\xi}} \widehat{N}_J\right) \mu w_Q \det \mathbf{J}^{(e)}\Big|_{\widehat{\boldsymbol{\xi}}_Q} \tag{7c}$$

where $\nabla_{\boldsymbol{\xi}} \widehat{N}_I$ denotes the gradient of the FE-cell level basis function within reference coordinate system $\boldsymbol{\xi}$, while $\mathbf{J}^{(e)}$ denotes the Jacobian matrix of the map from $\Omega^{(e)}$ to $\widehat{\Omega}$. Furthermore, a tensor structured n_q-point quadrature rule with quadrature points $\widehat{\boldsymbol{\xi}}_Q$ and the quadrature weights w_Q is used in Eq. (7c) for evaluating the integral involved in Eq. (7b).

Defining $D_{QI}^{(s)} = \nabla_{\boldsymbol{\xi}} \widehat{N}_I\left(\widehat{\boldsymbol{\xi}}_Q\right) \cdot \widehat{\boldsymbol{n}}_s$ as $n_q^3 \times n_p^3$ matrices where $\widehat{\boldsymbol{n}}_s$, $s = 0, 1, 2$ represents the unit vector along the s axis, we can now rewrite Eq. (7c) as

$$\mathbf{K}^{(e)} = \begin{bmatrix} \mathbf{D}^{(0)} \\ \mathbf{D}^{(1)} \\ \mathbf{D}^{(2)} \end{bmatrix}^T \begin{bmatrix} \boldsymbol{\mathcal{G}}^{(0,0)} & \boldsymbol{\mathcal{G}}^{(0,1)} & \boldsymbol{\mathcal{G}}^{(0,2)} \\ \boldsymbol{\mathcal{G}}^{(1,0)} & \boldsymbol{\mathcal{G}}^{(1,1)} & \boldsymbol{\mathcal{G}}^{(1,2)} \\ \boldsymbol{\mathcal{G}}^{(2,0)} & \boldsymbol{\mathcal{G}}^{(2,1)} & \boldsymbol{\mathcal{G}}^{(2,2)} \end{bmatrix} \begin{bmatrix} \mathbf{D}^{(0)} \\ \mathbf{D}^{(1)} \\ \mathbf{D}^{(2)} \end{bmatrix} \tag{8}$$

where $\boldsymbol{\mathcal{G}}^{(s,d)}$ for $s, d = 0, 1, 2$ are $n_q^3 \times n_q^3$ diagonal matrices with the diagonal entry $\mathcal{G}_{QQ}^{(s,d)} = \left[\left(\mathbf{J}^{(e)}\right)^{-1}\left(\mathbf{J}^{(e)}\right)^{-T}\right]_{sd} \det \mathbf{J}^{(e)} \mu w_Q\Big|_{\widehat{\boldsymbol{\xi}}_Q}$. We can rewrite the weighted mass matrix in the same framework as

$$\mathbf{M}^{\kappa,(e)} = \mathbf{N}^T \boldsymbol{\mathcal{G}} \mathbf{N} \tag{9}$$

where $N_{QI} = \widehat{N}_I(\widehat{\boldsymbol{\xi}}_Q)$ is an $n_q^3 \times n_p^3$ matrix and $\mathcal{G}_{QQ} = \kappa \det \mathbf{J}^{(e)} w_Q\Big|_{\widehat{\boldsymbol{\xi}}_Q}$ is an $n_q^3 \times n_q^3$ matrix. We obtain the unweighted mass matrix $\mathbf{M}^{(e)}$ by setting $\kappa = 1$.

A straightforward approach to evaluate the matrix-multivector product $\mathbf{V} = \mathbf{AU}$ as outlined in Eq. (6) is to construct the *global FE discretized matrix* $\mathbf{A}$ and perform the sparse-matrix dense-matrix product in a distributed setting. As demonstrated by Kronbichler and Kormann [9], this method is computationally less efficient than the alternative methods discussed herein. In the spirit of the strategies employed for FE discretized matrix-single

vector multiplication, we now discuss two computationally efficient methods for evaluating the matrix-multivector products $\mathbf{V} = \mathbf{A}\mathbf{U}$.

2.2.1. Evaluation -via- FE-cell level local dense matrices

The matrix multivector product $\mathbf{V} = \mathbf{A}\mathbf{U}$ can be evaluated using the FE-cell level matrices $\mathbf{A}^{(e)}$ and the FE-cell level multivectors [3, 7]. This strategy comprises the following steps :

1. Precompute the FE-cell level operator matrices $\mathbf{A}^{(e)}$
2. Extraction of the FE-cell level multivectors $\mathbf{U}^{(e,t)}$ using the *subdomain level to FE-cell level map*, the *constraint* and the *partitioner* matrices, i.e., $\mathbf{U}^{(e,t)} = \mathbf{Q}^{(e,t)}\mathbf{C}^{(t)}\mathbf{P}^{(t)}\mathbf{U}$ $\forall e = 1, 2, \ldots, E_t$
3. FE-cell level evaluation of the matrix multivector product $\mathbf{V}^{(e,t)} = \mathbf{A}^{(e)}\mathbf{U}^{(e,t)}$ using batched matrix-matrix multiplication.
4. Assembly of the global multivector $\mathbf{V}$ using the *subdomain level to FE-cell level map*, the *constraint* and *partitioner* matrices, i.e., $\mathbf{V} = \sum_t^{n_t} \mathbf{P}^{(t)^T}\mathbf{C}^{(t)^T}\sum_e^{E_t}\mathbf{Q}^{(e,t)^T}\mathbf{V}^{(e,t)}$

In the above framework, the FE-cell level evaluation (Step 3) is the computationally dominant step with the computational complexity of $O(n_p^6 n_v)$. Furthermore, this method requires us to store the FE-cell level matrices and multivectors, resulting in a memory footprint of $O(n_p^6 + n_p^3 n_v)$.

2.2.2. Evaluation -via- matrix-free approach

Here, we propose a matrix-free approach to compute matrix-multivector products, inspired by the existing matrix-free matrix-vector multiplication strategies [9]. In this approach, we avoid the precomputation of the FE-cell level matrices $\mathbf{A}^{(e)}$ and instead, the FE-cell level matrix multivector products $\mathbf{A}^{(e)}\mathbf{U}^{(e,t)}$ are evaluated *on-the-fly*. Using the expressions in Eqs. (8) and (9), we observe that the first step in evaluating $\mathbf{V}^{(e,t)} = \mathbf{A}^{(e)}\mathbf{U}^{(e,t)} = \left(\mathbf{K}^{(e)} + \mathbf{M}^{\kappa,(e)}\right)\mathbf{U}^{(e,t)}$ involves computing the action of $\mathbf{D}^{(k)}$ and $\mathbf{N}$ on $\mathbf{U}^{(e,t)}$. To accomplish this, we exploit the tensor-structured nature of the FE basis functions and the quadrature rules. Recalling $\widehat{N}_J(\boldsymbol{\xi})$ and w_Q denote the 3D FE-cell level basis functions and the 3D quadrature weights introduced in Eqs. (5) and (7c) respectively, we have

$$\widehat{N}_J(\boldsymbol{\xi}) = \widehat{N}^{1D}_{j_1}(\xi_1)\widehat{N}^{1D}_{j_2}(\xi_2)\widehat{N}^{1D}_{j_3}(\xi_3) \tag{10}$$

$$w_Q = w^{1D}_{q_1} w^{1D}_{q_2} w^{1D}_{q_3} \tag{11}$$

In the above Eq. (10), we express $\widehat{N}_J(\boldsymbol{\xi})$ as the product of three 1D Lagrange interpolating polynomials of order `FEOrder`$= n_p - 1$, defined on the Gauss Legendre Lobatto nodal points in $[-1, 1]$, with n_p denoting the number of nodal points in each direction. Further, Eq. (11) expresses 3D quadrature weights as the product of 1D quadrature weights w^{1D}_q with $q = 1, \ldots, n_q$ denoting the quadrature weights of the 1D quadrature rule.

Now, we treat the FE-cell level multivector $\mathbf{U}^{(e,t)}$ as a 4^{th} order tensor $\boldsymbol{\mathcal{U}}^{(e,t)}$ with its components denoted as $\mathcal{U}^{(e,t)}_{\beta,j_1,j_2,j_3} = u^{\beta,h\,(e)}\left(\mathbf{x}^{(e)}_J\right) = u^{\beta,h,(e)}\left(\mathbf{x}^{(e)}_{j_1,j_2,j_3}\right)$, with one dimension of $\boldsymbol{\mathcal{U}}^{(e,t)}$ corresponding to the vector index (β) and the other three corresponding to the spatial indices (j_1, j_2, j_3) of the node J. To this end, the action of $\mathbf{D}^{(k)}$ and $\mathbf{N}$ on $\mathbf{U}^{(e,t)}$ is represented as

$$\mathbf{N}\mathbf{U}^{(e,t)} \equiv \left(\mathbf{N}^{1D}\otimes\mathbf{N}^{1D}\otimes\mathbf{N}^{1D}\otimes\mathbf{I}\right)\boldsymbol{\mathcal{U}}^{(e,t)} \tag{12}$$

$$\begin{bmatrix}\mathbf{D}^{(0)}\\ \mathbf{D}^{(1)}\\ \mathbf{D}^{(2)}\end{bmatrix}\mathbf{U}^{(e,t)} \equiv \begin{bmatrix}\mathbf{N}^{1D}\otimes\mathbf{N}^{1D}\otimes\mathbf{D}^{1D}\otimes\mathbf{I}\\ \mathbf{N}^{1D}\otimes\mathbf{D}^{1D}\otimes\mathbf{N}^{1D}\otimes\mathbf{I}\\ \mathbf{D}^{1D}\otimes\mathbf{N}^{1D}\otimes\mathbf{N}^{1D}\otimes\mathbf{I}\end{bmatrix}\boldsymbol{\mathcal{U}}^{(e,t)} \tag{13}$$

where $\mathbf{N}^{1D}$ and $\mathbf{D}^{1D}$ are $n_q \times n_p$ matrices corresponding to the one-dimensional FE basis function values and gradients, respectively, at quadrature points and $\otimes$ represents the Kronecker product. Using the well-known result of tensor algebra, $(\mathbf{A}\otimes\mathbf{B})\,(\mathbf{I}\otimes\mathbf{B})\,(\mathbf{A}\otimes\mathbf{I})$ we can reduce the above expressions into a series of tensor contractions as enunciated in Algorithm 1 below.

Algorithm 1: Evaluation of $\mathbf{T} = \mathbf{N}\mathbf{U}^{(e,t)}$

Input: $\mathbf{U}^{(e,t)}\left(\equiv\boldsymbol{\mathcal{U}}^{(e,t)}\right)$
Data: $\mathbf{N}^{1D}$
Result: $\mathbf{T}\,(\equiv\boldsymbol{\mathcal{T}})$

$\boldsymbol{\mathcal{T}} \leftarrow \left(\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{N}^{1D}\otimes\mathbf{I}\right)\boldsymbol{\mathcal{U}}^{(e,t)}$;
$\boldsymbol{\mathcal{T}} \leftarrow \left(\mathbf{I}\otimes\mathbf{N}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\right)\boldsymbol{\mathcal{T}}$;
$\boldsymbol{\mathcal{T}} \leftarrow \left(\mathbf{N}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I}\right)\boldsymbol{\mathcal{T}}$;
return $\boldsymbol{\mathcal{T}}$

Similar in spirit to Fischer et al. [11], Deville et al. [30], we now evaluate $\mathbf{K}^{(e)}\mathbf{U}^{(e)}$ by expressing $\widehat{N}^{1D}_i(\xi)$ as $\widehat{N}^{1D}_i(\xi) = \sum_{\hat{q}}\widehat{N}^{1D}_i(\xi_{\hat{q}})\widetilde{N}^{1D}_{\hat{q}}(\xi)$ where $\widetilde{N}^{1D}_{\hat{q}}$ is the Lagrange polynomial defined at the quadrature point $\xi_{\hat{q}}$. This allows us to write $\frac{d\widehat{N}^{1D}_i(\xi)}{d\xi} = \sum_{\hat{q}}\frac{d\widetilde{N}^{1D}_{\hat{q}}(\xi)}{d\xi}\widehat{N}^{1D}_i(\xi_{\hat{q}})$. Consequently, we can now factorize $\mathbf{D}^{1D}$ as $\mathbf{D}^{1D} = \widetilde{\mathbf{D}}^{1D}\mathbf{N}^{1D}$ where $\widetilde{D}^{1D}_{\hat{q}q} = \left.\frac{d\widetilde{N}^{1D}_q(\xi)}{d\xi}\right|_{\xi_{\hat{q}}}$. Equation (13) can now be rewritten as

$$\begin{bmatrix}\mathbf{D}^{(0)}\\ \mathbf{D}^{(1)}\\ \mathbf{D}^{(2)}\end{bmatrix}\mathbf{U}^{(e,t)} = \begin{bmatrix}\mathbf{I}\otimes\mathbf{I}\otimes\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\\ \mathbf{I}\otimes\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\\ \widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I}\end{bmatrix}\mathbf{N}\mathbf{U}^{(e,t)} = \begin{bmatrix}\widetilde{\mathbf{D}}^{(0)}\\ \widetilde{\mathbf{D}}^{(1)}\\ \widetilde{\mathbf{D}}^{(2)}\end{bmatrix}\mathbf{N}\mathbf{U}^{(e,t)} \tag{14}$$

Using this factorization in Eq. (14), $\mathbf{K}^{(e)}\mathbf{U}^{(e,t)}$ can be evaluated with a computational complexity of $O((4(n_p^3 n_q + n_p^2 n_q^2 + n_p n_q^3) + 12n_q^4 + 3n_q^3)n_v)$. Note that this approach reduces the floating point operations required when $n_q = n_p$ by ~30% compared to tensor contractions in Eqs. (12) and (13). Even in the case of $n_q > n_p$, this factorization is beneficial for evaluating the action of $\mathbf{A}^{(e)} = \mathbf{K}^{(e)} + \mathbf{M}^{\kappa,(e)}$ as it allows us to reduce the number of required tensor contractions by factorizing out $\mathbf{N}$ and

$\mathbf{N}^T$ as follows:

$$\begin{aligned}\mathbf{A}^{(e)}\mathbf{U}^{(e,t)} &= \left(\mathbf{K}^{(e)} + \mathbf{M}^{\kappa,(e)}\right)\mathbf{U}^{(e,t)} \\ &= \mathbf{N}^T\left(\begin{bmatrix}\widetilde{\mathbf{D}}^{(0)}\\ \widetilde{\mathbf{D}}^{(1)}\\ \widetilde{\mathbf{D}}^{(2)}\end{bmatrix}^T \begin{bmatrix}\mathcal{G}^{(0,0)} & \mathcal{G}^{(0,1)} & \mathcal{G}^{(0,2)}\\ \mathcal{G}^{(1,0)} & \mathcal{G}^{(1,1)} & \mathcal{G}^{(1,2)}\\ \mathcal{G}^{(2,0)} & \mathcal{G}^{(2,1)} & \mathcal{G}^{(2,2)}\end{bmatrix}\begin{bmatrix}\widetilde{\mathbf{D}}^{(0)}\\ \widetilde{\mathbf{D}}^{(1)}\\ \widetilde{\mathbf{D}}^{(2)}\end{bmatrix} + \mathcal{G}\right)\mathbf{N}\mathbf{U}^{(e,t)}\end{aligned} \tag{15}$$

Using Eq. (15), we describe the algorithm for the evaluation of $\mathbf{V}^{(e,t)}$ in the case of $\mathbf{A}^{(e)} = \mathbf{K}^{(e)} + \mathbf{M}^{\kappa,(e)}$ in Algorithm 2.

Algorithm 2: Evaluation of $\mathbf{V}^{(e,t)} = \left(\mathbf{K}^{(e)} + \mathbf{M}^{\kappa,(e)}\right)\mathbf{U}^{(e,t)}$

Input: $\mathbf{U}^{(e,t)}\left(\equiv \mathcal{U}^{(e,t)}\right)$
Data: $\mathbf{N}^{1D}, \widetilde{\mathbf{D}}^{1D}, \mathcal{G}, \mathcal{G}^{(s,d)}$ where $s, d = 0, 1, 2$
Temporary Variables: $\mathcal{T}, \mathcal{T}^{(0)}, \mathcal{T}^{(1)}, \mathcal{T}^{(2)}$
Result: $\mathbf{V}^{(e,t)}$

$\mathcal{T} \leftarrow \mathbf{N}\mathbf{U}^{(e,t)}$; // Algorithm 1
$\mathcal{T}^{(0)} \leftarrow \left(\mathbf{I}\otimes\mathbf{I}\otimes\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\right)\mathcal{T}$;
$\mathcal{T}^{(1)} \leftarrow \left(\mathbf{I}\otimes\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\right)\mathcal{T}$;
$\mathcal{T}^{(2)} \leftarrow \left(\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I}\right)\mathcal{T}$;
$\mathcal{T}^{(s)} \leftarrow \sum_{d=0}^{2}\mathcal{G}^{(s,d)}\mathcal{T}^{(d)}$;
$\mathcal{T} \leftarrow \mathcal{G}\mathcal{T}$;
$\mathcal{T} \leftarrow \mathcal{T} + \left(\mathbf{I}\otimes\mathbf{I}\otimes\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\right)^T\mathcal{T}^{(0)}$;
$\mathcal{T} \leftarrow \mathcal{T} + \left(\mathbf{I}\otimes\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\right)^T\mathcal{T}^{(1)}$;
$\mathcal{T} \leftarrow \mathcal{T} + \left(\widetilde{\mathbf{D}}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I}\right)^T\mathcal{T}^{(2)}$;
$\mathbf{V}^{(e,t)} \leftarrow \mathbf{N}^T\mathbf{T}$; // Algorithm 1
return $\mathbf{V}^{(e,t)}$

3. Hardware-aware implementation of the Matrix-free algorithm

This section describes the proposed hardware-aware implementation procedures on multi-node CPU and GPU architectures for evaluating FE discretized matrix-multivector products using the matrix-free algorithm discussed in the previous section. The key steps involve: (i) the extraction step in which the FE-cell level multivectors $\mathbf{U}^{(e,t)}$ of size $n_v \times n_p^3$ are constructed from the global multivector $\mathbf{U}$ of size $n_v \times m$ using the *subdomain level to FE-cell level map* and the *partitioner*, (ii) the FE-cell level evaluation in the matrix-free framework involving tensor contractions (Eq. (15)) and a point-wise multiplication to represent the action of $\mathcal{G}$ and $\mathcal{G}_{ij}$, (iii) the assembly of the output FE-cell level matrices $\mathbf{V}^{(e,t)}$ to construct output node-level multivector $\mathbf{V}$ employing the same map and partitioner used in the extraction phase. This procedure is described in more detail in the following subsections.

3.1. Mathematical formulation of batched algorithm

The proposed algorithm involves processing several batches of a small number of vectors tailored to specific hardware architecture. This approach enables better data locality owing to the smaller size of each batch and permits further parallelization over multiple batches. We denote the number of vectors in each batch as 'b' and the number of batches as 'n_b'. We now present a mathematical description of a batched algorithm for evaluating matrix-multivector products within the matrix-free paradigm on both CPU and GPU architectures.

3.1.1. CPU Batched Strategy

We propose a strategy for batched evaluation of matrix-multivector products in the case of CPUs. To this end, we introduce a Boolean sparse matrix $\mathbf{B}^{(i_b)}$ whose action on the multivector results in the extraction of the multivector batch $\mathbf{U}^{(i_b)} = \mathbf{B}^{(i_b)}\mathbf{U}$, where $\mathbf{U}^{(i_b)}$ is the multivector batch indexed by i_b (i.e. the batch comprising of vectors indexed from $i_b \times b$ to $(i_b + 1) \times b$). Using this matrix, we recast Eq. (6) as

$$\mathbf{V} = \sum_{i_b}^{n_b}\mathbf{B}^{(i_b)^T}\left(\sum_{t}^{n_t}\mathbf{P}^{(i_b,t)^T}\mathbf{C}^{(i_b,t)^T}\left(\sum_{e}^{E_t}\mathbf{Q}^{(i_b,e,t)^T}\mathbf{A}^{(e)}\mathbf{Q}^{(i_b,e,t)}\right)\mathbf{C}^{(i_b,t)}\mathbf{P}^{(i_b,t)}\right)\mathbf{B}^{(i_b)}\mathbf{U} \tag{16}$$

Here, $\mathbf{P}^{(i_b,t)}$ represents the partitioner matrix that acts on $\mathbf{U}^{(i_b)}$ and extracts its portion belonging to the subdomain Ω^t (on `MPI` task t), i.e., $\mathbf{U}^{(i_b,t)} = \mathbf{P}^{(i_b,t)}\mathbf{B}^{(i_b)}\mathbf{U}$. Constraint matrix $\mathbf{C}^{(i_b,t)}$ then acts on $\mathbf{U}^{(i_b,t)}$ to ensure that all constraints are satisfied. This results in the constrained subdomain level multivector, $\mathbf{C}^{(i_b,t)}\mathbf{U}^{(i_b,t)}$. The *subdomain level to FE-cell level map*, $\mathbf{Q}^{(i_b,e,t)}$, then acts on $\mathbf{C}^{(i_b,t)}\mathbf{U}^{(i_b,t)}$ resulting in the FE-cell level multivector batch $\mathbf{U}^{(i_b,e,t)} = \mathbf{Q}^{(i_b,e,t)}\mathbf{C}^{(i_b,t)}\mathbf{U}^{(i_b,t)}$. We then evaluate the FE-cell level matrix multivector product $\mathbf{V}^{(i_b,e,t)} = \mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$. Subsequently, we map this FE-cell level product to the subdomain level product multivector via $\mathbf{Q}^{(i_b,e,t)^T}$ and then sum over the contributions from all the FE-cells belonging to Ω^t. The transpose of $\mathbf{C}^{(i_b,t)}$ then acts on the result to ensure that the constraints are satisfied, which results in the product multivector corresponding to batch i_b and subdomain Ω^t, i.e., $\mathbf{V}^{(i_b,t)} = \mathbf{C}^{(i_b,t)^T}\left(\sum_e^{E_t}\mathbf{Q}^{(i_b,e,t)^T}\mathbf{V}^{(i_b,e,t)}\right)$. The global product multivector batch can then be evaluated by summing over the action of the $\mathbf{P}^{(i_b,t)^T}$ on all the subdomain level product multivectors corresponding to batch i_b, i.e., $\mathbf{V}^{(i_b)} = \sum_t^{n_t}\mathbf{P}^{(i_b,t)^T}\mathbf{V}^{(i_b,t)}$. This process is repeated for every batch to compute the global product vector $\mathbf{V} = \sum_{i_b}^{n_b}\mathbf{B}^{(i_b)^T}\mathbf{V}^{(i_b)}$.

