An efficient matrix-free implementation for finite-element basis overlap matrix-multivector products on hybrid CPU-GPU architectures

Gourab Panigrahi, Nikhil Kodali, Debashis Panda, Phani Motamarri
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India
*phanim@iisc.ac.in

Abstract—Recent hardware-aware algorithms for finite-element matrix-vector multiplications suggest that on-the-fly matrix-vector products suggest that on-the-fly matrix-vector products reduce both arithmetic complexity and the data access costs. The current implementations of such matrix-free algorithms deal only with a single vector and are not readily applicable for matrix multivector products. We propose an efficient implementation procedure for the matrix-free algorithm to compute FE discretized matrix-multivector products on hybrid CPU-GPU architectures.

I. INTRODUCTION

Finite element methods are widely used in science and engineering to solve problems of practical interest. In the finite-element (FE) framework, these problems usually involve the construction of a FE discretized operator and often require either computing their action on an FE discretized fields or solving (usually utilizing iterative solvers) for the FE discretized fields given the action of the operator on them. In both these cases evaluating the product of the sparse matrix (FE discretized operator) and the vector (FE discretized field) product is the computationally dominant step. Traditionally, these matrix-vector products are computed directly by sparse-matrix vector multiplication modules. Previously, it has been noted that the evaluation of such matrix-vector products can be done efficiently on multi-threaded architectures using cell-level dense matrix-vector multiplications followed by the assembly of cell-level product vectors [1]. Recently this strategy has been employed for evaluating matrix-multivector products at the cell-level on both CPU only and hybrid CPU-GPU architectures, demonstrating good throughput performance. Recent hardware-aware algorithms for evaluating such matrix-vector multiplications suggest that on-the-fly matrix-vector products without storing the cell-level matrices reduce both arithmetic complexity and memory footprint [2] [3] [4]. These algorithms exploit the tensor-structured nature of the finite-element basis functions and recast the 3D integrals involved in the matrix-vector products as a sequence of tensor contractions.

Currently available open-source implementations of such methods [5] [6] [7] are not optimized for the case of an FE discretized operator acting on multiple FE discretized fields, while such situations are often encountered in ab-initio material modeling problems [8] [9]. Although some work has been done in this regard [10] [11], no widely available implementation procedures exist for performing generic matrix-multivector efficiently under the matrix-free paradigm.

This work proposes an efficient implementation procedure for the matrix-free algorithm to compute such matrix-multivector products on hybrid CPU-GPU architectures. We employ efficient vendor BLAS routines on CPUs to perform the underlying tensor contractions. On GPUs, the proposed implementation utilizes the concept of kernel fusion to minimize data access and efficiently utilizes the GPU shared memory and registers to pipeline data access and computation. Further, we also present a data layout for storing the FE discretized global multivector to minimize the non-contiguous data accesses during the extraction of the cell-level vector from the global vector resulting in efficient parallelization.

Finally, we assess the performance of our implementation to compute FE basis overlap (mass) matrix times multivector multiplications. For problem sizes commonly encountered in ab-initio material modeling, we find that computational gains up to 7x are achieved for matrix-multivector products using the proposed implementation compared to the FE cell-level matrix-multiplication approach. We further benchmark our performance against the matrix-free module in the deal.II library for the case of a single vector, and observe speedups up to 2.5x. We also present the results of a preliminary investigation of MPI scaling on both CPUs and GPUs.

II. BACKGROUND

We will present the matrix-free approach with a focus on the FE basis overlap matrix, also known as the mass matrix (M) but the methods presented are extensible to generic FE operators.

The mass matrix is defined as $M_{ij} = \int_{\Omega} \hat{\phi}_i \hat{\phi}_j d\tau$ where $\hat{\phi}_i$ are the FE basis functions and $\Omega$ is the domain volume. The cell level mass matrix is written as $M_{ij} = \int_{\Omega_c} \hat{\phi}_i \hat{\phi}_j d\tau = \int_{\Omega_c} \hat{\phi}_i \hat{\phi}_j [J]^T d\tau$ where $\hat{\phi}_i$ are the FE basis functions on reference cell and $J$ is the Jacobian matrix for the transformation from FE domain cell to the reference cell.

