Computing (2020) 102:1941–1965
https://doi.org/10.1007/s00607-020-00827-4

REGULAR PAPER

GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices

Utpal Kiran[1] · **Sachin Singh Gautam**[1] · **Deepak Sharma**[1]

Received: 4 January 2020 / Accepted: 10 June 2020 / Published online: 24 June 2020

Abstract

Matrix-free solvers for finite element method (FEM) avoid assembly of elemental matrices and replace sparse matrix-vector multiplication required in iterative solution method by an element level dense matrix-vector product. In this paper, a novel matrix-free strategy for FEM is proposed which computes element level matrix-vector product by using only the symmetric part of the elemental matrices. The proposed strategy is developed to take advantage of the massive parallelism of Graphics Processing Unit (GPU). A unique data structure is also introduced which ensures localized and coalesced memory access suitable for a GPU while storing only the symmetric part of the elemental matrices. In addition, the proposed strategy emphasizes the efficient use of register cache, uniform workload distribution, reducing thread synchronization, and maintaining sufficient granularity to make the best use of GPU resources. The performance of the proposed strategy is evaluated by solving elasticity and heat conduction problems using 4-noded quadrilateral element with two degrees of freedom (DOFs) and one DOF per node, respectively. The performance is compared with the matrix-free solver strategies on GPU from the literature. It is found that a maximum speedup of 4.9 $\times$ is obtained for the elasticity problem and a maximum of 3.2 $\times$ speedup for the heat conduction problem. Further, the proposed strategy takes the least amount of GPU memory as compared to the existing strategies.

Keywords Matrix-free solver · Finite element method · GPU · CUDA · Parallel computing

✉ Deepak Sharma
dsharma@iitg.ac.in

Utpal Kiran
ukiran@iitg.ac.in

Sachin Singh Gautam
ssg@iitg.ac.in

[1] Department of Mechanical Engineering, Indian Institute of Technology, Guwahati, Assam 781039, India

Mathematics Subject Classification 74S05 · 65Y05

1 Introduction

Finite element method (FEM) is one of the most extensively used numerical methods to solve real-world problems governed by ordinary/partial differential equations. The popularity of FEM is primarily due to its ability to handle complex geometries, high accuracy, and applicability to a wide range of problems. However, FEM can be computationally expensive for complex real-life problems [3,19,22] that require a large number of degrees of freedom (DOFs) to obtain desired solution. Although there has been an exponential increase in computational resources, the computational cost of FEM is still the main bottleneck for many large-scale problems.

In the literature, the high performance computing (HPC) techniques have been used to handle expensive computation required in FEM [41,45]. Recently, Graphics Processing Unit (GPU)-based computation has become immensely popular for HPC implementation. The importance of GPU for HPC applications can be understood from the fact that many supercomputers in Top500 list have GPU on each of its node [42]. It is because GPU is a massively-threaded many-core processor architecture that houses a large number of computational units for efficiently handling parallel workload.

The HPC techniques can be used for FEM since the computation involved can be done in parallel. However, FEM remains computationally expensive due to involvement of various steps. In FEM, a mesh is first created by dividing the problem domain into a number of polygon/polyhedra shaped entities known as elements. The governing field equations are recast into integral form (called the "weak form") and subsequent approximation of the primary variable over each element leads to elemental stiffness and force matrices respectively. The elemental matrices are then assembled into a global matrix using the mesh connectivity matrix. After application of suitable boundary conditions, the assembled matrix is solved using any suitable direct or iterative linear solver to obtain the value of unknown field variables. Every step in FEM procedure can incur significant computational overhead depending on the type of problem being solved. The previous attempts to accelerate each step of FEM on GPU have achieved a significant performance over single and multi-core CPU. The speedup of several folds has been observed in the elemental matrix evaluation [25,31,38,46] and its assembly [10,16,20,34,37,39] to the global matrix. However, it is found that the elemental matrix evaluation and assembly consume smaller fraction of the total computational time as compared to the solution of system of equations. Consequently, the acceleration of linear solver on GPU has received more attention [2,5,24,35]. The iterative solvers are more preferred for solution of a large system of equations. In addition of being memory efficient, the iterative solvers provide abundant amount of parallelism making them suitable for GPU. The main computational components of an iterative solver are sparse matrix-vector product (SpMV), vector-dot product, scalar-vector product and vector-vector addition/subtraction, respectively. Among these, SpMV is the most computationally expensive operation [44]. Since efficient parallel implementation of operations like vector-dot product can be achieved easily, the performance of an iterative solver directly depends on the performance of SpMV operation. A SpMV