3.1.2. GPU Batched Strategy

In contrast to the batched evaluation on CPU architectures discussed above, we recast Eq. (6) differently to better harness the SIMT nature of GPU architectures by further parallelizing over both FE-cells and batches. We define $\mathbf{B}^{(i_b,t)}$ to represent the Boolean sparse matrix for extracting the batch i_b of the subdomain multivector $\mathbf{U}^{(t)}$ corresponding to Ω^t and $\mathbf{Q}^{(i_b,e,t)}$ represents

the *subdomain level to FE-cell level map* for the FE-cell identified by e, batch i_b and task t. To this end, we interchange the order of the operations involved in Eq. (16) and consequently rewrite Eq. (6) as

$$\mathbf{V} = \sum_{t}^{n_t} \mathbf{P}^{(t)^T} \mathbf{C}^{(t)^T} \Bigg(\sum_{i_b}^{n_b} \sum_{e}^{E_t} \mathbf{B}^{(i_b,t)^T} \mathbf{Q}^{(i_b,e,t)^T} \mathbf{A}^{(e)} \mathbf{Q}^{(i_b,e,t)} \mathbf{B}^{(i_b,t)} \Bigg) \mathbf{C}^{(t)} \mathbf{P}^{(t)} \mathbf{U} \quad (17)$$

Recall $\mathbf{P}^{(t)}$ represents the partitioner acting on $\mathbf{U}$ to extract $\mathbf{U}^{(t)} = \mathbf{P}^{(t)}\mathbf{U}$, belonging to the subdomain Ω^t (on `MPI` task t). The constraint matrix $\mathbf{C}^{(t)}$ acts on $\mathbf{U}^{(t)}$, to ensure that all the constraints are satisfied, resulting in the constrained subdomain level multivector $\mathbf{C}^{(t)}\mathbf{U}^{(t)}$. Subsequently, we define $\mathbf{U}^{(i_b,e,t)} = \mathbf{Q}^{(i_b,e,t)}\mathbf{B}^{(i_b,t)}\mathbf{C}^{(t)}\mathbf{U}^{(t)}$ and now the steps involving the action of $\mathbf{A}^{(e)}$ on $\mathbf{U}^{(i_b,e,t)}$ to compute FE-cell level output $\mathbf{V}^{(i_b,e,t)}$ and its mapping to the subdomain level product multivector via the action of $\mathbf{B}^{(i_b,t)^T}\mathbf{Q}^{(i_b,e,t)^T}$ on $\mathbf{V}^{(i_b,e,t)}$ are accomplished collectively. We do this for every batch 'i_b' and FE-cell 'e' and sum over all batches and FE-cells to compute the subdomain level product multivector $\mathbf{V}^{(t)} = \sum_{i_b}^{n_b} \sum_{e}^{E_t} \mathbf{B}^{(i_b,t)^T}\mathbf{Q}^{(i_b,e,t)^T}\mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$. We note that the computation of summation terms necessary for evaluation of $\mathbf{V}^{(t)}$ is done concurrently for every FE-cell and batch through a single GPU kernel launch. The transpose of the constraint matrix $\mathbf{C}^{(t)^T}$, then acts on $\mathbf{V}^{(t)}$ to ensure that the constraints are satisfied. Finally, the transpose of the partitioner $\mathbf{P}^{(t)^T}$ acts on this subdomain level product multivector $\mathbf{V}^{(t)}$ to compute the global product multivector $\mathbf{V}$. Further elaboration on these operations will be provided in the subsequent discussion.

3.2. Numerical implementation strategy

Next, we delve into the computational strategies employed on CPU and GPU architectures used for the implementation of the batched algorithm discussed above. Therefore, we propose a batched layout for storing of the subdomain level multivector.

3.2.1. Data Layout: Storage of subdomain multivector

As discussed above, computations can be performed more efficiently for the matrix-free approach if the number of vectors simultaneously dealt with at a given FE node is tailored to hardware architectures, such as the SIMD vectorization width in CPUs or the shared memory size on GPUs. To this end, we propose a batched layout for storing the multivector, which we refer to as the *Batched Contiguous Vector* (BCV) layout. This BCV layout stores the nodal values of a batch of b vectors contiguously for all nodes in $n_b = \lceil n_v/b \rceil$ contiguous batches. We illustrate the layout in Fig. 1.

3.2.2. Applying the constraints: Action of $\mathbf{C}^{(t)}$ and $\mathbf{C}^{(t)^T}$

We now discuss the application of constraints, mathematically represented as the application of sparse matrices $\mathbf{C}^{(t)}$ and $\mathbf{C}^{(t)^T}$ ($\mathbf{C}^{(i_b,t)}$ and $\mathbf{C}^{(i_b,t)^T}$ in case of CPU architectures) as discussed in Sections 3.1.1 and 3.1.2. We note that the most commonly encountered constraints in non-conforming adaptively refined meshes are the hanging-node constraints [29], which are locally dense, as they involve interpolation along faces/edges. Consequently, we adopt a local dense matrix approach for applying constraints which allows for utilization of optimized BLAS level 3 routines. We store the constraints as multiple sets, and each set 'I' comprises four arrays to hold all the constraint information involving the same master nodes. The four arrays include an array containing master node indices, an array containing all the slave node indices, another consisting of the weight matrix ($\mathbf{W}_I$) for this set of constraints, and finally, an array containing the inhomogeneities corresponding to the slave nodes. A pictorial depiction of the process of application of $\mathbf{C}^{(t)}$ on a given batch of multivectors is shown in Fig. 2.

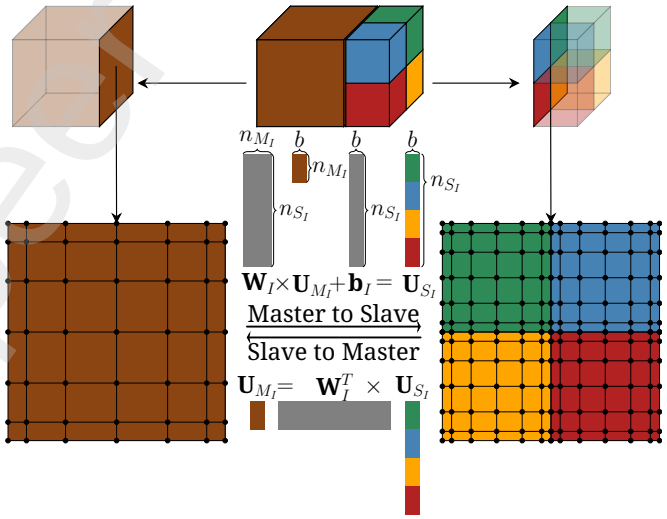


Figure 2: Pictorial depiction of the constraints strategy described in Section 3.2.2. Here $\mathbf{U}_{S_I}$ and $\mathbf{U}_{M_I}$ represent the subvectors corresponding to the slave and master nodes respectively

To this end, the nodal values corresponding to the master nodes for the given batch of multivector are extracted into a matrix $\mathbf{U}_{M_I}$, which is multiplied by the weight matrix $\mathbf{W}_I$ using BLAS `gemm` routines, and subsequently, the inhomogeneity vector $\boldsymbol{b}_I$ is added to the result. The resulting $\mathbf{U}_{S_I}$ is copied back to slave nodes of the multivector corresponding to the same batch. Hence, the application of constraints reduces to a sequence of

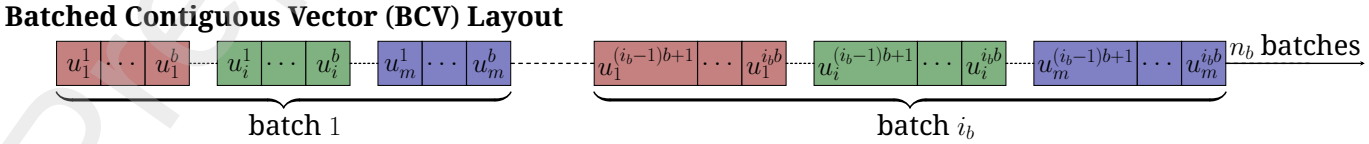


Figure 1: Pictorial depiction of the layout described in Section 3.2.1. Here $\boldsymbol{u}_J^\beta = u^\beta(\mathbf{x}_J)$ with β representing the vector index and J representing the spatial index.

dense matrix-matrix multiplications. The action of $\mathbf{C}^{(t)^T}$ is evaluated in the similar manner described above. We also apply the Dirichlet boundary conditions using this framework. In this case, the master index matrix $\mathbf{U}_{M_I}$ and weight matrix $\mathbf{W}_I$ are empty.

3.2.3. Extraction and Assembly: Action of $\mathbf{Q}^{(i_b,e,t)}$ and $\mathbf{Q}^{(i_b,e,t)^T}$

The action of the Boolean sparse matrix $\mathbf{Q}^{(i_b,e,t)}$ on the subdomain level multivector batch to *extract* $\mathbf{U}^{(i_b,e,t)}$ is implemented as a discontiguous read from $\mathbf{U}^{(i_b,t)}$ to obtain the data corresponding to the nodes within FE-cell Ω^e. Similarly, we compute the action of $\mathbf{Q}^{(i_b,e,t)^T}$ and the summation over e in Eqs. (16) and (17) (assembly step) as addition into discontiguous data. The FE-cell level multivector for batch i_b and FE-cell e is represented by

$$\texttt{U\_e\_ib}[i + bp_1 + bn_p p_2 + bn_p^2 p_3] \equiv U^{(i_b,e,t)}_{i,p_1,p_2,p_3} \qquad (18)$$
$$i = 1,\ldots,b \qquad p_1,p_2,p_3 = 1,\ldots,n_p$$

Note that the ordering of the subscript indices represents the data contiguity in memory. Further optimizations for this step on GPUs are discussed in Section 3.2.4.

3.2.4. Tensor Contractions: Evaluation of $\mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$

We now illustrate the methodology followed for the evaluation of $\mathbf{V}^{(i_b,e,t)} = \mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$ for the specific case of $\mathbf{A}^{(e)} = \mathbf{K}^{(e)} + \mathbf{M}^{\kappa,(e)}$ using Algorithms 1 and 2.

Note that in both Algorithms 1 and 2 we need to evaluate products of the forms $(\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{Y}\otimes\mathbf{I})\,\mathbf{U}^{(i_b,e,t)}$, $(\mathbf{I}\otimes\mathbf{Y}\otimes\mathbf{I}\otimes\mathbf{I})\,\mathbf{U}^{(i_b,e,t)}$ and $(\mathbf{Y}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I})\,\mathbf{U}^{(i_b,e,t)}$ using the tensor product vec-trick, $(\mathbf{Y}\otimes\mathbf{Z})\,\mathrm{vec}\,(\mathbf{X}) = \mathrm{vec}\left(\mathbf{Y}^T\mathbf{X}\mathbf{Z}\right)$ where vec($\mathbf{X}$) denotes the vectorization of the matrix $\mathbf{X}$ by stacking the columns of $\mathbf{X}$ into a single column vector, we rewrite these products as batched matrix-matrix multiplications. For instance, let $\mathcal{R}$ and $\mathcal{T}$ be fourth-order tensors and the dimension of $\mathcal{R}$ be $b\times n_p\times n_p\times n_p$. If $\mathbf{Y}$ is an $n_q\times n_p$ matrix, then we have

- $\mathcal{T}\leftarrow(\mathbf{Y}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I})\,\mathcal{R}$: Treating $\mathcal{R}_{\beta,p_1,p_2,p_3}$ and $\mathcal{T}_{\beta,p_1,p_2,q_3}$ as matrices $\mathbf{R}$ of dimensions $(bn_p^2\times n_p)$ and $\mathbf{T}$ of dimensions $(bn_p^2\times n_q)$ respectively, we write

$$\mathbf{T}\leftarrow\mathbf{R}\mathbf{Y}^T \qquad (19)$$

- $\mathcal{T}\leftarrow(\mathbf{I}\otimes\mathbf{Y}\otimes\mathbf{I}\otimes\mathbf{I})\,\mathcal{R}$: Treating $\mathcal{R}_{\beta,p_1,p_2,p_3}$ and $\mathcal{T}_{\beta,p_1,q_2,p_3}$ as sets of matrices $\mathbf{R}_{p_3}$ of dimensions $(bn_p\times n_p)$ and $\mathbf{T}_{p_3}$ of dimensions $(bn_p\times n_q)$ respectively, where $p_3 = 1,2,\ldots,n_p$ we write

$$\mathbf{T}_{p_3}\leftarrow\mathbf{R}_{p_3}\mathbf{Y}^T \qquad \forall p_3 = 1,\ldots,n_p \qquad (20)$$

- $\mathcal{T}\leftarrow(\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{Y}\otimes\mathbf{I})\,\mathcal{R}$: Treating $\mathcal{R}_{\beta,p_1,p_2,p_3}$ and $\mathcal{T}_{\beta,q_1,p_2,p_3}$ as sets of matrices $\mathbf{R}_{p_2,p_3}$ of dimensions $(b\times n_p)$ and $\mathbf{T}_{p_2,p_3}$ of dimensions $(b\times n_q)$ respectively, where $p_2,p_3 = 1,2,\ldots,n_p$ we write

$$\mathbf{T}_{p_2,p_3}\leftarrow\mathbf{R}_{p_2,p_3}\mathbf{Y}^T \qquad \forall p_2,p_3 = 1,\ldots,n_p \qquad (21)$$

The other major part of Algorithm 2 is the evaluation of $\mathcal{G}\mathcal{T}$ and $\sum_d\mathcal{G}^{(s,d)}\mathcal{T}^{(d)}$. To evaluate these products we redefine the $\mathbf{N}^{1D}$ and the $\widetilde{\mathbf{D}}^{1D}$ matrices as $N^{1D}_{q,p}\leftarrow N^{1D}_{q,p}\sqrt{w^{1D}_q}$ and $\widetilde{D}^{1D}_{q_1,q_2}\leftarrow\widetilde{D}^{1D}_{q_1,q_2}\sqrt{w^{1D}_{q_1}/w^{1D}_{q_2}}$ where w^{1D}_q are the 1D quadrature weights, as discussed in Section 2.2. This allows us to evaluate $\mathcal{G}\mathcal{T}$ and $\sum_d\mathcal{G}^{(s,d)}\mathcal{T}^{(d)}$ in the following manner

- $\mathcal{T}\leftarrow\mathcal{G}\mathcal{T}$: Considering $\boldsymbol{\kappa}^{(e)}$ to be the vector of length n_q^3 defined as $\kappa^{(e)}_Q = \kappa\left(\mathbf{x}^{(e)}_Q\right)\ \forall Q = 1,\ldots,n_q^3$ we can evaluate $\mathcal{G}\mathcal{T}$ as $\det\mathbf{J}^{(e)}\boldsymbol{\kappa}^{(e)}\circ\mathcal{T}$ where $\circ$ represents the batched Hadamard product defined as

$$\mathcal{T}_{\beta,Q}\leftarrow\det\mathbf{J}^{(e)}\kappa^{(e)}_Q\mathcal{T}_{\beta,Q}\quad\forall\beta = 1,\ldots,b \qquad (22)$$

  Note that this reduces to matrix scaling in the case of the Helmholtz operator as $\kappa(\mathbf{x})$ is a constant.

- $\mathcal{T}^{(s)}\leftarrow\sum_d\mathcal{G}^{(s,d)}\mathcal{T}^{(d)}\ \forall s = 0,1,2$: Defining a $bn_q^3\times 3$ matrix as $\left[\mathcal{T}^{(0)}\mathcal{T}^{(1)}\mathcal{T}^{(2)}\right]$ we can write this operation as a $bn_q^3\times 3$ times 3×3 matrix-matrix multiplication as

$$\left[\mathcal{T}^{(0)}\mathcal{T}^{(1)}\mathcal{T}^{(2)}\right]\left(\left(\mathbf{J}^{(e)}\right)^{-1}\left(\mathbf{J}^{(e)}\right)^{-T}\det\mathbf{J}^{(e)}\mu\right) \qquad (23)$$

We now discuss the implementation of the above algorithm on CPU and GPU architectures.

CPU Implementation: Evaluation of $\mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$

The implementation strategy used for the batch-wise evaluation of $\mathbf{V}^{(i_b,t)}$ on CPU architectures, including constraints, extraction, and assembly, is described in Algorithm 3. To perform the strided-batched matrix-matrix multiplications in Algorithm 3 (described by Eqs. (19) to (21)), we would need to have a function with the following signature

```
template <int m, int n, int k, int c,
     bool add, bool trans>
inline void
matmul(const double *E, const double
    *F, double *C)
```

Listing 1: function signature for strided batched matrix-matrix multiplication

which evaluates $\mathbf{C}_i = \mathbf{E}_i\mathrm{op}\,(\mathbf{F}) + \beta\mathbf{C}_i\quad\forall i = 1,\ldots,c$ where $\mathbf{E}_i$ is an $m\times k$ matrix and $\mathrm{op}\,(\mathbf{F})$ is a $k\times n$ matrix with $\beta = 1$ if `add=true` (0 otherwise) and $\mathrm{op}\,(\mathbf{F}) = \mathbf{F}^T$ if `trans=true` ($\mathbf{F}$ otherwise). To evaluate these batched matrix-matrix products involving $\mathbf{N}^{1D}$ and $\widetilde{\mathbf{D}}^{1D}$ we explored three strategies:

1. Employ `JIT` (Just-In-Time) modules from Intel® MKL version 2022.1.0 [31]. For this implementation, $b = 20$ yielded the best performance.
2. Handwritten matrix-matrix multiplication code using AVX-512 intrinsics to work with 8 vectors concurrently, i.e. $b = 8$.

3. Exploit the symmetry of the shape functions and quadrature points to reduce the floating point operations required by half via the *even-odd* decomposition [25, 26], and use AVX-512 intrinsics to work with eight vectors concurrently, i.e., $b = 8$. An illustration of the even-odd implementation strategy to evaluate $\left(\mathbf{N}^{1D} \otimes \mathbf{I} \otimes \mathbf{I} \otimes \mathbf{I}\right) \boldsymbol{\mathcal{U}}^{(i_b,e,t)}$ is provided in Fig. 4 and sample code for the same is provided in Listing 2.

To compare the three strategies, we computed the action of the FE discretized Helmholtz operator obtained by setting $\mu = 1$ and $\kappa(\mathbf{x}) = 2\pi; \forall \mathbf{x} \in \Omega$ in Eq. (1) on randomly generated multivectors. The results of our explorations are shown in Fig. 3. We find that the *even-odd* decomposition approach yielded the best performance. On a single core, using the BCV layout, we observe speedups of up to 40% of the *Even-Odd AVX-512 Multivector* implementation over the *MKL JIT Multivector*. We attribute this speedup to the fact that MKL JIT does not appear to use AVX-512 for matrices of such dimensions and instead falls back to AVX2.

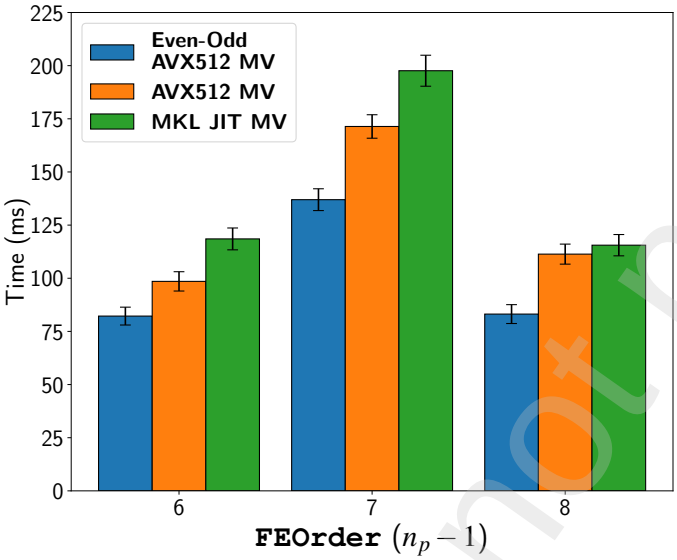


Figure 3: Performance benchmarks of our matrix-free implementation strategies using the proposed BCV layout on a single core of Intel® Xeon® Gold 6248R processor. Benchmark case studies: 15625 DoFs ($n_q = n_p = 7, 9$); 24389 DoFs ($n_q = n_p = 8$). For AVX512 implementations $b = 8$ is chosen and for the MKL JIT implementation $b = 20$ is chosen.