The matrix-free method relies on the evaluation of the cell level matrix-vector $Mu$ product based on the Gauss quadrature formula to compute the integral involved in $M_{ij}$
as shown below

$$(Mu)_i = \sum_j M_{ij} u_j = \sum_q \left[ J_i | w_q \phi_i(\xi_q) \sum_j \phi_j(\xi_q) u_j \right]$$

where $\xi_q$ are the quadrature points and $w_q$ are the quadrature weights.

For the specific choice of a tensor product structured basis, which is often the case in FE discretization, the arithmetic complexity, and the memory footprint can be further reduced.

As an illustration we can consider the 2D case. For a tensor product structured basis we can write our basis function as a product of 1D basis functions, i.e., $\phi_i(\xi) = \phi_{i1}(\xi_1) \phi_{i2}(\xi_2)$.

The above summation can then be rewritten as

$$
(Mu)_{i1i2} = \sum_{q_1} [\phi_{i1}(\xi_{q1})]_i [\phi_{i2}(\xi_{q2})]_j \sum_{q_2} [\phi_{j1}(\xi_{q1})]_j [\phi_{j2}(\xi_{q2})]_j \left( u_{j1} J_{q1} q_1 \right) 
$$

Suppose $FEOrder = p - 1$ denotes the order of the FE interpolating polynomial (FEOrder). In that case, this expression has a computational complexity of $O(p^3)$ as it is equivalent to 4 matrix multiplications of $p \times p$ with $p \times p$ and including Jacobian and quadrature weights, it is three floating-point operations for every $p^2$ cells. So, a total of $4p^2(2p - 1) + 3p^2$ floating point operations which is of $O(p^3)$. We can also see from equation (2) that only the determinant of the Jacobian matrix and the values of the shape function at $p$ quadrature points (assuming that the number of quadrature points is the same as base size) need to be stored for each cell, this results in a memory footprint of $O(p^2)$.

This analysis can be generalized to arbitrary dimensions. If $d$ is the number of dimensions then the matrix-free methods results in a memory footprint of $O(p^2)$ and arithmetic complexity of $O(p^{d+1})$ as opposed to the cell-matrix method where we have a memory footprint of $O(p^{2d})$ (to store a $p^d \times p^d$ matrix) and arithmetic complexity of $O(p^{2d})$ (multiplication of $p^d \times p^d$ matrix with a $p^d \times 1$ vector). Note that while we have illustrated the matrix-free method for the 2D case, the rest of the work will deal with three dimensions.

III. METHODOLOGY AND IMPLEMENTATION

The algorithm comprises of three phases - extraction, compute, and assembly as seen in figure 1. In extraction step, the input node level multivector $X$ of size $N_{vecs} \times N_{dof}$ is extracted to form the cell level multivector $U$ of size $N_{vecs} \times p^3 \times N_{cells}$ which is done using a map. The compute phase involves 6 tensor contractions (equations 3 and 5) and an element-wise multiplication with the determinant of the Jacobian and the quadrature weights (equation 4). After the compute, the output of contractions is the cell level multivector product $V$ which has the same dimensions as $U$ and is then assembled back into the output node level multivector $Y$ of size $N_{vecs} \times N_{dof}$ using the same map.

$$A_{q1q2q3k} = P_{j1} P_{j2} P_{j3} U_{j1j2j3k} \quad (3)$$
$$K_{q1q2q3} = A_{q1q2q3} | J_{q1q2q3} | \quad (4)$$
$$V_{i1j2j3k} = P_{i1} P_{i2} P_{i3} K_{q1q2q3k} \quad (5)$$

Einstein summation convention is assumed in equations (3) and (5).