Algorithm 3: Batchwise evaluation of $\mathbf{V}$ on CPUs

Input: $\mathbf{U}$
Data: $\mathbf{B}^{(i_b)}, \mathbf{P}^{(i_b,t)}, \mathbf{C}^{(i_b,t)}, \mathbf{Q}^{(i_b,e,t)}, \mathbf{N}^{1D}, \mathbf{D}^{1D}, \mathbf{J}^{e}, \kappa$
Temporary Variables: $\mathbf{T}, \mathbf{T}^{(0)}, \mathbf{T}^{(1)}, \mathbf{T}^{(2)}$
Result: $\mathbf{V}$
`MPI` **rank:** t

$\mathbf{U}^{(i_b,t)} \leftarrow \mathbf{C}^{(i_b,t)}\mathbf{U}^{(i_b,t)}$; `// Section 3.2.2`
for $e \leftarrow 1$ **to** E_t **do**
  $\mathbf{T} \leftarrow \mathbf{Q}^{(i_b,e,t)}\mathbf{U}^{(i_b,t)}$; `// Section 3.2.3`
  $\underset{(bn_p^2 \times n_q)}{\mathbf{T}} \leftarrow \underset{(bn_p^2 \times n_p)(n_p \times n_q)}{\mathbf{T}\ \mathbf{N}^{1D^T}}$; `// Eq. (19)`
  for $q \leftarrow 1$ **to** n_q **do**
    $\underset{(bn_p \times n_q)}{\mathbf{T}_q} \leftarrow \underset{(bn_p \times n_p)(n_p \times n_q)}{\mathbf{T}_q\ \mathbf{N}^{1D^T}}$; `// Eq. (20)`
  for $q \leftarrow 1$ **to** n_q^2 **do**
    $\underset{(b \times n_q)}{\mathbf{T}_q} \leftarrow \underset{(b \times n_p)(n_p \times n_q)}{\mathbf{T}_q\ \mathbf{N}^{1D^T}}$; `// Eq. (21)`
  $\underset{(bn_q^2 \times n_q)}{\mathbf{T}^{(2)}} \leftarrow \underset{(bn_q^2 \times n_q)(n_q \times n_q)}{\mathbf{T}\ \widetilde{\mathbf{D}}^{1D^T}}$; `// Eq. (19)`
  for $q \leftarrow 1$ **to** n_q **do**
    $\underset{(bn_q \times n_q)}{\mathbf{T}_q^{(1)}} \leftarrow \underset{(bn_q \times n_q)(n_q \times n_q)}{\mathbf{T}_q\ \widetilde{\mathbf{D}}^{1D^T}}$; `// Eq. (20)`
  for $q := 1$ **to** n_q^2 **do**
    $\underset{(b \times n_q)}{\mathbf{T}_q^{(0)}} \leftarrow \underset{(b \times n_q)(n_q \times n_q)}{\mathbf{T}_q\ \widetilde{\mathbf{D}}^{1D^T}}$; `// Eq. (21)`
  $\underset{(bn_q^3 \times 3)}{\left[\mathbf{T}^{(0)}\ \mathbf{T}^{(1)}\ \mathbf{T}^{(2)}\right]} \leftarrow$
  $\underset{(bn_q^3 \times 3)}{\left[\mathbf{T}^{(0)}\ \mathbf{T}^{(1)}\ \mathbf{T}^{(2)}\right]}\underset{(3 \times 3)}{\left(\mathbf{J}^{(e)-1}\mathbf{J}^{(e)-T}\det\mathbf{J}^{(e)}\mu\right)}$; `// Eq. (23)`
  $\underset{(bn_q^3)}{\mathbf{T}} \leftarrow \underset{(n_q^3)}{\left(\det\mathbf{J}^{(e)}\boldsymbol{\kappa}\right)} \circ \underset{(bn_q^3)}{\mathbf{T}}$; `// Eq. (22)`
  $\underset{(bn_q^2 \times n_q)}{\mathbf{T}} \leftarrow \underset{(bn_q^2 \times n_q)(bn_q^2 \times n_q)(n_q \times n_q)}{\mathbf{T} + \mathbf{T}^{(2)}\ \widetilde{\mathbf{D}}^{1D}}$; `// Eq. (19)`
  for $q \leftarrow 1$ **to** n_q **do**
    $\underset{(bn_q \times n_q)}{\mathbf{T}_q} \leftarrow \underset{(bn_q \times n_q)(bn_q \times n_q)(n_q \times n_q)}{\mathbf{T}_q + \mathbf{T}_q^{(1)}\ \widetilde{\mathbf{D}}^{1D}}$; `// Eq. (20)`
  for $q \leftarrow 1$ **to** n_q^2 **do**
    $\underset{(b \times n_q)}{\mathbf{T}_q} \leftarrow \underset{(b \times n_q)(b \times n_q)(n_q \times n_q)}{\mathbf{T}_q + \mathbf{T}_q^{(0)}\ \widetilde{\mathbf{D}}^{1D}}$; `// Eq. (21)`
  $\underset{(bn_q^2 \times n_p)}{\mathbf{T}} \leftarrow \underset{(bn_q^2 \times n_q)(n_q \times n_p)}{\mathbf{T}\ \mathbf{N}^{1D}}$; `// Eq. (19)`
  for $p \leftarrow 1$ **to** n_p **do**
    $\underset{(bn_q \times n_p)}{\mathbf{T}_p} \leftarrow \underset{(bn_q \times n_q)(n_q \times n_p)}{\mathbf{T}_p\ \mathbf{N}^{1D}}$; `// Eq. (20)`
  for $p \leftarrow 1$ **to** n_p^2 **do**
    $\underset{(b \times n_p)}{\mathbf{T}_p} \leftarrow \underset{(b \times n_q)(n_q \times n_p)}{\mathbf{T}_p\ \mathbf{N}^{1D}}$; `// Eq. (21)`
  $\mathbf{V}^{(i_b,t)} \leftarrow \mathbf{V}^{(i_b,t)} + \mathbf{Q}^{(i_b,e,t)^T}\mathbf{T}$; `// Section 3.2.3`
$\mathbf{V}^{(i_b,t)} \leftarrow \mathbf{C}^{(i_b,t)^T}\mathbf{V}^{(i_b,t)}$; `// Section 3.2.2`
return $\mathbf{V}$

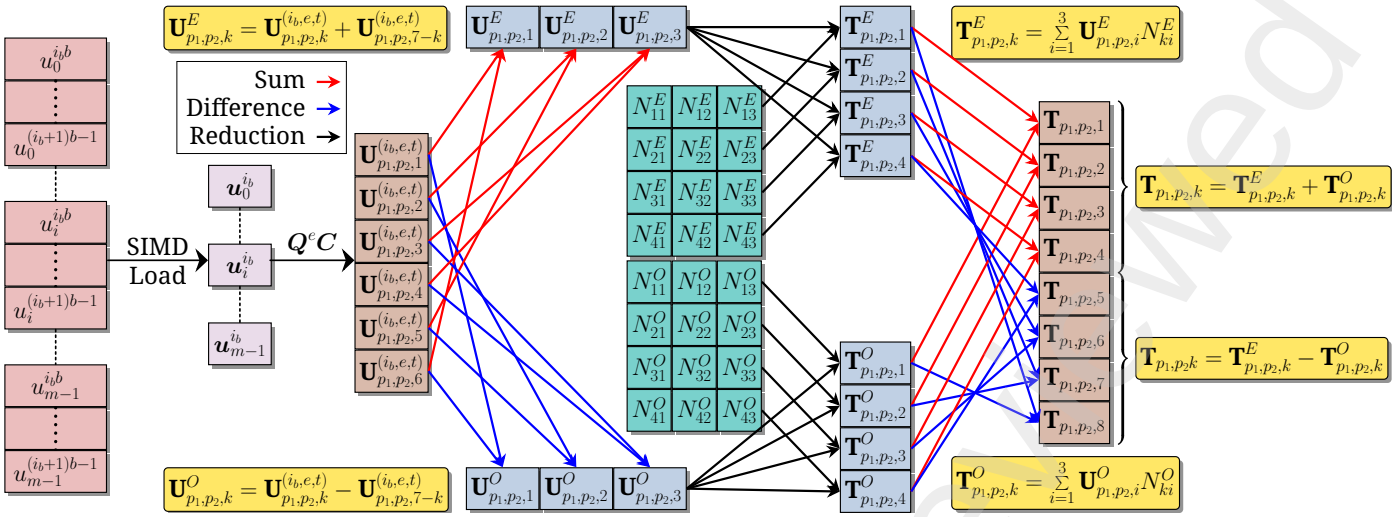


Figure 4: Evaluation of $\left(\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{N}^{1D}\right)\boldsymbol{\mathcal{U}}^{(i_b,e)}$ on CPU architectures using the even-odd decomposition strategy. An example with $n_p = 6$ and $n_q = 8$, each block in $\mathbf{U}$ represents an n_p^2 sized array of AVX-512 doubles, which are decomposed into even and odd components to be multiplied by the corresponding shape function matrices. The results are combined to form $\mathbf{T}$.

```
1 template <int m, int n, int k>
2 inline void
3 matmul(const __m512d *A, const __m512d
      *B, __m512d *C){
4 /*Here m = n_p^2, n = n_q, k = n_p                    */
5 /*and A ← U^(i_b,e,t), B ← [N^E N^O], C ← U^(i_b,e,t) N^1D^T */
6 constexpr int ko = k / 2;
7 constexpr int no = n / 2;
8 for (auto i = 0; i < m; ++i){
9     /*Temporary arrays for storage of
          even and odd components of
          rows of A*/
10    __m512d tempAe[ko], tempAo[ko];
11    /*Evaluate even and odd
          components of row i (= p1 + np p2)
          of A*/
12    for (auto q = 0; q < ko; ++q){
13        /* tempAe[q]=U^(i_b,e,t)_(p1,p2,q) + U^(i_b,e,t)_(p1,p2,k-q) */
14        tempAe[q] = A[i + q * m] + A[
              i + (k - 1 - q) * m];
15        /* tempAo[q]=U^(i_b,e,t)_(p1,p2,q) - U^(i_b,e,t)_(p1,p2,k-q) */
16        tempAo[q] = A[i + q * m] - A[
              i + (k - 1 - q) * m];}
17    for (auto j = 0; j < no; ++j){
18    /*Temporary storage even and odd
          components of C*/
19    __m512d tempCe, tempCo;
20    /*tempCe = sum_q tempAe[q] N^E_(j,q)  */
21    tempCe = tempAe[0] * B[j];
22    for (auto q = 1; q < ko; ++q)
23        tempCe += tempAe[q] * B[j + q
              * no];
24    /*tempCo = sum_q tempAo[q] N^O_(j,q)  */
25    tempCo = tempAo[0] * B[j + ko *
          no];
26    for (auto q = 1; q < ko; ++q)
27        tempCo += tempAo[q] * B[j + q
              * no + ko * no];
28    /*Recombining tempCe and tempCo
          to get elements of C */
29    /*T_(p1,p2,j) = tempCe + tempCo */
30    C[i + m * j]           = tempCe +
          tempCo;
31    /*T_(p1,p2,n-j) = tempCe - tempCo*/
32    C[i + m * (n - 1 - j)] = tempCo -
          tempCe;}}}
```

Listing 2: Code snippet for the evaluation of $\left(\mathbf{N}^{1D}\otimes\mathbf{I}\otimes\mathbf{I}\otimes\mathbf{I}\right)\mathbf{U}^{(i_b,e,t)}$ using the even-odd decomposition strategy. Note that this snippet is purely used to illustrate the implementation strategy, and as such `k` and `n` are assumed to be even. The actual implementation is generic. Note that if $n = k$, then `C = A` is allowed, which results in a lower memory footprint.

GPU Implementation: Evaluation of $\mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$

The implementation strategy, including extraction, and assembly, used for the evaluation of $\mathbf{V}^{(t)}$ on GPU architectures is described in Algorithm 4. To evaluate the tensor contractions in Eqs. (19) to (21), vendor optimized *gemm* libraries (for eg. *cuBLAS*, *hipBLAS*) modules seem to be a natural choice at first glance for GPUs. However, the sequential library calls for each tensor contraction requires multiple reads from and writes to the device memory. Hence, to avoid such data movement, we design a shared memory gemm implementation on GPUs by taking advantage of kernel fusion, accessing data only once from the device memory. This implementation combines the extraction, tensor contractions, and assembly steps in one kernel, performs

all the computations inside the fast shared memory to finally write the data back to the device memory.

In Fig. 5, we compare these two strategies, one using `cuBLAS dgemm` and the other using the shared memory implementation as discussed above. We observe speedups of 4x – 5x for the proposed *Multivector GPU Matrix-Free* (MV GPU Matrix-Free) approach compared to the `cuBLAS dgemm` approach.

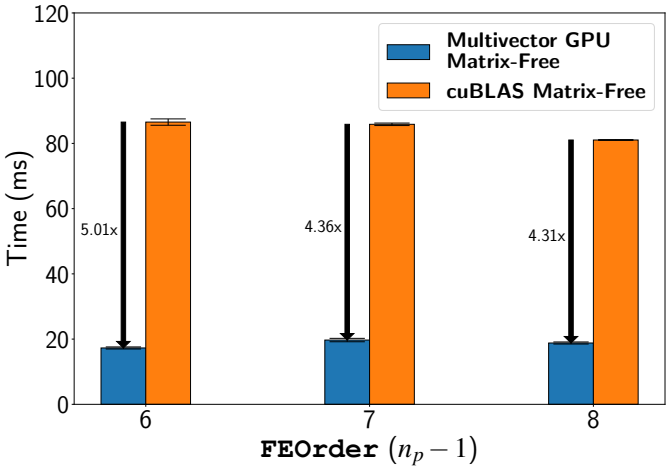


Figure 5: Performance benchmark of `cuBLAS dgemm` matrix-free implementation with our *Multivector GPU Matrix-Free* approach for evaluating tensor contractions. Studies conducted on NVIDIA® Tesla® V100 SXM2 16GB (Summit Supercomputer). GPU benchmark case studies: 117649 DoFs (`FEOrder` = 6, 8); 125000 DoFs (`FEOrder` = 7) with BCV layout where $b = n_v = 1024$.

We discuss the shared memory implementation on GPUs in more detail. Unlike matrix-multivector multiplication using the FE-cell level local dense matrices approach (see Section 2.2.1), the shared memory kernel does not explicitly construct the cell level multivectors $\mathbf{U}^{(e,t)}$ and $\mathbf{V}^{(e,t)}$ in the device memory. This helps to further reduce the memory footprint. The kernel launch associated with the shared memory implementation of our MV GPU Matrix-Free approach is as follows:

```
1 compute <b, n_p, n_q> <<<dim3(E_t, n_b),
    dim3(n_tx, n_ty), smem>>> (double *C,
    const double *A, const double *B,
    ...)
```

Listing 3: Kernel launch for Multivector GPU Matrix-Free implementation

The kernel is templated with b, n_p and n_q and launched with a 2-D grid of $E_t \times n_b$ thread blocks, each with a 2-D block of $n_{t_x} = b$ threads in the x-direction and $n_{t_y} = \texttt{warpSize} \times \alpha$ threads in the y-direction where `warpSize` = 32 for NVIDIA GPUs and α is a tunable parameter. This choice of n_{t_y} ensures that the total number of threads per thread block is a multiple of `warpSize`. The kernel is also configured with a dynamic shared memory of `smem` = $4bn_q^3$.

The matrices $\mathbf{N}^{1D}$ and $\widetilde{\mathbf{D}}^{1D}$ are stored in constant memory as they are constant for all cells and batches. This helps reduce shared memory usage, and the matrices can be reused for all subsequent tensor contractions. We rewrite these tensor contractions as batched-matrix-matrix multiplications as discussed in Section 3.2.4 and execute them as linear combinations of columns of $\mathbf{N}^{1D}$ and $\widetilde{\mathbf{D}}^{1D}$ as illustrated in Fig. 6. Thus, evaluations like Eq. (19) can be written as

$$\mathcal{T}_{t_x,t_y,q} = \sum_{k=1}^{n_p} \mathcal{R}_{t_x,t_y,k} N^{1D}_{qk} \qquad \forall q = 1, \ldots, n_p \tag{24}$$

where t_x is `threadIdx.x` and t_y is `threadIdx.y`. This execution method enables us to combine the extraction and the first tensor contraction steps. Thus, the floating point operations are performed as soon as a portion of $\mathbf{U}^{(t)}$ is read from the device memory, without having to wait for its complete bn_p^3 data to be loaded inside the shared memory. Furthermore, as each thread accesses the same values from $\mathbf{N}^{1D}$ and $\widetilde{\mathbf{D}}^{1D}$, the accesses are broadcast, and because the access is from constant memory, the GPU pipelines are better utilized. To further improve performance, we utilize registers to keep the data local to each thread as much as possible. This optimization reduces data movement from shared memory and bank conflicts. Finally, in the assembly step `atomicAdd` is used to avoid race conditions and safely assemble the output $\mathbf{V}^{(t)}$. Similar to the extraction step, we also combine the assembly step with the last tensor contraction. Unlike CPUs, we do not employ the *even-odd* decomposition on GPUs because the stall wait state of the GPU warps increases due to the requirement of additional accesses to compute the even and odd components in the *even-odd* decomposition approach (see Fig. 4 and Listing 2).

```
1   /* Snippet for Eq. (21) */
2   /* m = n_p^2, n = n_q, k = n_p */
3   /* A ← R, B ← N^{1D^T}, C ← T = RN^{1D^T} */
4   for (int i = threadIdx.y; i < m;
        i += blockDim.y) {
5     /* Temporary arrays for storage
          of rows of A and C*/
6     double y[n], x[k];
7     for (int j = 0; j < n; j++)
8       y[j] = 0.0;
9     /* x[q] = R_{t_x,i,q} */
10    for (int q = 0; q < k; q++) {
11      x[q] = A[threadIdx.x + i * b
            + q * b * m];
12      /* y[j] = Σ_q^{n_p} R_{t_x,i,q} N_{j,q} */
13      for (int j = 0; j < n; j++)
14        y[j] += B[j + q * n] * x[q
              ]; }
15    /* T_{t_x,i,j} = y[j] */
16    for (int j = 0; j < n; j++)
17      C[threadIdx.x + i * b + j * b
            * m] = y[j]; }
18
19  /* Snippet for Eq. (20) */
20  /* m = n_p^2, n = n_q, k = n_p */
21  /* A ← R_v, B ← N^{1D^T},
        C ← T_v = R_v N^{1D^T}  ∀v = 1, ..., n_p */
```

```
22    for (int i = threadIdx.y; i < m;
          i += blockDim.y) {
23      /* Temporary arrays for storage
           of rows of A and C*/
24      double y[n], x[k];
25      int u = i % k, v = i / k;
26      for (int j = 0; j < n; j++)
27        y[j] = 0.0;
28      /* x[q] = R_{t_x,u,q,v}  */
29      for (int q = 0; q < k; q++) {
30        x[q] = A[threadIdx.x + u * b
              + q * b * k + v * b * k
              ^2];
31        /* y[j] = sum_q^{n_p} R_{t_x,u,q,v} N_{j,q}  */
32        for (int j = 0; j < n; j++)
33          y[j] += B[j + q * n] * x[q
                ]; }
34      /* T_{t_x,u,j,v} = y[j]  */
35      for (int j = 0; j < n; j++)
36        C[threadIdx.x + u * b + j * b
             * k + v * b * k * n] = y[
            j]; }
37
38    /* Snippet for Eq. (19) */
39    /* m = n_q^2, n = n_q, k = n_p  */
40    /* A ← R_v, B ← N^{1D^T},
         C ← T_v = R_v N^{1D^T}   ∀v = 1,...,n_p^2  */
41    for (int i = threadIdx.y; i < m;
          i += blockDim.y) {
42      /* Temporary arrays for storage
           of rows of A and C*/
43      double y[n], x[k];
```

```
44      for (int j = 0; j < n; j++)
45        y[j] = 0.0;
46      /* x[q] = R_{t_x,q,i}  */
47      for (int q = 0; q < k; q++) {
48        x[q] = A[threadIdx.x + q * b
              + i * b * k];
49        /* y[j] = sum_q^{n_p} R_{t_x,q,i} N_{j,q}  */
50        for (int j = 0; j < n; j++)
51          y[j] += B[j + q * n] * x[q
                ]; }
52      /* T_{t_x,j,i} = y[j]  */
53      for (int j = 0; j < n; j++)
54        C[threadIdx.x + j * b + i * b
             * n] = y[j]; }
```

Listing 4: Code snippets for the evaluation of Eqs. (19) to (21) on GPUs.

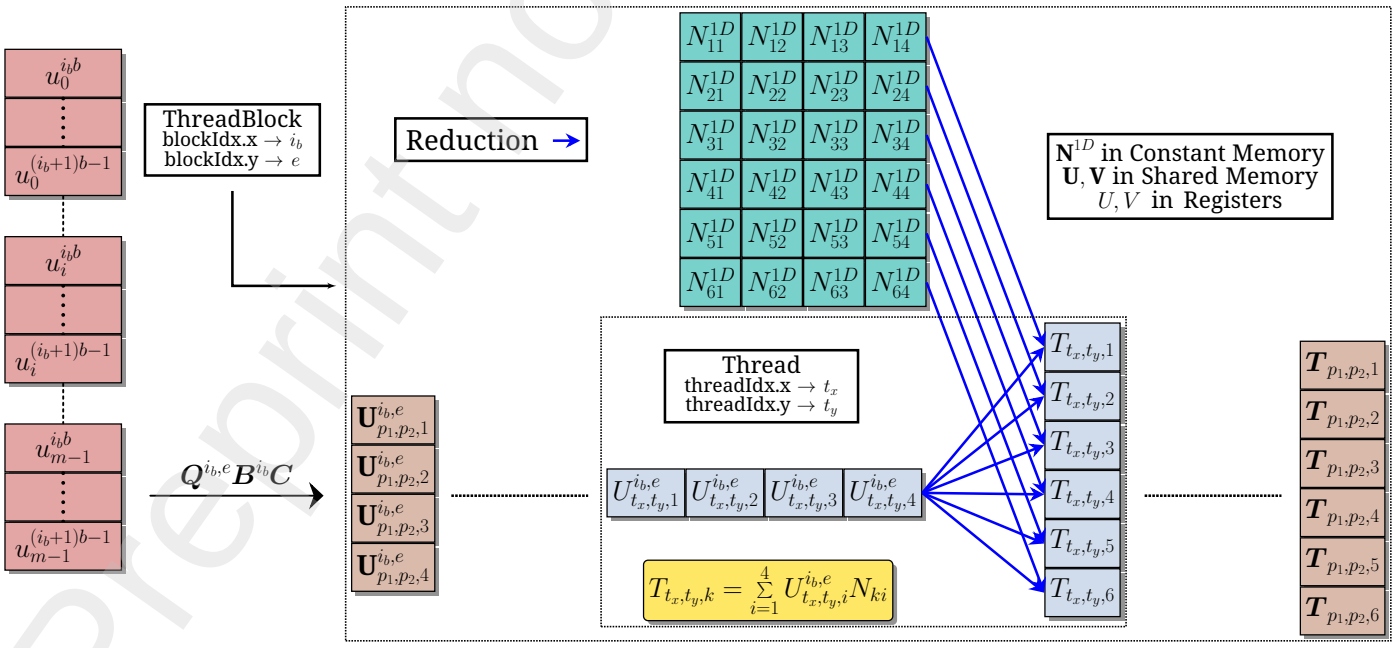


Figure 6: Pictorial depiction of tensor contractions done on GPUs. The extraction and first tensor contraction steps of evaluation of $\mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$ are depicted for the case of $n_p = 4$ and $n_q = 6$. Each block in $\mathbf{U}$ represents n_p^2 sized array of b doubles.

Algorithm 4: Batchwise evaluation of $\mathbf{V}$ on GPUs

Input: $\mathbf{U}$

Data: $\mathbf{C}^{(t)}, \mathbf{B}^{(i_b,t)}, \mathbf{Q}^{(i_b,e,t)}, \mathbf{N}^{1D}, \mathbf{D}^{1D}, , \mathbf{J}^e, \boldsymbol{\kappa}$

Temporary Variables: $\mathbf{T}, \mathbf{T}^{(0)}, \mathbf{T}^{(1)}, \mathbf{T}^{(2)}$

Result: $\mathbf{V}$

`MPI` **rank:** t

`blockIdx.x`**:** e

`blockIdx.y`**:** i_b

$\mathbf{U}^{(t)} \leftarrow \mathbf{C}^{(t)}\mathbf{U}^{(t)}$; `// Section 3.2.2`

`/*` **Device kernel compute starts** `*/`

$\underset{(bn_p^2 \times n_q)}{\mathbf{T}} \leftarrow \underset{(bn_p^2 \times n_p)}{\mathbf{Q}^{(i_b,e,t)}} \underset{(n_p \times n_q)}{\mathbf{B}^{(i_b,t)}} \mathbf{U}^{(t)} \mathbf{N}^{1D^T}$;

`// Sections 3.2.1 and 3.2.3 and Eq. (19)`

$\underset{(bn_p \times n_q)}{\mathbf{T}_q^{(0)}} \leftarrow \underset{(bn_p \times n_p)(n_p \times n_q)}{\mathbf{T}_q} \mathbf{N}^{1D^T} \; \forall q = 1, \ldots, n_q$; `// Eq. (20)`

$\underset{(b \times n_q)}{\mathbf{T}_q} \leftarrow \underset{(b \times n_p)(n_p \times n_q)}{\mathbf{T}_q^{(0)}} \mathbf{N}^{1D^T} \quad \forall q = 1, \ldots, n_q^2$; `// Eq. (21)`

$\underset{(bn_q^2 \times n_q)}{\mathbf{T}^{(2)}} \leftarrow \underset{(bn_q^2 \times n_q)(n_q \times n_q)}{\mathbf{T}} \widetilde{\mathbf{D}}^{1D^T}$; `// Eq. (19)`

$\underset{(bn_q \times n_q)}{\mathbf{T}_q^{(1)}} \leftarrow \underset{(bn_q \times n_q)(n_q \times n_q)}{\mathbf{T}_q} \widetilde{\mathbf{D}}^{1D^T} \; \forall q = 1, \ldots, n_q$; `// Eq. (20)`

$\underset{(b \times n_q)}{\mathbf{T}_q^{(0)}} \leftarrow \underset{(b \times n_q)(n_q \times n_q)}{\mathbf{T}_q} \widetilde{\mathbf{D}}^{1D^T} \quad \forall q = 1, \ldots, n_q^2$; `// Eq. (21)`

$\underset{(bn_q^3 \times 3)}{\left[\mathbf{T}^{(0)}\ \mathbf{T}^{(1)}\ \mathbf{T}^{(2)}\right]} \leftarrow \underset{(bn_q^3 \times 3)}{\left[\mathbf{T}^{(0)}\ \mathbf{T}^{(1)}\ \mathbf{T}^{(2)}\right]} \underset{(3 \times 3)}{\left(\mathbf{J}^{(e)^{-1}} \mathbf{J}^{(e)^{-T}} \det \mathbf{J}^{(e)} \mu\right)}$;

$\underset{(bn_q^2 \times n_q)}{\mathbf{T}} \leftarrow \underset{(n_q^3)}{\left(\det \mathbf{J}^{(e)} \boldsymbol{\kappa}\right)} \circ \underset{(bn_q^3)}{\mathbf{T}} + \underset{(bn_q^2 \times n_q)(n_q \times n_q)}{\mathbf{T}^{(2)}} \widetilde{\mathbf{D}}^{1D}$;

`// Eq. (19)`

$\underset{(bn_q \times n_q)}{\mathbf{T}_q} \leftarrow \underset{(bn_q \times n_q)}{\mathbf{T}_q} + \underset{(bn_q \times n_q)(n_q \times n_q)}{\mathbf{T}_q^{(1)}} \widetilde{\mathbf{D}}^{1D} \quad \forall q = 1, \ldots, n_q$;

`// Eq. (20)`

$\underset{(b \times n_q)}{\mathbf{T}_q} \leftarrow \underset{(b \times n_q)}{\mathbf{T}_q} + \underset{(b \times n_q)(n_q \times n_q)}{\mathbf{T}_q^{(0)}} \widetilde{\mathbf{D}}^{1D} \quad \forall q = 1, \ldots, n_q^2$;

`// Eq. (21)`

$\underset{(bn_q^2 \times n_p)}{\mathbf{T}} \leftarrow \underset{(bn_q^2 \times n_q)(n_q \times n_p)}{\mathbf{T}} \mathbf{N}^{1D}$; `// Eq. (19)`

$\underset{(bn_q \times n_p)}{\mathbf{T}_p} \leftarrow \underset{(bn_q \times n_q)(n_q \times n_p)}{\mathbf{T}_p} \mathbf{N}^{1D} \quad \forall p = 1, \ldots, n_p$; `// Eq. (20)`

$\underset{(b \times n_q)}{\mathbf{V}^{(i_b,t)}} \leftarrow \mathbf{V}^{(i_b,t)} + \mathbf{B}^{(i_b,t)^T} \mathbf{Q}^{(i_b,e,t)^T} \underset{(b \times n_q)(n_q \times n_p)}{\mathbf{T}_p} \mathbf{N}^{1D}$

$\forall p = 1, \ldots, n_p^2$; `// Sections 3.2.1 and 3.2.3 and Eq. (21)`

`/*` **Device kernel compute ends** `*/`

$\mathbf{V}^{(t)} \leftarrow \mathbf{C}^{(t)^T} \mathbf{V}^{(t)}$; `// Section 3.2.2`

return $\mathbf{V}$

3.2.5. *Distributed Parallelism: `MPI` aspects*

We now discuss the `MPI` communication strategies employed to reduce the communication overheads encountered when deploying on multi-node CPU and GPU architectures.

CPU Implementation: `MPI` aspects

It is important to note that in our implementation, we do not explicitly construct $\mathbf{B}^{(i_b)}\mathbf{U}$ in memory. Instead, we evaluate the action of $\mathbf{P}^{(i_b,t)}$ on it through `MPI` communication of boundary data for the multivectors across tasks that share subdomain boundaries. In addition, we evaluate the summations over e and i_b in Eq. (16) as serial loops. Furthermore, we overlap the communication involved in the action of $\mathbf{P}^{(i_b,t)}\mathbf{B}^{(i_b)}$ and $\mathbf{B}^{(i_b)^T}\mathbf{P}^{(i_b,t)^T}$ with the computation involved in the action of $\mathbf{C}^{(i_b,t)}$, $\mathbf{C}^{(i_b,t)^T}$, $\mathbf{Q}^{(e,i_b,t)}$, $\mathbf{Q}^{(e,i_b,t)^T}$ and $\mathbf{A}^{(e)}$ as illustrated in Algorithm 5. For a given MPI task t, $m_{loc}^{(t)}$ represents the number of locally-owned degrees of freedom (DoFs), while $m_{ghost}^{(t)}$ denotes the number of DoFs on the shared subdomain boundary that are not owned by task t, commonly known as "ghost" DoFs. The storage layout of the multivector in each process consists of a $b \times m_{loc}^{(t)} \times n_b$ sized array for storing the locally-owned data followed by a contiguous storage of $b \times m_{ghost}^{(t)} \times 2$ sized array to hold the data received from tasks that share subdomain boundaries. This allows us to store the subdomain boundary data for two batches so that we can overlap the compute of one batch with the communication involved in another batch. We note that the dimensions of these arrays are in the order of their corresponding fastest index in storage.

GPU Implementation: `MPI` aspects

On GPUs, we do not explicitly construct $\mathbf{U}$. Instead, we evaluate the action of $\mathbf{P}^{(t)}$ on $\mathbf{U}$ through `MPI` communication of boundary data for the multivectors across `MPI` tasks that share subdomain boundaries to extract the subdomain level multivector $\mathbf{U}^{(t)}$. We denote the number of *locally-owned* DoFs on task t as $m_{loc}^{(t)}$ and the number of DoFs on the shared subdomain boundary of task t that are not owned by it but *locally-owned* by task $\hat{t}$ as $m_{ghost}^{(t,\hat{t})}$. Further, let $\hat{n}_t$ denote the number of such tasks $\hat{t}$ for a given task t. The storage layout of the multivector comprises of $b \times m_{loc}^{(t)} \times n_b$ sized array for the locally-owned data followed by $b \times m_{ghost}^{(t,\hat{t})} \times n_b \times \hat{n}_t$ sized array for the data received from the tasks $\hat{t}$ that share subdomain boundaries with task t. Note that the dimensions indicated above are in the order of their corresponding fastest index. This storage layout helps in evaluating $\mathbf{A}^{(e)}\mathbf{U}^{(i_b,e,t)}$ for each i_b, e and t concurrently. We use CUDA-Aware MPI to optimize communications which pipelines message transfers and uses NVIDIA® GPUDirect® for various inter-rank communications like intra-node, inter-node, and RDMA inter-node communication. We further explore a mixed precision strategy to communicate data on the shared subdomain boundary where the boundary data communicated is recast as FP32 floats, which reduces the amount of data that needs to be communicated.

Algorithm 5: Overlap of computation and communication

Input: $\mathbf{U}$
Data: $\mathbf{B}^{(i_b)}, \mathbf{P}^{(i_b,t)}, \mathbf{C}^{(i_b,t)}, \mathbf{Q}^{(i_b,e,t)}$ for $e = 1, \ldots, E_t$ and $i_b = 1, \ldots, n_b$
Result: $\mathbf{V}$
`MPI` **rank:** t

$\mathbf{U}^{t,1} \leftarrow \mathbf{P}^{t,1}\mathbf{B}^{1}\mathbf{U}$;
for $i_b \leftarrow 1$ **to** n_b **do**
  `/* Start communication for batch` $i_b + 1$ `required for evaluating` $\mathbf{P}^{t,i_b+1}\mathbf{B}^{(i_b+1)}\mathbf{U}$ `using MPI_Isend and MPI_Irecv */`
  if $i_b < n_b$ **then**
    Start : $\mathbf{U}^{t,i_b+1} \leftarrow \mathbf{P}^{t,i_b+1}\mathbf{B}^{(i_b+1)}\mathbf{U}$;
  `/* Start communication for batch` $i_b - 1$ `required for evaluating` $\mathbf{P}^{t,i_b-1^T}\mathbf{B}^{(i_b-1)^T}\mathbf{V}^{t,i_b-1}$ `using MPI_Isend and MPI_Irecv. */`
  if $i_b > 1$ **then**
    Start : $\mathbf{V} \leftarrow \mathbf{V} + \mathbf{P}^{t,i_b-1^T}\mathbf{B}^{(i_b-1)^T}\mathbf{V}^{t,i_b-1}$;
  `/* Using Algorithm 3 for the following evaluation. */`
  $\mathbf{V}^{(i_b,t)} \leftarrow \mathbf{C}^{(i_b,t)^T}\left(\sum_e^{E_t} \mathbf{Q}^{e,i_b,t^T}\mathbf{A}^{(e)}\mathbf{Q}^{e,i_b,t}\right)\mathbf{C}^{(i_b,t)}\mathbf{U}^{(i_b,t)}$;
  `/* MPI_Waitall for finishing communication and processing the received data. */`
  if $i_b < n_b$ **then**
    Finish : $\mathbf{U}^{t,i_b+1} \leftarrow \mathbf{P}^{t,i_b+1}\mathbf{B}^{(i_b+1)}\mathbf{U}$;
  if $i_b > 1$ **then**
    Finish : $\mathbf{V} \leftarrow \mathbf{V} + \mathbf{P}^{t,i_b-1^T}\mathbf{B}^{(i_b-1)^T}\mathbf{V}^{t,i_b-1}$;
$\mathbf{V}^{t,n_b} \leftarrow \mathbf{P}^{t,n_b^T}\mathbf{B}^{n_b T}\mathbf{V}^{t,n_b}$;
return $\mathbf{V}$

4. Performance Benchmarks

We now assess the performance of the proposed matrix-free algorithm for multivectors using representative benchmark problems. To this end, we first consider the action of the finite-element (FE) discretized Helmholtz operator on randomly generated multivectors using multi-node CPU and GPU architectures. We begin by examining the sustained performance and strong scaling efficiencies of our implementation for various higher-order FE interpolating polynomial orders. Subsequently, we benchmark our performance against established baselines. The first baseline chosen for benchmarking our performance on both multi-node CPUs and GPUs involves a cell-matrix approach suited naturally for multivectors, as discussed in Section 2.2.1, and has also been employed in previous works [3–5] and particularly in recent works [7, 27] that have been nominated as one of the 2019 ACM Gordon Bell Prize finalists [32]. We also consider a second baseline for benchmarking the performance of matrix multivector products on multi-node CPUs, which involves the single-component matrix-free framework of `deal.II` by looping over the constituent vectors. We consider two benchmark problems to test and evaluate our implementation: (a) The evaluation of the Helmholtz operator action on randomly generated multivectors. In this case, we set $\mu = 1$ and $\kappa(\mathbf{x}) = 2\pi; \forall \mathbf{x} \in \Omega$ in Eq. (1), and (b) the solution of Helmholtz eigenvalue problem using the Chebyshev Filtered Subspace Iteration (ChFSI) method. Here, we set $\mu = 1/2$ and κ as a precomputed potential in Eq. (1).

For our evaluations, we selected the number of nodes for the 1D base mesh to be $n_p = 7, 8, 9$, resulting in Lagrange interpolating polynomial orders `FEOrder` $= 6, 7, 8$. This selection is motivated by the potential of the proposed methods to accelerate the eigensolvers employed to solve FE discretized large-scale eigenvalue problems arising in the domain of quantum modeling of materials. The chosen `FEOrder` offers a balanced approach between the reduction in the number of DoFs required to achieve the desired accuracy and the increased cost per DoF associated with a higher `FEOrder`, as discussed by Motamarri et al. [6], Das et al. [7], Motamarri et al. [33]. It is worth noting that the desired accuracy of evaluating the integrals in Eq. (3) may not always be achieved by using the quadrature rule of order $n_q = n_p$. Therefore, we benchmark the cases involving $n_q = n_p$ and $n_q > n_p$.

To conduct these benchmarks, we employ the computing clusters, Param Pravega (for benchmarking on CPUs) and Summit supercomputer (for benchmarking on GPUs), the configurations of which are described in Table 1. We also report GPU performance benchmarks conducted on the Selene supercomputer in Appendix C.5.

System Config	Summit Supercomputer	Param Pravega (CPU only nodes)
Processor	IBM® POWER9	Intel® Xeon® Platinum 8268
GPU	NVIDIA® Tesla® V100 SXM2 16GB	-
Nodes	4608	428 + 156 (High Memory)
CPU cores/Node	32	48
GPUs/Node	6	-
Node Performance	42 TFLOP/s (V100 FP64)	1.459 TFLOP/s (AVX-512 FP64)
Memory/Node	512 GB DDR4 + 96 GB HBM2	192 GB or 768 GB (High Memory) DDR4
Interconnect	Mellanox® EDR 100G InfiniBand	Mellanox® ConnectX®-6 MT28908
OS	RHEL 8.2	CentOS 7

Table 1: System configurations for the benchmark architectures.

The compilers, MPI and BLAS libraries used are listed in Table 2.

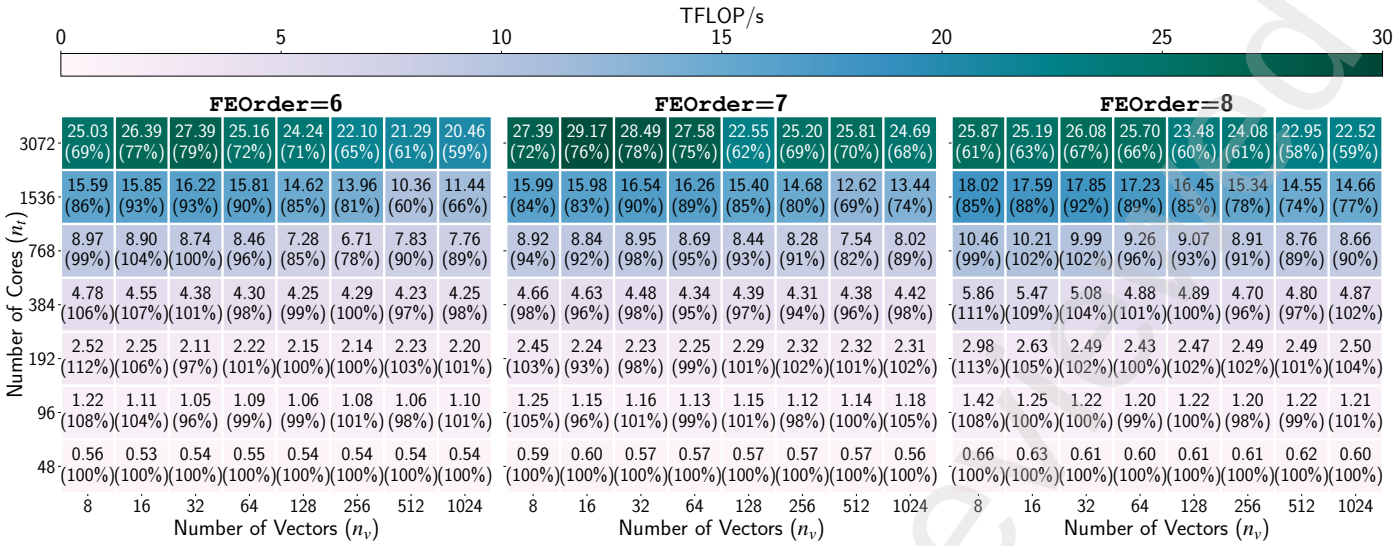


Figure 7: Scaling study of our matrix-free implementation. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8).

Library	GPU Benchmarks	CPU Benchmarks
Compiler	gcc 9.1.0 nvcc 11.0	gcc 12.2.0
Compiler Flags	`-O3 -arch=sm_70 -lcublas`	`-O3 -fopenmp-simd -march=native`
MPI	IBM Spectrum MPI 10.4	Intel® oneAPI MPI 2021.9.0
BLAS	cuBLAS 11.0	Intel® oneAPI MKL 2023.1.0

Table 2: External libraries and compiler flags used for compilation.

4.1. Helmholtz Operator action

We use `deal.II` library version 9.4.2 [34] with the `p4est` [35] backend to perform the MPI-parallel meshing and domain decomposition. We consider a uniform FE mesh with homogeneous Dirichlet boundary conditions.

4.1.1. CPU Benchmarks

We use the marker API of the LIKWID tool [36] with the `perf_event` backend to obtain performance metrics on CPU architectures. To this end, we executed the MPI executable using the command :

```
likwid-mpirun -np $NTASKS -g MEM_DP -m \ $EXECUTABLE
```

Listing 5: `MPI` execution call

In Fig. 7, we show the sustained performance and strong scaling efficiencies of our implementation for `FEOrder` = 6, 7, 8 and the number of vectors $n_v = 8, 16, 32, 64, 128, 256, 512, 1024$ with $n_q = n_p$ until 3072 `MPI` tasks. This scaling study ranges from ∼43,000 DoFs per `MPI` task to ∼670 DoFs per `MPI` task in the case of `FEOrder`= 6, 7 and ∼45,000 DoFs per `MPI` task to ∼700 DoFs per `MPI` task in the case of `FEOrder`= 8. We note that even in the extreme scaling regime of a few hundred DoFs per `MPI` task, our implementation maintains strong scaling efficiencies of about 60% − 70%. The high scaling efficiency observed in our experiments can be attributed to two key factors. First, compute and communication overlap, as described in Section 3.2.5, allows for concurrent execution of computation and communication tasks, thereby minimizing idle time and maximizing resource utilization. Second, using SIMD parallelism and MPI parallelism over different indices, namely multivector batches and subdomains, further enhances the scaling efficiency. We achieve sustained performance of 24.69 TFLOP/s for `FEOrder`=7 and n_v=1024, which is ∼26% of the theoretical peak performance. It is imperative to note that despite achieving a lower percentage of the theoretical peak performance compared to the baseline cell-matrix implementations, the matrix-free approach, as will be demonstrated, achieves a lower time to solution.

We benchmark our implementation against the baselines of the cell-matrix and `deal.II` matrix-free implementation. On CPU architectures, we employ the BCV layout with a batchsize of 128, which gives the best performance for the cell-matrix implementation (see Appendix A.2). We also implement the extraction/assembly operations in the same manner as we do for the matrix-free implementation (discussed in Section 3.2.3). We evaluate the FE-cell level products $\mathbf{V}^{(e,t)} = \mathbf{A}^{(e)}\mathbf{U}^{(e,t)}$ using the `dgemm` module from Intel® oneAPI MKL version 2023.1.0 and compute the matrix multivector products in batches of 128 vectors. Further, we note that many of the optimizations done in the case of the matrix-free implementation are not transferable to the cell-matrix implementation. For instance, the cost of construction of the intermediate data structures required for the application of constraints is prohibitive in case of the cell-matrix implementation due to the larger batchsize employed. Additionally, the overlap of MPI communication with compute across batches leads to cache pollution due to the larger batchsize and causes performance degradation. Thus, in the case of cell-matrix implementation, we utilize level-1 BLAS modules for

the application of constraints and do not overlap compute and communication. Choosing a smaller batchsize to mitigate these issues leads to performance degradation in the computation of FE-cell level matrix-multivector products using `dgemm` modules due to a reduction in arithmetic intensity.

For the second baseline, the `deal.II` matrix-free implementation, we find that the multi-component vector implementation is not very efficient when the number of components is in the order of hundreds. Instead, we compute the FE discretized matrix-multivector product using `deal.II`'s single-component matrix-free implementation by looping over the constituent vectors (see Appendix A.3 for details). Note that `deal.II` also utilizes SIMD vectorization, but unlike our approach, they treat multiple FE-cells concurrently using hardware intrinsics.