Two layouts for storing the global multivector were explored. The first is to store the $N_{vecs} \times N_{dof}$ multivector in column-major order i.e. with $N_{vecs}$ as the contiguous index. This contiguous vector layout allows us to communicate all the vectors at a given dof across MPI tasks as a single contiguous data block. The other layout is to reshape the multivector into $N_k \times N_{dof} \times N_{Batch}$, as shown in figure 2, where $N_k$ is chosen according to the FEOrder used and $N_{Batch} = N_{vecs}/N_k$. This batched contiguous vector layout allows us to overlap computation and communication as the batches are independent. We find that the batched contiguous vector layout shows better performance, as shown in figure 3, in the case of a single MPI task on CPUs and GPUs. The impact on its performance due to non-contiguous MPI communication is yet to be investigated.

A. CPU Implementation

Each tensor contraction is rewritten as matrix multiplication, and $dgemm$ modules from vendor BLAS libraries are used.
to perform these contractions. Note that the cell level multivector is of size \( p^3 \times N_{vecs} \times N_{cells} \) and the way the matrix multiplications are written allows us to explore various blocking/tiling patterns over both \( N_{vecs} \) and \( N_{cells} \) indices. For the problem sizes considered, it is found that in the case of the contiguous vector layout, the most efficient way is to perform the matrix multiplications in a loop over the \( N_{cells} \) index. In contrast, for the batched contiguous vector layout, tiling over the \( N_{vecs} \) index with a tile length of \( N_k \) gives the best performance. It is also ensured that the matrix multiplications are written such that the \( \text{dgemm} \) calls can efficiently use the CPU cache. For the the \( \text{dgemm} \) calls involved in equation (5), this requires a reshaping of the \( K \) tensor in equation (4). It is noted that there is a trade-off between the cost of reshaping and sub-optimal use of cache by the subsequent \( \text{dgemm} \) calls, and the conclusions of this investigation are presented in section IV.

B. GPU Implementation

The three steps in matrix-free multiplication can be performed in a single kernel launch to avoid unnecessary data movement and device memory accesses as only the final node level vector \( Y \) needs to be written back. This single kernel launch does not explicitly construct the cell level multivectors \( U \) and \( V \) in the device memory, thus reducing memory footprint compared to the cell-matrix approach.

The matrix-free matrix multivector multiplication kernel is launched with a 3-D grid of \( N_{cells} \times N_{Batch} \times 1 \) threadblocks. The threadblocks are launched with \( N_k \) threads in the x-direction to handle the \( N_k \) vectors. In the y-direction, the threads handle the rows of the second matrix involved in tensor contraction. As will be explained in section IV, the choice of \( N_k \) and the number of threads launched in the y-direction depend on the FEOrder of the problem.

In the compute step, the tensor contractions are executed as linear combinations of columns of the shape function \( P \) matrix as described in figure 4. The tensors involved in the compute: \( P_{p \times p} \) and \( U_{p^3 \times N_k} \) are small in size and can be fit in the shared memory. Thereby \( P \) can be reused for the subsequent tensor contractions. As the first matrix \( P \) is present in the shared memory before starting the first tensor contraction, the same can be started as soon as data from \( X \) is read without waiting for the whole data of the second matrix involved in the tensor contraction. This method of tensor contractions utilizes each data access to its full compute potential and avoids rereading the same data.

The convention followed for the layouts of all the tensors involved is that the first index is contiguous in memory. Thus the tensor contractions in each threadblock are done over the outermost index of the second tensor \( U_{j_1 j_2 j_3} \) for all \( N_k \) vectors. Each tensor contraction’s output is reshaped to compute the next tensor contraction correctly.

Furthermore, as each thread accesses the same values from the \( P \) matrix, the accesses get broadcasted, which helps reduce bank conflicts in shared memory. To further improve performance, registers are utilized to keep the compute local to each thread as much as possible. This reduces data movement even from shared memory and better utilizes the hardware.

Finally, in the assembly step \( \text{atomicAdd} \) is used to avoid race conditions and safely assemble the output node level multivector \( Y \).

The tensor contractions are performed similarly in both the contiguous vector layout and batched contiguous vector layout.