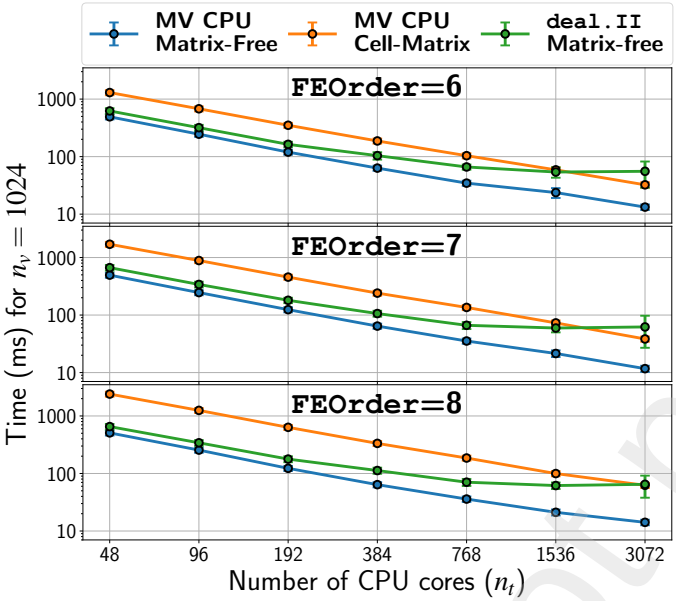


Figure 8: Comparative scaling study of our matrix-free implementation with respect to the cell-matrix method and the `deal.II` matrix-free implementation for $n_v = 1024$ with $n_q = n_p$ and for uniform meshes. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8).

In Fig. 8, we show the scaling data of the proposed implementation compared to the cell-matrix and the `deal.II` matrix-free implementations for $n_q = n_p$. Our implementation has a clear and noticeable performance advantage over the cell-matrix and the `deal.II` matrix-free implementations across various `MPI` tasks. The quantitative performance advantage over both the baseline implementations varies with `MPI` tasks. In particular, we show comparisons in more detail (with varying n_v) for 48 and 3072 `MPI` tasks in Figs. 9 and 10 respectively.

From Fig. 9, we observe that `deal.II` matrix-free implementation is the closest competitor to our proposed approach at all values of n_v in the regime of $\sim 43k - 45k$ DoFs per `MPI` task. Our implementation shows a performance improvement ranging from 5% – 35% over the `deal.II` matrix-free implementation and achieves a speedup of 2.6x – 27.1x over the cell-matrix implementation in this scaling regime.

Fig. 10 shows benchmark comparisons in the extreme scaling regime with $\sim 670 - 700$ DoFs per `MPI` task. To this end,

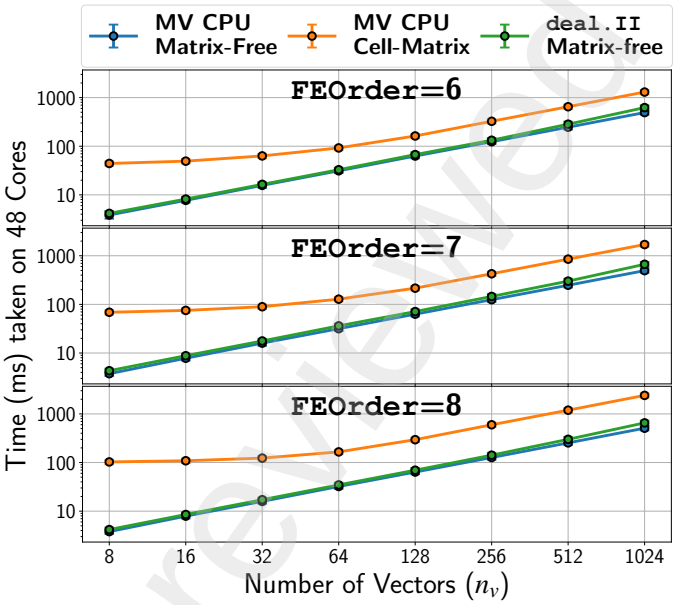


Figure 9: Performance benchmarks of our matrix-free implementation compared to the cell-matrix and `deal.II` matrix-free baseline implementations on 48 `MPI` tasks with $n_q = n_p$ and for uniform meshes. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8).

we observe poor scaling behavior of the `deal.II` matrix-free implementation, attributed to the inefficient utilization of SIMD vectorization for FE cells because there are fewer FE cells per `MPI` task in this regime. However, the proposed matrix-free implementation does not suffer from this drawback. Furthermore, our implementation shows a performance improvement ranging from 2.9x – 5.9x over the `deal.II` matrix-free implementation and 2.4x – 4.4x over the cell-matrix implementation in this

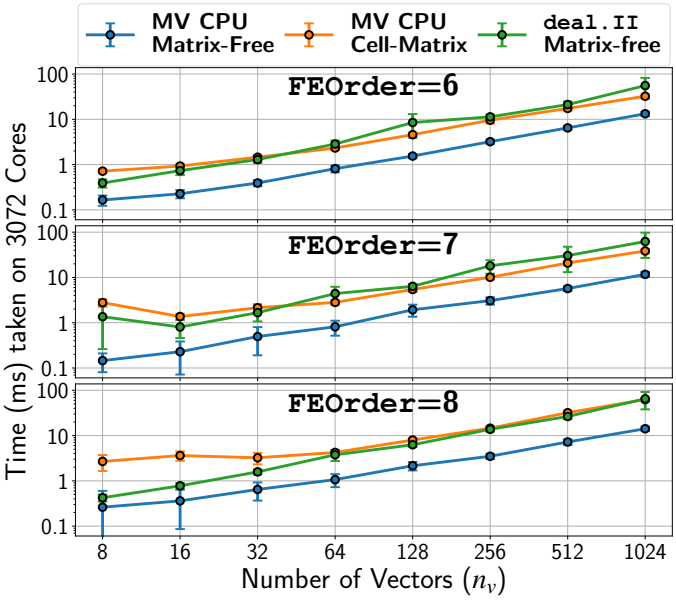


Figure 10: Performance benchmarks of our matrix-free implementation compared to the cell-matrix and `deal.II` matrix-free baseline implementations on 3072 `MPI` tasks with $n_q = n_p$ and for uniform meshes. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8).

scaling regime (for $n_v \geq 64$).

We further benchmark our implementation for the case $n_q > n_p$ and, to that end, choose $n_q = n_p + 2$ for our investigations. We achieve a performance improvement ranging from 3% – 29% over the `deal.II` matrix-free implementation, and speedups ranging from 1.8x – 19.9x over the cell-matrix implementation in the regime of $\sim 43k - 45k$ DoFs per `MPI` task. On the other extreme, in the regime of $\sim 670 - 700$ DoFs per `MPI` task, we achieve speedups ranging from 2.8x – 7.3x over the `deal.II` matrix-free implementation and 1.2x – 5.4x over the cell-matrix implementation (for $n_v \geq 64$). A discussion of these results is provided in the Appendix (see Fig. A.22)

4.1.2. GPU Benchmarks

We use NVIDIA® Tesla® V100 SXM2 16GB GPUs, available on the Summit supercomputer, to analyze the performance of our proposed approach on multi-node GPUs. The computational times are measured using the `clock_gettime` function with the `CLOCK_MONOTONIC` argument as it has a nanosecond resolution. Appropriate barriers such as `MPI_Barrier` and `cudaDeviceSynchronize` are used around the code of interest. To reduce the noise in our reported timings, the collected data is averaged over 100 repetitions. NVIDIA® Nsight™ Compute 2021.2 profiler is used to obtain the total floating point operations. `cudaProfilerStart` and `cudaProfilerStop` are used to mark the code of interest, and the following wrapper script is used in conjunction with `mpirun` to profile:

```
metrics+="
sm__sass_thread_inst_executed_op_dadd_pred_on.
    sum,\
sm__sass_thread_inst_executed_op_dfma_pred_on.
    sum,\
sm__sass_thread_inst_executed_op_dmul_pred_on.
    sum"

ncu --metrics $metrics --profile-from-start
    off --target-processes all $EXECUTABLE
```

Listing 6: Wrapper script for profiling with Nsight Compute for multi-node GPUs

The `compute` kernel (Listing 3) is launched with a 2-D grid of $E_t \times n_b$ thread blocks, each with a 2-D block of $n_{t_x} = b$ threads in the x-direction and $n_{t_y} = \texttt{warpSize} \times \alpha$ threads in the y-direction where `warpSize` = 32 for NVIDIA GPUs and α is a tunable parameter. The optimal batchsize b and parameter α are determined for each `FEOrder` by benchmarking for various values within the limits allowed by the GPU hardware. For instance, V100 GPUs have a default shared memory limit of 48 kB, which can be increased to a maximum of 96 kB by the user. Hence, b is limited by available shared memory. The value of α is limited by the maximum number of threads per thread block and the maximum number of registers per thread block. Furthermore, n_{t_y} threads in each thread block are used to loop over an index of size n_q^2 (or n_p^2 or $n_p n_q$) which in turn affects the optimal values of n_{t_y} to be used for each `FEOrder` (see Listing 4). A sustained performance analysis is performed to obtain the optimal values of n_{t_x} and n_{t_y} (see Fig. C.23) and are tabulated in Table 3. We note that the subsequent GPU benchmarking studies in this section employ these tabulated optimal values. Furthermore, the above analysis (see Fig. C.23) indicates that our Multivector GPU Matrix-Free implementation achieves a sustained performance of ~2.99 TFLOP/s on a single GPU involving 1024 vectors and ~120k DoFs which is about 38% of the peak performance of a NVIDIA® V100 GPU.

FEOrder	b	α
6	8	2
7	4	2
8	2	4

Table 3: Optimal values of b and α for various `FEOrder` to decide the values of n_{t_x} and n_{t_y}

We subsequently evaluate the performance of our matrix-free implementation by conducting a strong scaling study on number of GPUs ranging from 6 to 96 employing the number of vectors n_v = 8, 16, 32, 64, 128, 256, 512, 1024. Figure 11 shows the heatmap corresponding to this study. We observe that for a large number of vectors (512-1024), our implementation results in parallel scaling efficiencies of 30% – 50% for 12 GPUs (~90k DoFs/GPU) and 10% – 15% for 96 GPUs (~12k DoFs/GPU). The matrix-free method has a reduced arithmetic complexity compared to the cell-matrix approach; hence, inter-node communication quickly becomes the dominant cost on GPUs as the number of nodes increases, resulting in a lower scaling efficiency. It is important to note that the matrix-free multivector approach proposed here achieves lower solution times than the cell-matrix approach, as will be demonstrated, despite the lower percentage of theoretical peak performance. This can be attributed to the reduced arithmetic complexity and the proposed hardware-aware implementation strategies for the matrix-free approach minimizing the data movement costs during matrix-multivector multiplication.

We now compare our matrix-free implementation with the cell-matrix approach as a baseline (Section 2.2.1) for matrix-multivector products. For the cell-matrix approach, we follow the method described in [7, 27], i.e., after extraction of the global nodal vector $\mathbf{U}$ to a cell-level vector $\mathbf{U}^{(e,t)}$ in device memory, we evaluate the FE-cell level products $\mathbf{V}^{(e,t)} = \mathbf{A}^{(e)}\mathbf{U}^{(e,t)}$ using the `cublasDgemmStridedBatched` module from NVIDIA® CUDA 11.0 and compute these dense matrix multivector products sequentially over batches with a batchsize $b = 256$ vectors in BCV layout (Section 3.2.1) when $n_v > 256$. This batchsize is chosen after conducting a performance study with varying batchsizes in the case of the cell-matrix approach (see Fig. C.25). Finally, an assembly operation is performed to build the global product vector $\mathbf{V}$ as discussed in Section 2.2.1. Currently, state-of-the-art FE libraries such as `deal.II` do not have a multivector matrix-free implementation on GPUs. Hence, we compare our proposed matrix-free approach in the case of FE discretized matrix-single vector multiplication against `deal.II`'s single vector matrix-free implementation and observe speedups up to ~17x for `FEOrder` = 6, 7 and 8 for the Helmholtz operator (see Fig. C.24). Thus, we compare our matrix-free implementation with only the cell-matrix approach for multivectors.

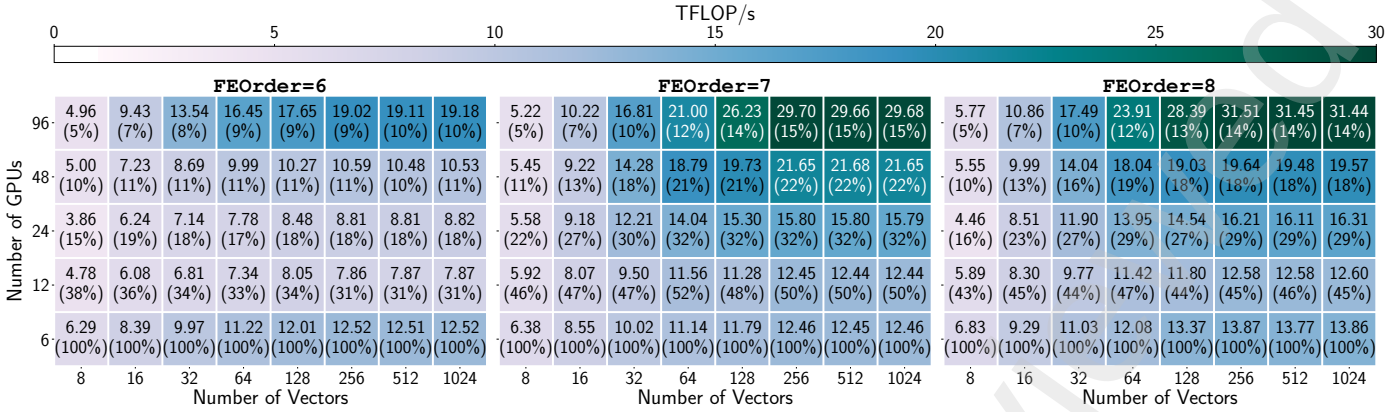


Figure 11: Scaling study of our matrix-free implementation on 6 to 96 V100 GPUs employing the number of vectors n_v = 8, 16, 32, 64, 128, 256, 512, 1024. For a large number of vectors (512-1024), our implementation results in parallel scaling efficiencies of ~30-50% for 12 GPUs (~90k DoFs/GPU) and ~10-15% for 96 GPUs (~12k DoFs/GPU). Case studies: 1092727 DoFs (`FEOrder`=6); 1191016 DoFs (`FEOrder`=7); 1157625 DoFs (`FEOrder`=8).

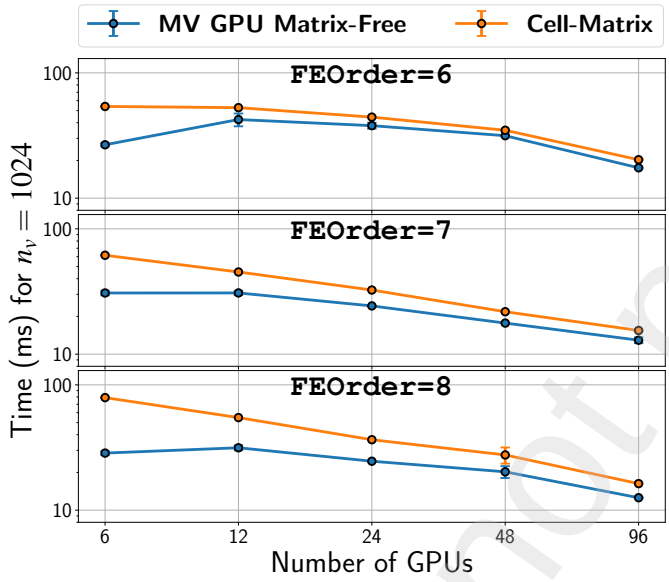


Figure 12: Scaling study comparisons of the proposed matrix-free multivector implementation with cell-matrix baseline for 1024 vectors. Case studies: 1092727 DoFs (`FEOrder`=6); 1191016 DoFs (`FEOrder`=7); 1157625 DoFs (`FEOrder`=8) for the Helmholtz problem on V100 GPUs.

To this end, Fig. 12 shows a comparative strong scaling study with the two approaches for 1024 vectors on a problem involving ~1.2m DoFs, and we note that GPU matrix-free implementation has a noticeable performance advantage over the cell-matrix method across all MPI tasks for `FEOrder` = 6, 7 and 8. In Fig. 12, we observe a slight increase in computational wall time in the case of matrix-free approach from one Summit node to two Summit nodes in contrast to the cell-matrix approach. We attribute this increase in time to the increased cost of inter-node communication, which is more apparent in the case of matrix-free approach because of its reduced arithmetic complexity compared with the cell-matrix approach. We also note that the timings begin to decrease with an increase in the number of nodes beyond two, since the degrees of freedom per GPU involved in communication reduce. Figs. 13 and 14 illustrate the timing comparisons for 1 Summit node (6 GPUs) and 16 Summit nodes (96 GPUs), respectively for varying number of vectors (n_v).

On 1 Summit node (6 GPUs, ~200k DoFs/GPU), we observe speedups of close to 2.0x for `FEOrder` = 6 and 7 and a 2.8x speedup for `FEOrder` = 8 in comparison to the cell-matrix approach for the case of 1024 vectors. In the case of 8 vectors, we observe speedups close to 6x for `FEOrder`=6 and 7 and close to 11x for `FEOrder` = 8 for 1024 vectors on 1 Summit node.

Fig. 14 shows the performance comparisons in the case of 16 Summit nodes (96 GPUs, ~12k DoFs/GPU), and we observe

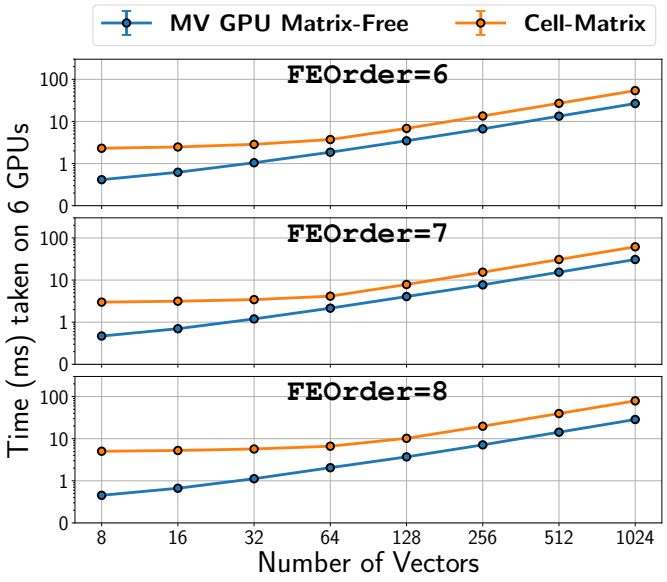


Figure 13: Performance benchmarks of our matrix-free implementation compared to the cell-matrix method on 1 node. Case studies: 1092727 DoFs (`FEOrder`=6); 1191016 DoFs (`FEOrder`=7); 1157625 DoFs (`FEOrder`=8) for the Helmholtz problem on V100 GPUs.

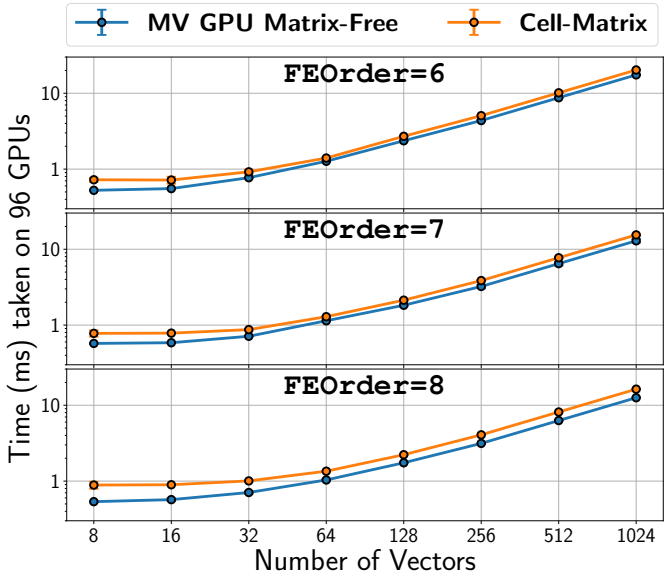


Figure 14: Performance benchmarks of our matrix-free implementation compared to the cell-matrix method on 16 nodes. Case studies: 1092727 DoFs (`FEOrder`=6); 1191016 DoFs (`FEOrder`=7); 1157625 DoFs (`FEOrder`=8) for the Helmholtz problem on V100 GPUs.

computational gains of 16%, 20% and 30% for `FEOrder` = 6, 7 and 8 respectively against the cell-matrix method for 1024 vectors. On 4 Summit nodes (24 GPUs, ~45k DoFs/GPU), we observe better performance improvements of up to 50% for `FEOrder` = 8 in this case of 1024 vectors.

4.2. Helmholtz Eigenvalue Problem

We now present an important benchmark involving the solution of the FE discretized eigenvalue problem (EVP), leveraging the proposed matrix-free implementation to evaluate matrix multivector products arising during the course of an iterative procedure adopted to solve the EVP. Consequently, we consider the FE discretized EVP corresponding to the Helmholtz operator, which can be expressed as follows:

$$\mathbf{HU} = \mathbf{MU\Lambda} \tag{25}$$

where $\mathbf{H} = \mathbf{K} + \mathbf{M}^{\kappa}$ and $\mathbf{M}$ is the FE basis overlap matrix (mass matrix), as discussed in Eq. (2). We employed the Chebyshev Filtered Subspace Iteration (ChFSI) algorithm [37] to solve for n_{ev} smallest eigenvalue/eigenvector pairs. To this end, the generalized eigenvalue problem is converted into a standard eigenvalue problem by defining $\widetilde{\mathbf{H}} = \mathbf{M}^{-1/2}\mathbf{H}\mathbf{M}^{-1/2}$ and $\widetilde{\mathbf{U}} = \mathbf{M}^{1/2}\mathbf{U}$, resulting in

$$\widetilde{\mathbf{H}}\widetilde{\mathbf{U}} = \widetilde{\mathbf{U}}\mathbf{\Lambda} \tag{26}$$

To efficiently compute $\mathbf{M}^{-1/2}$ in Eq. (26), the overlap integral involved in $\mathbf{M}$ is evaluated using a Gauss-Lobatto-Legendre quadrature rule of order `FEOrder` resulting in a diagonal matrix $\mathbf{M}$ [6]. Algorithm 6 below describes the ChFSI procedure that is employed to solve the standard eigenvalue problem in Eq. (26).