IV. RESULTS

The CPU performance and timings were collected on Intel® Xeon Gold 6248R CPUs, and the GPU performance analysis was done on Tesla V100-SXM2-32GB. Each measurement was repeated 100 times and averaged over. The problem sizes per GPU or MPI task considered in the following analysis represent the ideal scaling regime for the finite-element ab-initio simulation code DPT-PE [8]. However, the qualitative behavior of the results would not change irrespective of the problem size one chooses. Higher order FEOrders 6, 7, and 8 were chosen for all the analyses on CPUs and GPUs to be consistent with the use of higher order finite-elements in ab-initio modeling of materials.

The proposed matrix-free implementation is benchmarked against the cell-matrix method which is implemented by utilizing the \( \text{dgemm}_\text{batch}_\text{strided} \) from Intel® MKL version 2021.2.0 on CPUs and \( \text{cublasDgemmStridedBatched} \) CUDA 11.0 on GPUs for computing the \( H_{p^3 \times p^3} \times U_{p^3 \times N_{vecs}} \) dense matrix-matrix product for every cell.

Our matrix-free implementation is also compared with the existing matrix-free implementation in deal.II library [5]. The deal.II implementation of matrix-free currently only
exists for single vectors. Hence the comparison is only made for the case of a single vector.

A. CPU

Performance for the batched contiguous vector layout was studied by varying the batch size $N_k$, and it has been observed that best performance is achieved for a batch size of around $N_k = 20$ for the FEOrders and problem sizes considered.

![Fig. 5: Timings and sustained performance for various polynomial orders on a single CPU task.](image)

The plots in figure 5 show that our CPU matrix-free implementation for the mass matrix is faster than the cell-matrix method for both cases of single and multivectors. In comparison to existing matrix-free implementation in deal.II library, our implementation has approximately 1.5x speedup for FEOrder 6, 7, and 8 for single vector case. The single vector comparison with cell-matrix shows that our implementation has 23.75x speedup for FEOrder 6, 54x speedup for FEOrder 7, and 93.7x speedup for FEOrder 8. This speedup is due to the cell matrix being bounded by memory transactions in the extraction and assembly steps and low floating-point operations in the compute step.

The strong scaling studies in figure 6 are performed with the contiguous vector layout as the batched vector layout is still being investigated for its non-contiguous MPI communication. The sizes chosen are 100 vectors and 512 cells for all the FEOrders 6, 7, and 8.

B. GPU

Two matrix-free implementations were first compared. One uses the cuBLAS library to perform the six tensor contractions, and the other uses a naive shared memory implementation. The shared memory implementation was faster as it avoids multiple reads from and writes to the global memory in making the cuBLAS calls, increasing the overall performance.

![Fig. 6: Speedups for various polynomial orders on multiple MPI tasks.](image)

After the optimizations, for each FEOrder, $\text{blockDim.x}$ is varied in multiples of 32 to test for best performance, and then $\text{blockDim.y}$ which is the same as $N_k$ is varied to study the same. The default shared memory limit of the V100 GPU is 48 kB which limits the value of $N_k$ and consequently the number of threads that can be launched in the x-direction. For varying $N_k$ in the default shared memory limit, the static shared memory of declaring the arrays __shared__ is used. And for further increasing $N_k$, the shared memory limit is increased by using the dynamic memory method to call the API

```c
cudaFuncSetAttribute(kernel, cudaMemcpyDefault, cudaMemcpyDefault, shared_memory_size);
```

Additionally, performance was tested by both storing the map in shared memory and by accessing it directly from device memory to save space in shared memory.

The final analysis was done using the best implementation for each FEOrder: $N_k = 5$ for FEOrder $7$ and $N_k = 4$ for other FEOrders. The map is stored in shared memory for FEOrder $6$ and accessed from the device memory for other FEOrders. Static shared memory is used for all FEOrders.

The plots in figure 7 show that our GPU matrix-free implementation for the mass matrix is faster than the cell-matrix method for both cases of single and multivectors. In the multivector case, 5x speedups for FEOrder 6 and 7, and a 7x speedup for FEOrder 8 over the cell-matrix route were observed.