Algorithm 6: Chebyshev Filtered Subspace Iteration

Input: Initial Guess of $\mathbf{U}$
Data: Chebyshev polynomial order m, estimates of the bounds of the eigenspectrum $\lambda_{max}, \lambda_{min}$, estimate of the upper bound of the wanted spectrum λ_u and the tolerance for the residual τ
Temporary Variables: $e, c, \sigma, \sigma_1, \gamma, \alpha_1, \alpha_2, \mathbf{X}$ and $\mathbf{Y}$
Result: $\mathbf{U}$ and $\mathbf{\Lambda}$

while $\|\widetilde{\boldsymbol{H}}\mathbf{U} - \mathbf{U\Lambda}\| > \tau$ **do**
  $e \leftarrow \frac{\lambda_{max}-\lambda_u}{2}$;
  $c \leftarrow \frac{\lambda_{max}+\lambda_u}{2}$;
  $\sigma \leftarrow \frac{e}{\lambda_{min}-c}$;
  $\sigma_1 \leftarrow \sigma$;
  $\gamma \leftarrow \frac{2}{\sigma_1}$;
  $\alpha_1 \leftarrow \frac{\sigma_1}{e}$;
  $\alpha_2 \leftarrow -c$;
  $\mathbf{Y} \leftarrow 0$;
  $\mathbf{X} \leftarrow \mathbf{U}$;
  $\mathbf{Y} \leftarrow \alpha_1\widetilde{\boldsymbol{H}}\mathbf{X} + \alpha_1\alpha_2\mathbf{X}$;
  for $d \leftarrow 2$ **to** m **do**
    $\sigma_2 \leftarrow \frac{1}{\gamma-\sigma}$;
    $\alpha_1 \leftarrow \frac{2\sigma_2}{e}$;
    $\alpha_2 \leftarrow -\sigma\sigma_2$;
    $\mathbf{X} \leftarrow \alpha_1\widetilde{\mathbf{H}}\mathbf{Y} + \alpha_2\mathbf{X} - c\alpha_1\mathbf{Y}$;
    swap($\mathbf{X}, \mathbf{Y}$);
  $\mathbf{X} \leftarrow \mathbf{Y}$;
  solve $\mathbf{X}^T\widetilde{\mathbf{H}}\mathbf{X}\mathbf{Q} = \mathbf{X}^T\mathbf{X}\mathbf{Q}\mathbf{\Lambda}$;
  $\mathbf{U} \leftarrow \mathbf{X}\mathbf{Q}$;
return $\mathbf{U}$ *and* $\mathbf{\Lambda}$

We use the `deal.II` library version 9.4.2 [34] with the `p4est` [35] backend to perform MPI-parallel meshing and domain decomposition. We compute $n_{ev} = 1024$ smallest eigenvalue/eigenvector pairs driving the eigenvalue problem residual to a value $\tau = 5 \times 10^{-5}$, and we consider a buffer of 25%, resulting in the trial subspace $\mathbf{U}$ comprising of 1280 vectors to be employed in the ChFSI Algorithm 6. We employ our baselines described earlier to compute matrix-multivector products during the ChFSI procedure and conduct comparative studies with our proposed matrix-free implementation on both multi-node CPUs and GPUs as described subsequently.

4.2.1. CPU Benchmarks

Fig. 15 shows the strong scaling data of our implementation compared to the cell-matrix and the `deal.II` matrix-free implementations for uniform meshes. Our implementation has a clear and noticeable performance advantage over the cell-matrix and the `deal.II` matrix-free implementations across various `MPI` tasks. On 48 `MPI` tasks, we achieve speedups of about 2.2x, 3.0x, and 4.0x compared to the cell-matrix implementation and around 1.5x compared to the `deal.II` implementation for `FEOrder` values of 6, 7, and 8. Similarly, on 3072 `MPI` tasks, our implementation yields speedups of about 2.1x, 3.0x, and 3.7x

over the cell-matrix implementation, and about 2.7x, 3.4x, and 3.0x over the `deal.II` implementation for `FEOrder` values of 6, 7, and 8, respectively.

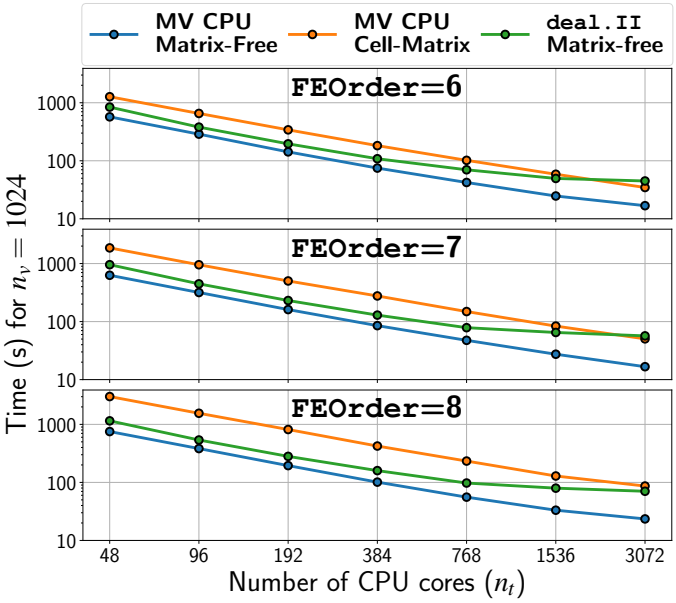


Figure 15: Performance benchmarks of our matrix-free implementation compared to the cell-matrix and `deal.II` matrix-free baseline implementations for the eigenvalue problem on uniform meshes. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8). Chebyshev polynomial orders 67, 76 and 83 were chosen for `FEOrder`=6, 7 and 8 respectively.

We also consider the case of an adaptively refined FE mesh with hanging node constraints (see Fig. 16) to benchmark the performance of our matrix-free implementation within the eigenvalue solver framework using ChFSI.

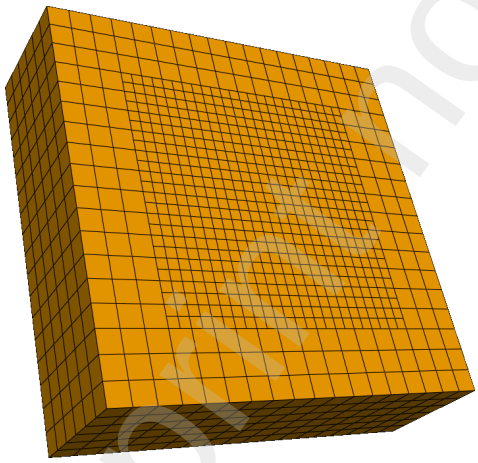


Figure 16: Example of an adaptively refined mesh with a single level of refinement

The results plotted in Fig. 17 indicate speedups of 1.9x, 2.3x, and 2.8x, for `FEOrder`=6, 7 and 8 respectively, over the cell-matrix implementation, and 1.5x for `FEOrder`=6, 7 and 1.3x for `FEOrder`=8, over the `deal.II` implementation in the extreme-scaling regime (3072 `MPI` tasks with ~700 DoFs per `MPI` task). We attribute the drop in speedups in comparison

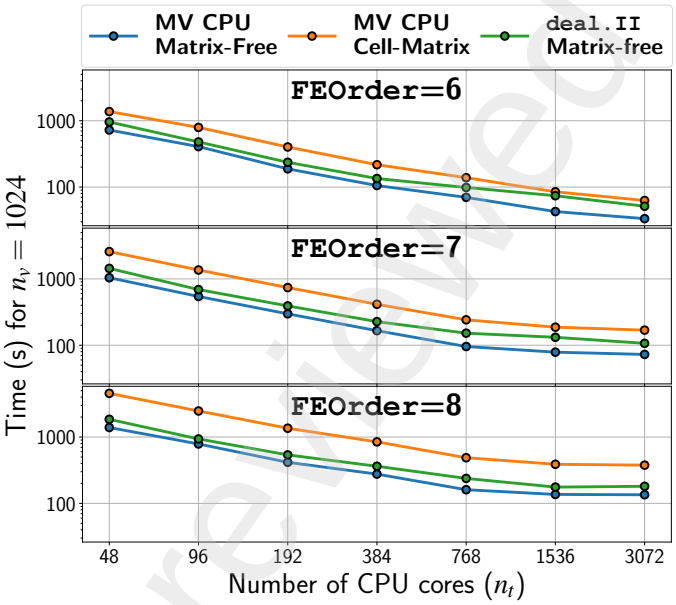


Figure 17: Performance benchmarks of our matrix-free implementation compared to the cell-matrix and `deal.II` matrix-free baseline implementations for the eigenvalue problem on adaptively refined meshes (1 level of refinement). Case studies: 2032609 DoFs (`FEOrder`=6); 2018187 DoFs (`FEOrder`=7); 2054553 DoFs (`FEOrder`=8). Chebyshev polynomial orders 67, 71 and 83 were chosen for `FEOrder`=6, 7 and 8 respectively.

with the uniform mesh to the increase in the time taken for the application of hanging-node constraints as described in Fig. 2.

4.2.2. GPU Benchmarks

Figure 18 shows the performance benchmark of our Multivector GPU Matrix-Free implementation compared with the cell-matrix implementation in the case of uniform meshes for the solution of the Helmholtz eigenvalue problem. We also explore a mixed precision strategy to communicate data on the shared subdomain boundary of MPI task 't'. To this end, the boundary data communicated is recast as FP32 floats, which reduces the amount of data that needs to be communicated. The results indicate that our implementation has a clear and noticeable performance advantage over the cell-matrix implementation across varying `MPI` tasks, and this advantage improves with increase in `FEOrder`. For instance, on one node (~200k DoFs/GPU), we obtain performance improvements of up to 60% for `FEOrder`=6, 64% for `FEOrder`=7 over the cell-matrix approach. Furthermore, a speedup of 2.2x is obtained in the case of `FEOrder`=8. On four nodes (~50k DoFs/GPU), we obtain performance improvements of around 14% for `FEOrder`=6, 13% for `FEOrder`=7, and 41% for `FEOrder`=8 over the cell-matrix implementation. For 16 nodes (~12k DoFs/GPU), speedups of ~10% are observed for all `FEOrders`. This drop in speedup with an increase in the number of nodes can be attributed to communication costs becoming dominant compared to the cost of floating point operations due to the reduced arithmetic complexity of matrix-free multivector products. Further performance on multi-node GPUs can be obtained by overlapping compute on GPUs, MPI communication between GPUs, and data movement from device memory, which will be a part of future investigations.

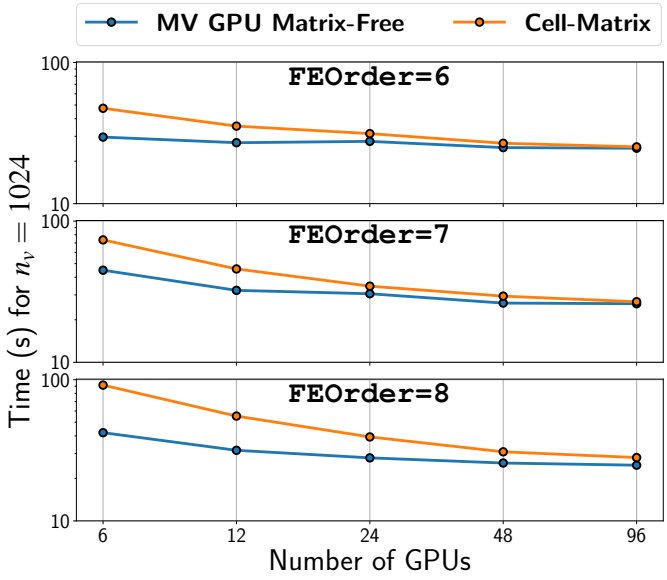


Figure 18: Performance benchmarks of our matrix-free implementation compared to the cell-matrix implementation for the eigenvalue problem on uniform meshes. Case studies: 1092727 DoFs (`FEOrder`=6); 1191016 DoFs (`FEOrder`=7); 1157625 DoFs (`FEOrder`=8). Chebyshev polynomial orders 67, 76 and 83 were chosen for `FEOrder`=6, 7 and 8 respectively.

Similar to CPU benchmarks, we also consider the case of an adaptively refined (FE) mesh with hanging node constraints to report our performance benchmarks on GPUs. The results plotted in Fig. 19 demonstrate a performance advantage over the cell-matrix implementation across various `MPI` tasks. On one node, we obtain a performance advantage of 62% for `FEOrder`=6 and 7 and that of 65% for `FEOrder`=8 over the cell-matrix implementation. For 16 nodes, a performance advantage of ~10% is observed for all `FEOrders`. These results in the case of adaptively refined mesh indicate a similar performance advantage over the cell-matrix approach as in the case of a uniform FE mesh.

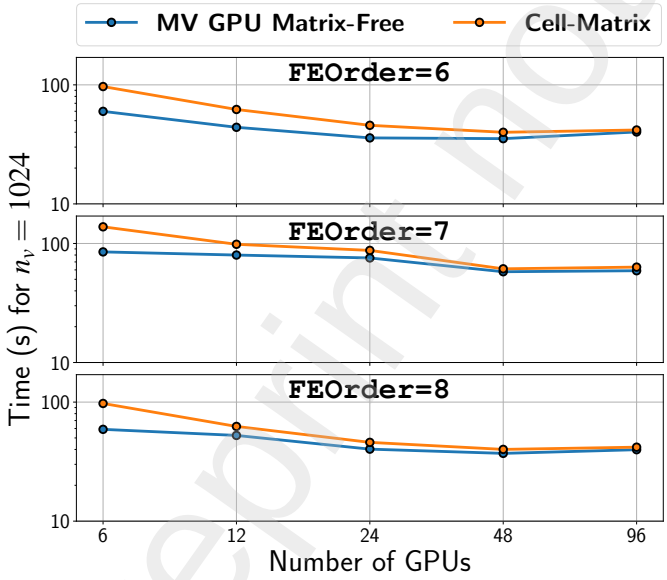


Figure 19: Performance benchmarks of our matrix-free implementation compared to the cell-matrix implementation for the eigenvalue problem on adaptively refined meshes (1 level of refinement). Case studies: 1185321 DoFs (`FEOrder`=6); 1177963 DoFs (`FEOrder`=7); 1226673 DoFs (`FEOrder`=8). Chebyshev polynomial orders 67, 76, and 83 were chosen for `FEOrder`=6, 7 and 8 respectively.

5. Conclusion and future work

In conclusion, this work presents an efficient hardware-aware algorithm and implementation strategies for computing FE discretized matrix-multivector products in the matrix-free paradigm on multi-node CPU and multi-node GPU architectures. The proposed method addresses a significant gap in the currently available implementations of matrix-free methods, which are neither optimal nor directly applicable for the action of an FE discretized operator on a large number of FE discretized fields. We propose a batched layout for storing the multivector whose batchsize can be tuned to the underlying hardware architectures. Our implementation employs different batched evaluation strategies to compute the matrix-multivector products depending on the architecture to achieve the best possible performance. We also employ architecture-specific implementation strategies to evaluate the tensor contractions encountered in the matrix-free approach. For CPU architectures, we use even-odd decomposition to reduce computation, SIMD vectorization to exploit thread-level parallelism, and overlapping computation and communication to increase scaling efficiency. On GPU architectures, we employ GPU shared memory and kernel fusion for GPU architectures to reduce accesses to and from device memory and registers to reduce bank conflicts. Furthermore, we utilize constant memory on GPUs to broadcast accesses and reduce shared memory usage and bank conflicts. We also design an algorithm to overlap computation and data movement in conjunction with the proposed batched layout on GPUs. These techniques have allowed us to achieve significant performance gains. Our results indicate that this implementation outperforms the closest benchmark, achieving computational gains of 2.77x on 1 Summit node (6 GPUs, ~200k DoFs/GPU), 30% on 16 Summit nodes (96 GPUs, 12k DoFs/GPU), and 4.43x on 64 nodes of Param Pravega (3072 CPU cores, ~700 DoFs/core) for matrix-multivector products (1024 vectors) for polynomial order 8. Additionally, the strong scaling studies and performance benchmarks we showed on both multi-node CPU and GPU architectures demonstrate the effectiveness of this implementation in solving large-scale problems over existing matrix-free implementations. We also demonstrated that the proposed method is particularly suitable for solving large-scale nonlinear eigenvalue problems. To this end, we have performed benchmark studies for the solution of an eigenvalue problem using the Chebyshev Filtered Subspace Iteration (ChFSI) [37] approach and achieved speedups of 1.6x – 2.17x on a uniform mesh for 1 Summit node (6 GPUs, ~200k DoFs/GPU), 14% – 41% for 4 Summit nodes (24 GPUs, ~50k DoFs/GPU), ~10% on 16 Summit nodes (96 GPUs, ~12k DoFs/GPU), and 2.0x – 3.0x on 64 nodes of Param Pravega (3072 CPU cores, ~700 DoFs/core)

for matrix-multivector products (1024 vectors) compared to the best baseline implementation for polynomial order 6, 7, and 8.

The methodologies discussed in this work can be straightforwardly extended to other blocked iterative eigensolvers. Furthermore, these methodologies can also be utilized for solving linear systems of equations arising from FE discretizations with multiple RHS vectors and can accelerate algorithms such as Block Krylov subspace methods [38] employed to solve these problems.

Although we observe significant performance improvements of our matrix-free implementation compared to the baselines, we note that the performance advantage of our implementation decreases with an increase in the number of nodes on multi-node GPU architectures. We attribute this drop in performance advantage to inter-node communication becoming the dominant cost due to the reduction in floating point operations in the matrix-free approach. This necessitates the optimization of communication involved in the action of the Helmholtz operator on the multivector to achieve further performance. Hence, in this regard, as part of future work, strategies like CUDA streams will be employed to overlap computations with communication. Extensions of the proposed algorithm to more complicated FE discretized operators such as Kohn-Sham DFT [39, 40] Hamiltonian will be part of future investigations.

6. Acknowledgments

The authors gratefully acknowledge the seed grant from Indian Institute of Science (IISc) and the SERB Startup Research Grant from the Department of Science and Technology (DST), India (Grant Number:SRG/2020/002194) for the purchase of a GPU cluster, which provided computational resources for this work. The research used the resources of PARAM Pravega at the Indian Institute of Science, supported by National Supercomputing Mission (NSM) R&D for exa-scale grant (DST/NSM/R&D_Exascale/2021/14.02). This research also used resources of the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory (ORNL), supported by the Office of Science of the U.S. Department of Energy (DoE) under Contract No. DE-AC05-00OR22725. We also acknowledge financial support in the form of the Prime Minister's Research Fellowship (PMRF) from the Ministry of Education (MoE), India, and the Junior Research Fellowship (JRF) from the Council of Scientific and Industrial Research (CSIR), Ministry of Science and Technology, India. We also thank Prathu Tiwari, Vinay Deshpande and Bharatkumar Sharma, all from NVIDIA, India, for fruitful discussions and for helping us run a few of our benchmarks on NVIDIA Selene.

7. Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used Paperpal in order to proofread. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Appendix

A. Matrix multivector products – CPU implementations

A.1. Multivector CPU matrix-free implementation

We discuss the effect of varying the batchsize in our CPU multivector matrix-free implementation and its sustained performance as shown in Fig. A.20. We observe no appreciable gain upon increasing the batchsize to be greater than the SIMD width, and hence, the SIMD width has been chosen for all our CPU studies reported in this work.

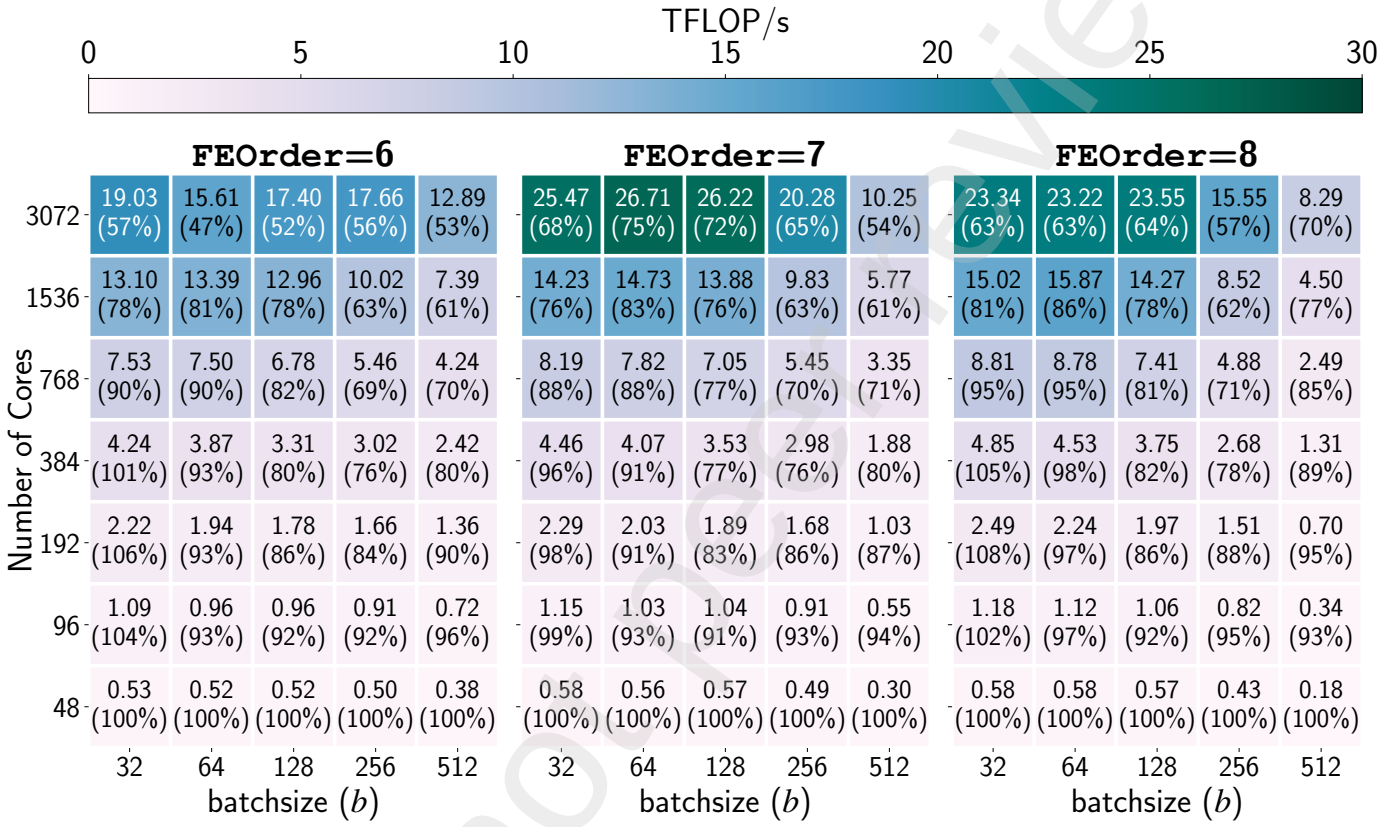


Figure A.20: Performance study of our multivector CPU implementation for varying batchsizes. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8).

A.2. Cell-matrix CPU implementation

We also vary the batchsize in our CPU multivector cell-matrix implementation, and the resulting sustained performance is shown in Fig. A.21. We note that the best performance is obtained using a batchsize of 128 and is used as a baseline to compare our matrix-free implementations with all the results reported in this work.

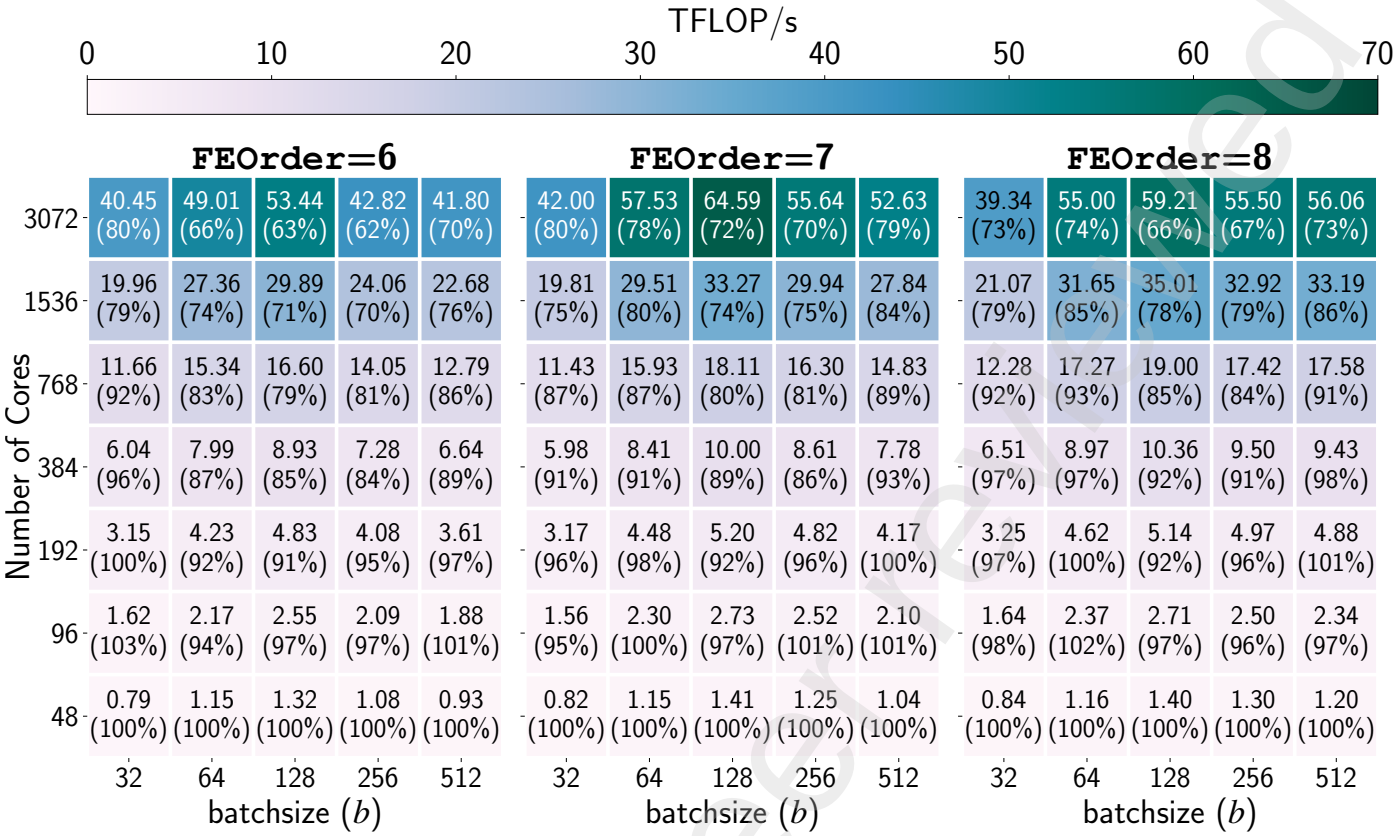


Figure A.21: Performance study of our cell-matrix CPU implementation for varying batchsizes. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8).

A.3. `deal.II` matrix-free CPU implementation

In the case of `deal.II` matrix-free implementation, we implement the FE discretized matrix multivector product using the `deal.II`'s single-component matrix-free implementation by looping over the constituent vectors, which is equivalent to setting $b = 1$ in our framework.

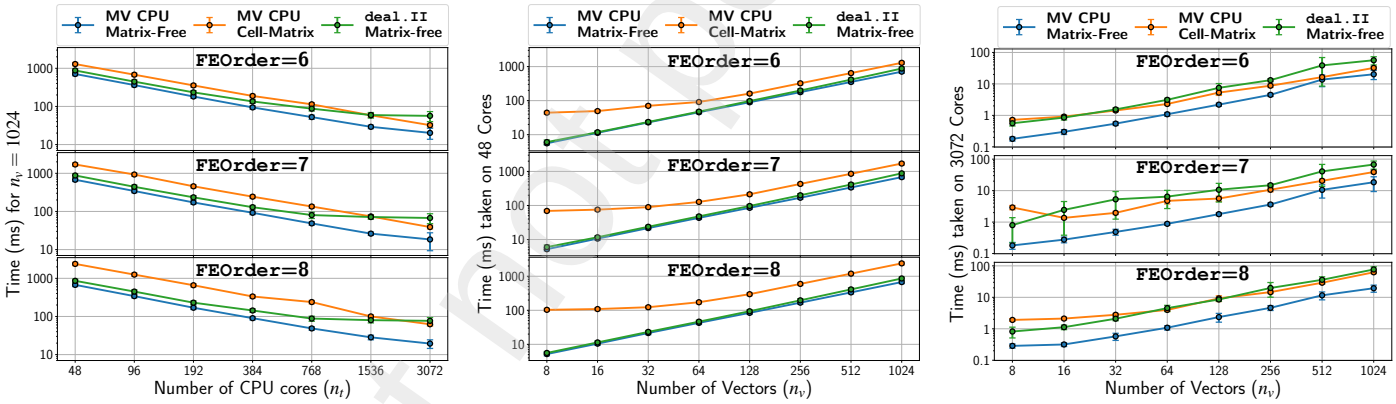
```
template <unsigned int FEOrder>
void
SolverProblem<FEOrder>::vmult(
    dealii::LinearAlgebra::distributed::BlockVector<double> &Ax,
    dealii::LinearAlgebra::distributed::BlockVector<double> &x)
{
    for (auto i = 0; i < d_blocksize; ++i)
        d_matrixFreeDataPtr->cell_loop(
            &SolverProblem<FEOrder>::AX,
            this,
            Ax.block(i),
            x.block(i),
            true);
}
template <unsigned int FEOrder>
void
SolverProblem<FEOrder>::AX(
  const dealii::MatrixFree<3, double>                  &matrixFreeData,
  dealii::LinearAlgebra::distributed::Vector<double>   &y,
  const dealii::LinearAlgebra::distributed::Vector<double> &x,
  const std::pair<unsigned int, unsigned int>          &cell_range) const
{
const dealii::VectorizedArray<double> tpi =
```

```
dealii::make_vectorized_array((2.0 * M_PI));
dealii::FEEvaluation<3, FEOrder, FEOrder + 3> fe_eval(
matrixFreeData,
d_matrixFreeVectorComponent,
d_matrixFreeQuadratureComponentAX);

for (unsigned int cell = cell_range.first; cell < cell_range.second; ++cell)
{
    fe_eval.reinit(cell);
    fe_eval.gather_evaluate(x,
                            dealii::EvaluationFlags::gradients |
                                dealii::EvaluationFlags::values);
    for (unsigned int q = 0; q < fe_eval.n_q_points; ++q)
    {
        fe_eval.submit_gradient(fe_eval.get_gradient(q), q);
        fe_eval.submit_value(fe_eval.get_value(q) * tpi, q);
    }
    fe_eval.integrate_scatter(dealii::EvaluationFlags::gradients |
                            dealii::EvaluationFlags::values,
                            y);
}
}
```

Listing 7: Implementation of multivector using the `deal.II` matrix-free framework

A.4. Performance comparisons for $n_q = n_p + 2$



(a) Comparative scaling study of our implementation with respect to the cell-matrix method and `deal.II` matrix-free implementation for $n_v = 1024$.

(b) Performance benchmark of our implementation against the cell-matrix and `deal.II` matrix-free baseline implementations on 48 `MPI` tasks.

(c) Performance benchmark of our implementation against the cell-matrix and `deal.II` matrix-free baseline implementations on 3072 `MPI` tasks.

Figure A.22: Benchmarks of our implementation with cell-matrix and `deal.II` matrix-free implementations for the case of $n_q = n_p + 2$ and uniform mesh. Case studies: 2048383 DoFs (`FEOrder`=6, 7); 2146689 DoFs (`FEOrder`=8).

In this section, we discuss the comparative studies conducted in the case of $n_q = n_p + 2$. To this end, Fig. A.22a shows the scaling data of our implementation compared with that of the cell-matrix and the `deal.II` matrix-free implementations. Our implementation has a clear and noticeable performance advantage over the cell-matrix and the `deal.II` matrix-free implementations across various `MPI` tasks. We show the comparisons in more detail (with varying n_v) for 48 and 3072 `MPI` tasks in Figs. A.22b and A.22c respectively. From Fig. A.22b, we see that the closest competitor to our implementation at every value of n_v in the regime of $\sim 43k - 45k$ DoFs per core is the `deal.II` matrix-free implementation. We note that our implementation shows a performance improvement ranging from 1.03x to 1.29x over the `deal.II` matrix-free implementation and 1.80x to 19.94x over the cell-matrix implementation in this scaling regime. However, as shown in Fig. A.22c, we see that the closest competitor to our implementation in the regime of $\sim 670 - 700$ DoFs per MPI task is not well-defined in the case of $n_q = n_p + 2$ just as in the case of $n_q = n_p$ reported in Section 4.1.1. Our implementation shows a performance improvement ranging from 2.79x to 7.32x over the `deal.II` matrix-free implementation and 1.20x to 5.37x over the cell-matrix implementation in this scaling regime (for $n_v \geq 64$).

B. Eigensolver implementations using ChFSI on CPUs

B.1. Multivector matrix-free CPU implementation

To implement matrix-free multivector products in CPUs, the core compute kernel was modified to compute

$$\mathbf{Y} = a\mathbf{M}^{-1/2}\mathbf{H}\mathbf{M}^{-1/2}\mathbf{X} + b\mathbf{X} + c\mathbf{Y} \tag{B.1}$$

Scaling of the data structures $\mathbf{X}$ and $\mathbf{Y}$ with the diagonal matrix $\mathbf{M}^{-1/2}$ and scalar constants a, b and c is performed during extraction and assembly, as this allows us to reuse cached data more often as opposed to scaling $\mathbf{X}$ and $\mathbf{Y}$ in entirety before/after the extraction/assembly. The constraint matrices are modified appropriately to allow for the computation of $\mathbf{M}^{-1/2}\mathbf{X}$ at the cell-level.

```
for (unsigned int iDoF = 0; iDoF < d_ndofsPerCell; ++iDoF)
{
    unsigned int l2g =
    singleVectorGlobalToLocalMap[iDoF + d_ndofsPerCell * iCell];
    temp1ov[iDoF] =
    x[getMultivectorIndex(l2g, iBatch)] *
    d_invSqrtElementalMassVector[iDoF + d_ndofsPerCell * iCell];
}
```

Listing 8: Extraction of cell-level multivector combined with the action of $\mathbf{M}^{-1/2}$ for the cell indexed by `iCell` and batch indexed by `iBatch`

```
for (auto i = 0; i < d_ndofsPerCell; ++i)
{
    unsigned int l2g =
    singleVectorGlobalToLocalMap[i + d_ndofsPerCell * iCell];
    if (dofEncountered[l2g])
    y[getMultivectorIndex(l2g, iBatch)] +=
        a * (temp1ov[i] *
        d_invSqrtElementalMassVector[i + d_ndofsPerCell * iCell]);
    else
    {
        dofEncountered[l2g] = true;
        if (isConstrained[l2g]  l2g >= d_nLocalDofs)
        y[getMultivectorIndex(l2g, iBatch)] =
            scalar1 *
            (temp1ov[i] *
            d_invSqrtElementalMassVector[i +
                                         d_ndofsPerCell * iCell]);
        else
        y[getMultivectorIndex(l2g, iBatch)] =
            scalar1 *
            (temp1ov[i] *
               d_invSqrtElementalMassVector[i + d_ndofsPerCell *
                                            iCell]) +
            c * y[getMultivectorIndex(l2g, iBatch)] +
            b * x[getMultivectorIndex(l2g, iBatch)];
    }
}
```

Listing 9: Assembly from cell-level multivector combined with the action of $\mathbf{M}^{-1/2}$ and scaling with a, b and c for the cell indexed by `iCell` and batch indexed by `iBatch`

B.2. Cell-matrix CPU implementation

The extraction and assembly operations in the cell-matrix implementation are also modified to account for the scaling of the data structures $\mathbf{X}$ and $\mathbf{Y}$ with the diagonal matrix $\mathbf{M}^{-1/2}$ and scalar constants a, b and c using a methodology similar to that used for the matrix-free implementation.

B.3. `deal.II` matrix-free CPU implementation

For the `deal.II` implementation we utilize the *pre-* and *post-* operations as described in Kronbichler et al. [41].

```
const double ratio = c / a;
for (auto i = 0; i < numberWaveFunctions; ++i)
    {
    const auto &pre = [&](const unsigned int start_range,
                          const unsigned int end_range) {
        for (int j = start_range; j < end_range; ++j)
        {
            x.block(i).local_element(j) *=
            d_invSqrtMassVector.local_element(j);
        }
        for (int j = start_range; j < end_range; ++j)
        {
            y.block(i).local_element(j) *=
            ratio * d_sqrtMassVector.local_element(j);
        }
    };
    const auto &post = [&](const unsigned int start_range,
                          const unsigned int end_range) {
        for (int j = start_range; j < end_range; ++j)
        {
            y.block(i).local_element(j) *=
            a * d_invSqrtMassVector.local_element(j);
        }
        for (int j = start_range; j < end_range; ++j)
        {
            x.block(i).local_element(j) *=
            d_sqrtMassVector.local_element(j);
            y.block(i).local_element(j) +=
            b * x.block(i).local_element(j);
        }
    };

    dftPtr->matrix_free_data.cell_loop(
        &kohnShamDFTOperatorClass<FEOrder, FEOrderElectro>::
        computeLocalHamiltonianTimesXdealii,
        this,
        y.block(i),
        x.block(i),
        pre,
        post
        );
    }
```

Listing 10: Implementaion of the $\mathbf{Y} = a\mathbf{M}^{-1/2}\mathbf{H}\mathbf{M}^{-1/2}\mathbf{X} + b\mathbf{X} + c\mathbf{Y}$ operation using `deal.II` matrix-free

C. Matrix multivector products – GPU implementations

C.1. Multivector matrix-free GPU implementation

A sustained performance analysis for the multivector matrix-free GPU implementation on an NVIDIA® Tesla® V100 SXM2 16GB varying the total number of vectors n_v and the batchsize b (or n_{t_x}) shows the optimal batchsize for each `FEOrder`. We observe that $b = 8$ for `FEOrder`=6, $b = 4$ for `FEOrder`=7 and $b = 2$ for `FEOrder`=8 exhibits the best performance. Note that for `FEOrder`=7 and 8, batchsize $b = 8$ is represented with "-" in the above table as the kernel launch fails due to exceeding the maximum dynamic shared memory of V100 GPU. The threads launched in the y-direction n_{t_y} for each thread block are used to loop over an index of size

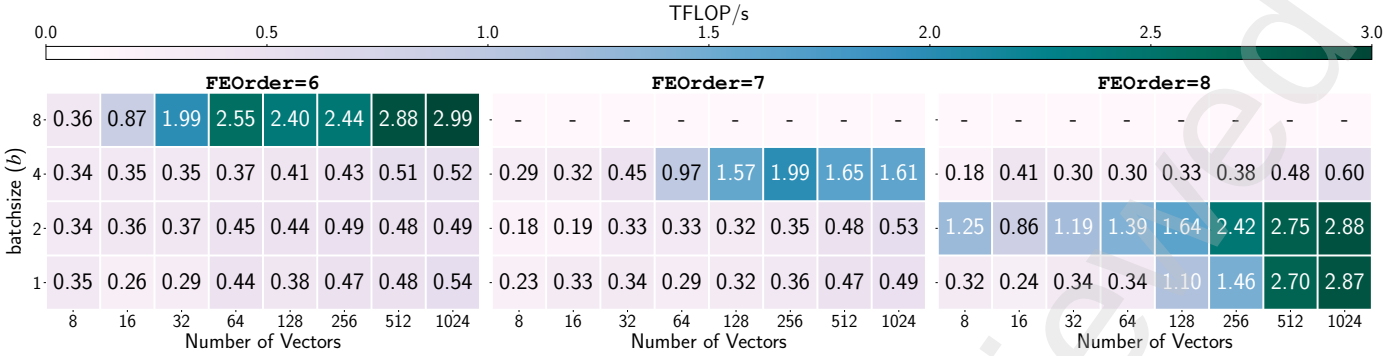


Figure C.23: Performance study of our multivector matrix-free GPU implementation for varying batchsizes on a NVIDIA® Tesla® V100 SXM2 16GB (Summit Supercomputer). Case studies: 117649 DoFs (`FEOrder`=6 and 8); 125000 DoFs (`FEOrder`=7).

n_q^2 (or n_p^2 or $n_p n_q$) which in turn affects the optimal values of n_{t_y} for each `FEOrder`. The optimal value of n_{t_y} is determined by varying n_{t_y} in multiples of `warpSize` (32 for NVIDIA GPUs), and the optimal values are found to be as follows: $n_{t_y} = 64$ for `FEOrder`=6 and 7, and $n_{t_y} = 128$ for `FEOrder`=8. These values of n_{t_x} and n_{t_y} are used to launch the GPU kernel (see Listing 3).

C.2. *Comparison between matrix-free GPU implementation, cell-matrix and `deal.II`'s matrix-free implementations for single-vector*

We note that `deal.II` does not have a multivector matrix-free implementation on GPUs; hence, we compare our single vector matrix-free implementation against `deal.II`'s single vector matrix-free implementation and results are illustrated in Fig. C.24. Speedups of about 16x-18x are observed for our single vector matrix-free implementation for the Helmholtz operator compared to deal.II's matrix-free baseline on a V100 GPU. Our implementation results in even larger speedups of about 19x-25x compared with the cell-matrix approach for a single vector on a V100 GPU.

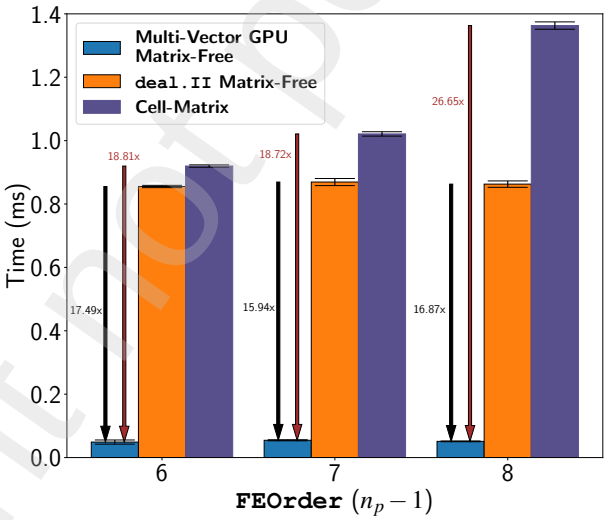


Figure C.24: Comparison of our single-vector matrix-free implementation against `deal.II`'s matrix-free method and the cell-matrix method on a NVIDIA® Tesla® V100 SXM2 16GB (Summit Supercomputer). Case studies: 117649 DoFs (`FEOrder`=6 and 8); 125000 DoFs (`FEOrder`=7).

C.3. *Cell-matrix GPU implementation*

We adopt the BCV layout in the cell-matrix implementation to compute the Helmholtz operator action on a total number of vectors $n_v = 1024$. To this end, a performance study is conducted where the Helmholtz action is evaluated sequentially over batches with varying batchsizes $b = 8, 16, 32, 64, 128, 256$ on 1 to 16 GPU nodes (Summit supercomputer). The resulting sustained performance is shown in Fig. C.25. We note that the time taken for computing the Helmholtz operator action on multivectors with $n_v = 1024$, does not vary appreciably from batchsize $b = 128$ to $b = 256$. We choose $b = 256$ as the batchsize for performing all the benchmark studies since it gives the best sustained performance out of other batchsizes considered in the study.