In comparison to existing matrix-free implementation in deal.II library, our implementation has 1.5x speedup for FEOrder 6, 1.9x for 7, and a 2.5x speedup for FEOrder 8 for single vector case. The same single vector comparison with cell-matrix, shows that our implementation has 23x speedup for FEOrder 6, 39x speedup for FEOrder 7, and 80x speedup...
As mentioned previously, the strong scaling studies in figure 8 are performed with the contiguous vector layout. The problem sizes chosen are 100 vectors and 1728 cells for all the FEOrders 6, 7 and 8. And similar to batched vector layout, the most optimized contiguous vector layout was used.

The speedups with respect to 1 GPU are higher in case of cellmatrix than matrixfree approach due to higher floating point operations which hide latency incurred when increasing the GPUs.

But, the decreased memory footprint of the proposed implementation of matrixfree approach enables it to run on 1 GPU for FEOrder 8 whereas the same case fails for cellmatrix approach due to memory limitations.

V. Conclusions

In conclusion, we have illustrated and implemented an efficient matrix-free algorithm for mass matrix multivector multiplication. We have also shown significant speedups our implementation provides compared to the traditional cellmatrix method. We also note that our matrix multivector multiplication implementation is significantly more efficient than the deal.II matrix-free implementation and the cellmatrix method even for the single vector case.

The future work involves comparing our implementation to the matrix-free implementation in MFEM [6]. We also intend to extend the implementation to work with the Laplacian operator to implement the method in the Poisson solver in DFT-FE [8]. We would also like to explore the efficiency of an MPI implementation of the algorithm.

REFERENCES

[1] G. F. Carey, E. Bazzagray, R. McEary, and M. Sharma, “Element-by-element vector and parallel computations,” Communications in Applied Numerical Methods, vol. 4, no. 3, pp. 299–307, 5 1988. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cnm.1630040303

K. Ljungkvist, “Matrix-Free Finite-Element Computations On Graphics Processors with Adaptively Refined Unstructured Meshes,” in 25th High Performance Computing Symposium (HPC 2017), vol. 49, no. 3. Society for Modeling and Simulation International (SCS), 2017, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3108097

[3] M. Kronbichler and K. Kormann, “A generic interface for parallel cell-based finite element operator application,” Computers & Fluids, vol. 63, pp. 135–147, 6 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0099695912000142

[4] D. Davydov, J. Pelteret, D. Arndt, M. Kronbichler, and P. Steinmann, “A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid,” International Journal for Numerical Methods in Engineering, vol. 121, no. 13, pp. 2874–2895, 7 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/nme.6336

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

[6] R. Anderson, J. Andrej, A. Barker, J. Brunwell, J. S. Camier, J. Cerveny, V. Dobrev, Y. Doudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, “MFEM: A modular finite element methods library,” Computers & Mathematics with Applications, vol. 81, pp. 42–74, 1 2021.

[7] J. Brown, A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, V. Dobrev, Y. Doudouit, L. Ghaifari, T. Kolev, D. Medina, W. Pazner, T. Ratnayaka, J. Thompson, and S. Tomov, “libCEED: Fast algebra for high-order element-based discretizations,” Journal of Open Source Software, vol. 6, no. 63, p. 2945, 7 2021.

[8] S. Das, P. Motamarri, V. Subramanian, D. M. Rogers, and V. Gavini, “DFT-FE 1.0: A massively parallel hybrid CPU-GPU density functional theory code using finite-element discretization,” Computer Physics Communications, vol. 280, p. 108473, 11 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465522001928

[9] E. Tsuchida and M. Tsukada, “Adaptive finite-element method for electronic-structure calculations,” Physical Review B - Condensed Matter and Materials Physics, vol. 54, no. 11, pp. 7602–7605, 1996.

[10] D. Davydov and M. Kronbichler, “Algorithms and Data Structures for Matrix-Free Finite Element Operators with MPI-Parallel Sparse Multi-Vectors,” ACM Transactions on Parallel Computing, vol. 7, no. 3, 7 2020. [Online]. Available: http://arxiv.org/abs/1907.01005
[11] N. Beams, A. Abdelfattah, S. Tomov, J. Dongarra, T. Kolev, and 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.