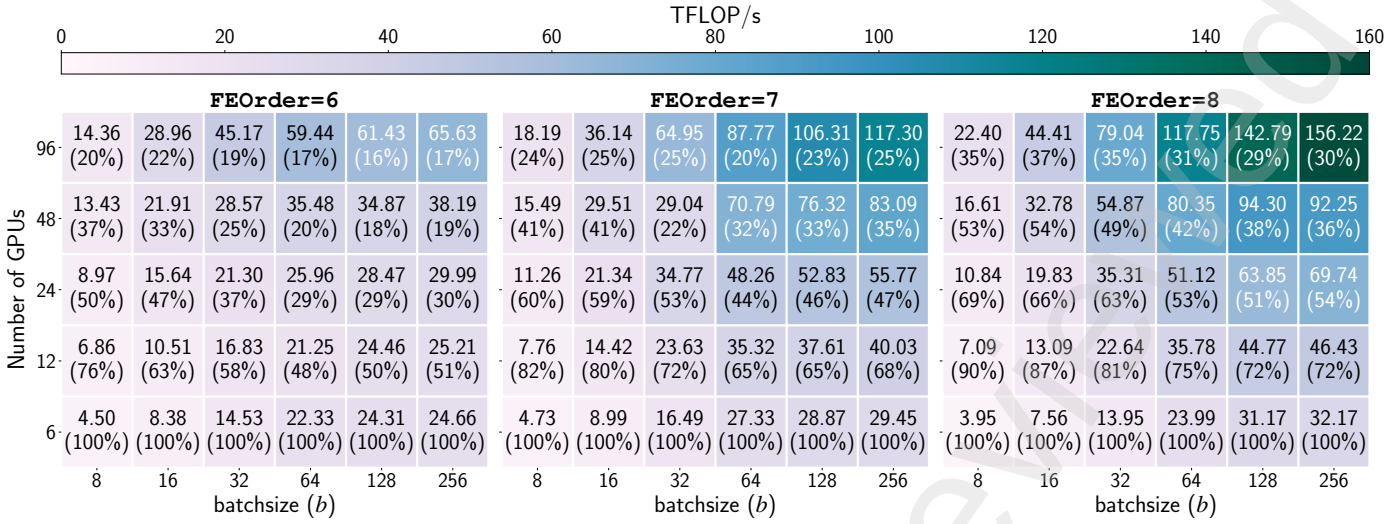
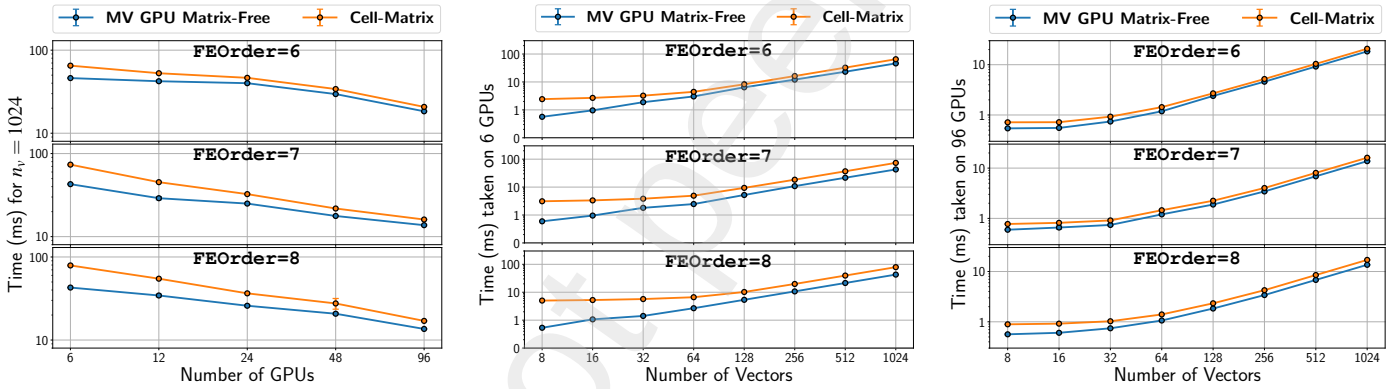


Figure C.25: Performance study of the cell-matrix GPU implementation for various batchsizes on 1 to 16 nodes of Summit supercomputer. Case studies: 1092727 DoFs (FEOrder=6); 1191016 DoFs (FEOrder=7); 1157625 DoFs (FEOrder=8) for the Helmholtz problem and $n_v = 1024$ on GPUs.

C.4. *Performance comparisons for $n_q = n_p + 2$*



(a) Comparative scaling study of our implementation against the cell-matrix method for 1024 vectors.

(b) Performance benchmark of our implementation against the cell-matrix method on 1 node.

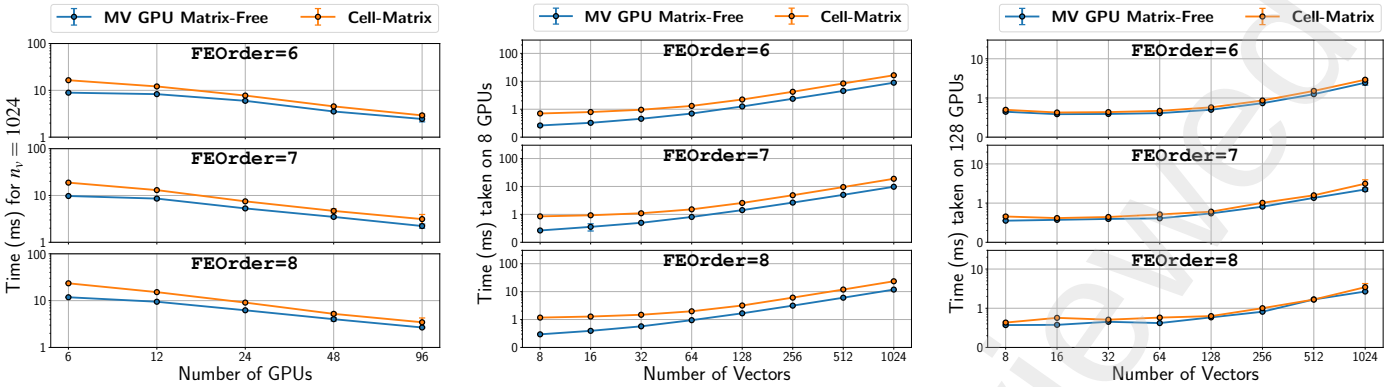
(c) Performance benchmark of our implementation against the cell-matrix method on 16 nodes.

Figure C.26: Benchmarks of our matrix-free implementation with cell-matrix implementation for the case of $n_q = n_p + 2$ and uniform mesh. Case studies: 1092727 DoFs (FEOrder=6); 1191016 DoFs (FEOrder=7); 1157625 DoFs (FEOrder=8) for the Helmholtz problem on NVIDIA® Tesla® V100 SXM2 16GB (Summit Supercomputer).

Choosing a FE mesh of around $1.2m$ DoFs, we conducted a strong scaling study of the proposed matrix-free multivector implementation in the case of $n_q = n_p + 2$ for 1024 vectors and compared it with the cell-matrix approach. Fig. C.26a shows the time to solution for this comparative study. Our GPU matrix-free implementation has a noticeable performance advantage over the cell-matrix method across all MPI tasks for FEOrder = 6, 7 and 8. In particular, we show the comparisons (with varying n_v) for 6 and 96 GPUs in Figs. C.26b and C.26c respectively. On a single node we observe speedup of 41% for FEOrder = 6, a speedup of 72% for FEOrder = 7 and a 85% speedup for FEOrder = 8 over the cell-matrix method in the case of 1024 vectors. In the case of 8 vectors on 1 node (6 GPUs, ~200k DoFs/GPU), we observe a speedup of 4.3x for FEOrder = 6, a speedup of 5.2x for FEOrder = 7, and a 9.4x speedup for FEOrder = 8 over the cell-matrix method. On the other extreme, benchmarks for various numbers of vectors on 16 nodes (96 GPUs, ~12k DoFs/GPU) show performance gains of 13% for FEOrder = 6, a speedup of 17% for FEOrder = 7, and around 25% for FEOrder = 8 against the cell-matrix method for 1024 vectors. In the case of 8 vectors, we observe improvements of up to 30% for FEOrder = 6, 7 and around 58% for FEOrder = 8 against the cell-matrix method on 96 GPUs.

C.5. *Multivector matrix-free GPU implementation on Selene supercomputer*

This subsection reports the performance benchmarks obtained using multi-node A100 GPUs on the Selene supercomputer. A single node of the Selene supercomputer has 2 AMD® EPYC™ 7742 64-Core Processors and 8 NVIDIA® A100-SXM4-80GB GPUs with 640 GB HBM2e memory and 156 TFLOP/s performance (A100 FP64). The interconnect is Mellanox® ConnectX®-6 MT28908,



(a) Comparative scaling study of our matrix-free implementation against the cell-matrix method for 1024 vectors.

(b) Performance benchmark of our matrix-free implementation against the cell-matrix method on 1 Selene node.

(c) Performance benchmark of our matrix-free implementation against the cell-matrix method on 16 Selene nodes.

Figure C.27: Benchmarks of our matrix-free implementation with cell-matrix implementation for the case of $n_q = n_p$ and uniform mesh on NVIDIA® Tesla® A100 SXM2 80GB. Case studies: 1092727 DoFs (`FEOrder`=6); 1191016 DoFs (`FEOrder`=7); 1157625 DoFs (`FEOrder`=8) for the Helmholtz problem on GPUs.

the OS is Ubuntu 20.04.3 LTS and compilers gcc 11.3.0, nvcc 11.8 and Open MPI 4.1.5 with flags `-O3 -arch=sm_70 -lcublas`. Employing an uniform mesh with $n_p = n_q$ comprising ∼1.2m DoFs, a strong scaling study is conducted to compare the proposed matrix-free multivector implementation with the cell-matrix approach in the case of 1024 vectors as shown in Fig. C.27a. Our GPU matrix-free implementation has a noticeable performance advantage over the cell-matrix method across all MPI tasks for `FEOrder` = 6, 7, and 8. In particular, we show the comparisons in more detail (with varying n_v) for 1 Selene node (8 GPUs, ∼150k DoFs/GPU) and 16 Selene nodes (128 GPUs, ∼9k DoFs/GPU) in Figs. C.27b and C.27c respectively. On a single Selene node (8 GPUs, ∼150k DoFs/GPU), we observe speedups of up to 1.8x-1.9x for `FEOrders`=6, 7, 8 over the cell-matrix approach in the case of 1024 vectors. In the case of 8 vectors, we observe a speedup of around 3x-4x for `FEOrders`=6, 7, 8 over the cell-matrix approach on the single GPU node. On the other extreme of 16 Selene nodes (128 GPUs, ∼9k DoFs/GPU), benchmarks for various number of vectors (Fig. C.27c) show performance gains of up to 19% for `FEOrder` = 6, a speedup of 41% for `FEOrder` = 7, and a speedup of 29% for `FEOrder` = 8 against the cell-matrix method for 1024 vectors.

D. Eigensolver GPU implementations using ChFSI

On GPUs, the eigensolver employing ChFSI approach has been implemented similar to CPUs following the steps outlined in Appendix B.1.

References

[1] R. C. Kirby, M. Knepley, A. Logg, L. R. Scott, Optimizing the evaluation of finite element matrices, SIAM Journal on Scientific Computing 27 (2006) 741–758. URL: https://epubs.siam.org/terms-privacy. doi:10.1137/040607824.

[2] T. J. R. Hughes, The Finite Element Method: Linear Static and Dynamic Finite Element Analysis, Dover Civil and Mechanical Engineering, Dover Publications, 2012. URL: https://books.google.co.in/books?id=cHH2n_qBK0IC.

[3] G. F. Carey, E. Barragy, R. McLay, M. Sharma, Element-by-element vector and parallel computations, Communications in Applied Numerical Methods 4 (1988) 299–307. URL: https://onlinelibrary.wiley.com/doi/10.1002/cnm.1630040303. doi:10.1002/cnm.1630040303.

[4] T. J. Hughes, R. M. Ferencz, J. O. Hallquist, Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients, Computer Methods in Applied Mechanics and Engineering 61 (1987) 215–248. doi:10.1016/0045-7825(87)90005-3.

[5] C. Cantwell, S. Sherwin, R. Kirby, P. Kelly, From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements, Computers & Fluids 43 (2011) 23–28. doi:10.1016/j.compfluid.2010.08.012.

[6] P. Motamarri, S. Das, S. Rudraraju, K. Ghosh, D. Davydov, V. Gavini, DFT-FE – A massively parallel adaptive finite-element code for large-scale density functional theory calculations, Computer Physics Communications 246 (2020) 106853. URL: https://linkinghub.elsevier.com/retrieve/pii/S0010465519302309. doi:10.1016/j.cpc.2019.07.016.

[7] S. Das, P. Motamarri, V. Subramanian, D. M. Rogers, V. Gavini, DFT-FE 1.0: A massively parallel hybrid CPU-GPU density functional theory code using finite-element discretization, Computer Physics Communications 280 (2022) 108473. URL: https://linkinghub.elsevier.com/retrieve/pii/S0010465522001928. doi:10.1016/j.cpc.2022.108473.

[8] K. Ljungkvist, Matrix-Free Finite-Element Computations on Graphics Processors with Adaptively Refined Unstructured Meshes, in: Proceedings of the 25th High Performance Computing Symposium, HPC '17, Society for Computer Simulation International, San Diego, CA, USA, 2017, pp. 1–12.

[9] M. Kronbichler, K. Kormann, A generic interface for parallel cell-based finite element operator application, Computers & Fluids 63 (2012) 135–147. URL: https://linkinghub.elsevier.com/retrieve/pii/S0045793012001429. doi:10.1016/j.compfluid.2012.04.012.

[10] D. Davydov, J. Pelteret, D. Arndt, M. Kronbichler, P. Steinmann, A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid, International Journal for Numerical Methods in Engineering 121 (2020) 2874–2895. URL: https://onlinelibrary.wiley.com/doi/10.1002/nme.6336. doi:10.1002/nme.6336.

[11] P. Fischer, M. Min, T. Rathnayake, S. Dutta, T. Kolev, V. Dobrev, J.-S. Camier, M. Kronbichler, T. Warburton, K. Świrydowicz, J. Brown, Scalability of high-performance PDE solvers, The International Journal of High Performance Computing Applications 34 (2020) 562–586. URL: http://journals.sagepub.com/doi/10.1177/1094342020915762. doi:10.1177/1094342020915762.

[12] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, The deal.II finite element library, 2019. URL: https://www.dealii.org/.

[13] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J. S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, S. Zampini, MFEM: A modular finite element methods library, Computers {\&} Mathematics with Applications 81 (2021) 42–74. doi:10.1016/J.CAMWA.2020.06.009.

[14] J. Brown, A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, V. Dobrev, Y. Dudouit, L. Ghaffari, T. Kolev, D. Medina, W. Pazner, T. Ratnayaka, J. Thompson, S. Tomov, libCEED: Fast algebra for high-order element-based discretizations, Journal of Open Source Software 6 (2021) 2945. doi:10.21105/joss.02945.

[15] K. Świrydowicz, N. Chalmers, A. Karakus, T. Warburton, Acceleration of tensor-product operations for high-order finite element methods, The International Journal of High Performance Computing Applications 33 (2019) 735–757. URL: http://journals.sagepub.com/doi/10.1177/1094342018816368. doi:10.1177/1094342018816368.

[16] J. Sun, A. Zhou, Finite Element Methods for Eigenvalue Problems, Chapman and Hall/CRC, 2016. URL: https://www.taylorfrancis.com/books/9781482254655. doi:10.1201/9781315372419.

[17] E. Tsuchida, M. Tsukada, Adaptive finite-element method for electronic-structure calculations, Physical Review B - Condensed Matter and Materials Physics 54 (1996) 7602–7605. doi:10.1103/PhysRevB.54.7602.

[18] K. Ghosh, H. Ma, V. Gavini, G. Galli, All-electron density functional calculations for electron and nuclear spin interactions in molecules and solids, Physical Review Materials 3 (2019) 43801. URL: https://link.aps.org/doi/10.1103/PhysRevMaterials.3.043801. doi:10.1103/PhysRevMaterials.3.043801.

[19] T. Martynova, G. Muratova, P. Oganesyan, O. Shtein, The Numerical Solution of Large-Scale Generalized Eigenvalue Problems Arising from Finite-Element Modeling of Electroelastic Materials, Symmetry 15 (2023) 171. doi:10.3390/sym15010171.

[20] X. Fan, P. Chen, R. Wu, S. Xiao, Parallel computing study for the large-scale generalized eigenvalue problems in modal analysis, Science China Physics, Mechanics and Astronomy 57 (2014) 477–489. doi:10.1007/s11433-013-5203-5.

[21] X. Fan, K. Wang, S. Xiao, Q. Liu, Z. Mo, Some Progress on Parallel Modal and Vibration Analysis Using the JAUMIN Framework, Mathematical Problems in Engineering 2015 (2015) 1–8. doi:10.1155/2015/253569.

[22] S. Markidis, The Old and the New: Can Physics-Informed Deep-Learning Replace Traditional Linear Solvers?, Frontiers in Big Data 4 (2021). doi:10.3389/fdata.2021.669097.

[23] N. Beams, A. Abdelfattah, S. Tomov, J. Dongarra, T. Kolev, Y. Dudouit, High-Order Finite Element Method using Standard and Device-Level Batch GEMM on GPUs, in: Proceedings of ScalA 2020: 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, Held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 53–60. doi:10.1109/ScalA51936.2020.00012.

[24] D. Davydov, M. Kronbichler, Algorithms and Data Structures for Matrix-Free Finite Element Operators with MPI-Parallel Sparse Multi-Vectors, ACM Transactions on Parallel Computing 7 (2020). URL: http://arxiv.org/abs/1907.01005. doi:10.1145/3399736.

[25] D. A. Kopriva, Implementing Spectral Methods for Partial Differential Equations, Scientific Computation, Springer Netherlands, Dordrecht, 2009. URL: http://link.springer.com/10.1007/978-90-481-2261-5. doi:10.1007/978-90-481-2261-5.

[26] A. Solomonoff, A fast algorithm for spectral differentiation, Journal of Computational Physics 98 (1992) 174–177. URL: https://linkinghub.elsevier.com/retrieve/pii/002199919290182X. doi:10.1016/0021-9991(92)90182-X.

[27] S. Das, P. Motamarri, V. Gavini, B. Turcksin, Y. W. Li, B. Leback, Fast, Scalable and Accurate Finite-Element Based Ab Initio Calculations Using Mixed Precision Computing: 46 PFLOPS Simulation of a Metallic Dislocation System, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, volume 11, ACM, New York, NY, USA, 2019, pp. 1–11. URL: https://dl.acm.org/doi/10.1145/3295500.3357157. doi:10.1145/3295500.3357157.

[28] S. C. Brenner, L. R. Scott, The Mathematical Theory of Finite Element Methods, volume 15 of *Texts in Applied Mathematics*, Springer New York, New York, NY, 2008. URL: http://link.springer.com/10.1007/978-0-387-75934-0. doi:10.1007/978-0-387-75934-0.

[29] W. Bangerth, O. Kayser-Herold, Data structures and requirements for hp finite element software, ACM Transactions on Mathematical Software 36 (2009) 1–31. URL: https://dl.acm.org/doi/10.1145/1486525.1486529. doi:10.1145/1486525.1486529.

[30] M. O. Deville, P. F. Fischer, E. H. Mund, High-Order Methods for Incompressible Fluid Flow, Cambridge University Press, 2002. URL: https://www.cambridge.org/core/product/identifier/9780511546792/type/book. doi:10.1017/CBO9780511546792.

[31] G. Fedorov, L. Huot, Intel® Math Kernel Library Improved Small Matrix Performance Using Just-in-Time (JIT) Code Generation for Matrix Multiplication (GEMM), 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/onemkl-improved-small-matrix-performance-using-just-in-time-jit-code.html.

[32] Gordon Bell Prize Finalists Named — Careers — Communications of the ACM, 2019. URL: https://cacm.acm.org/careers/

240486-gordon-bell-prize-finalists-named/fulltext.
[33] P. Motamarri, M. Nowak, K. Leiter, J. Knap, V. Gavini, Higher-order adaptive finite-element methods for Kohn–Sham density functional theory, Journal of Computational Physics 253 (2013) 308–343. URL: https://linkinghub.elsevier.com/retrieve/pii/S0021999113004774. doi:10.1016/j.jcp.2013.06.042.
[34] D. Arndt, W. Bangerth, M. Feder, M. Fehling, R. Gassmöller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J. P. Pelteret, S. Sticko, B. Turcksin, D. Wells, The deal.II library, Version 9.4, Journal of Numerical Mathematics 30 (2022) 231–246. doi:10.1515/JNMA-2022-0054.
[35] C. Burstedde, L. C. Wilcox, O. Ghattas, p4est : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees, SIAM Journal on Scientific Computing 33 (2011) 1103–1133. URL: http://epubs.siam.org/doi/10.1137/100791634. doi:10.1137/100791634.
[36] T. Gruber, J. Eitzinger, G. Hager, G. Wellein, LIKWID, 2022. URL: https://doi.org/10.5281/zenodo.7432487. doi:10.5281/zenodo.7432487.
[37] Y. Zhou, Y. Saad, M. L. Tiago, J. R. Chelikowsky, Self-consistent-field calculations using Chebyshev-filtered subspace iteration, Journal of Computational Physics 219 (2006) 172–184. doi:10.1016/J.JCP.2006.03.017.
[38] D. P. O'Leary, The block conjugate gradient algorithm and related methods, Linear Algebra and its Applications 29 (1980) 293–322. URL: https://linkinghub.elsevier.com/retrieve/pii/0024379580902475. doi:10.1016/0024-3795(80)90247-5.
[39] P. Hohenberg, W. Kohn, Inhomogeneous Electron Gas, Physical Review 136 (1964) B864–B871. URL: https://link.aps.org/doi/10.1103/PhysRev.136.B864. doi:10.1103/PhysRev.136.B864.
[40] W. Kohn, L. J. Sham, Self-consistent equations including exchange and correlation effects, Physical Review 140 (1965). doi:10.1103/PhysRev.140.A1133.
[41] M. Kronbichler, D. Sashko, P. Munch, Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations, https://doi.org/10.1177/10943420221107880 (2022). URL: https://journals.sagepub.com/doi/abs/10.1177/10943420221107880. doi:10.1177/10943420221107880.