operation performs multiplication of a sparse matrix stored in specialized formats (like CSR, COO, ELL, etc. [7]) with a vector. These sparse storage formats reduces the storage requirement of a sparse matrix. However, they introduce irregular memory access pattern that prevents realization of true computational performance of a GPU device. In spite of these challenges, the SpMV operation has benefited greatly by the use of GPU acceleration [4,7,15].

Another way to improve the performance of an iterative solver is to replace SpMV operation of a sparse matrix with a vector by a matrix-free approach. In this approach, the multiplication is performed at the level of smaller dense constituent matrices. Since the elemental matrices are dense and of the same size, the matrix-free approach can provide finer level of parallelism along with regular memory access pattern suitable for GPU. The matrix-free approach was first introduced in [18] for the low memory vector machines and primarily used for solving large FEM problems on microprocessor with limited memory [9]. The recent revival of interest in the matrix-free approach can be attributed to an introduction of massively parallel many-core architectures. Since the advent of CUDA in 2006 the matrix-free approach has been actively pursued by many researchers. GPU-based matrix-free FEM solver has been developed for applications like elasticity [27], heat conduction [21], weather prediction [29], fluid flow [14], and topology optimization [33] among many others. The elemental matrices in the matrix-free solver can be precomputed for all elements in the mesh and stored as dense matrices or it can be recalculated on-the-fly [6,23] during matrix-vector product. As shown in [34], the approach that recalculates elemental matrices becomes highly compute bound for GPU implementation, whereas the best performance is achieved by the local matrix approach for low-order FEM. The local matrix approach is also found to perform better than the assembly-based solver [26,34]. Overall, the superior performance of the matrix-free approach can be ascribed to lesser memory transfer, better access pattern, and fine grain parallelism. However, the performance of the matrix-free approach on GPU is still found to be limited by memory bandwidth.

To the best of author's knowledge studies implementing the matrix-free approach use full elemental matrices for matrix-vector product evaluation. The elemental matrices obtained in FEM are symmetric for most of the problems. Implementing matrix-vector product using the symmetric part of the elemental matrices can significantly reduce the storage requirement as well as data transfer. On the memory bound architectures like GPU reduction in data requirement is expected to improve the performance of a kernel substantially. So far the matrix-free approach has not been implemented on GPU using only the symmetric part of the elemental matrices. Therefore, the main contributions of this paper are as follows.

1. A novel matrix-free solver for FEM is developed which uses only the symmetric part of the elemental matrices.
2. In order to optimize the data access pattern, a unique data structure is developed which ensures coalesced memory access for efficient GPU implementation while storing only the upper triangular part of elemental matrices.
3. Comparative analysis of the proposed solver with the existing matrix-free methods is presented on two test problems using linear quadrilateral elements over unstructured mesh generated through an FEM software package.

The paper is organized as follows. Section 2 presents the existing and the most common matrix-free solvers for FEM and discusses their GPU implementations. In Sect. 3, the proposed matrix-free method is described along with the data structure and access pattern. The performance evaluation and comparison are presented in Sect. 4. Section 5 concludes the paper with scope of future work.

2 Background

2.1 Matrix-free FEM

The finite element discretization produces the global system of algebraic equations

$$\mathbf{KU} = \mathbf{F}, \tag{1}$$

where $\mathbf{K}$ is the sparse global stiffness matrix, $\mathbf{U}$ is the unknown displacement vector and $\mathbf{F}$ is the global extended force vector. The global stiffness matrix and nodal force vector are assembled from the elemental matrices as

$$\mathbf{K} = \underset{e\in\mathcal{E}}{\mathcal{A}}\, \mathbf{K}^e, \tag{2}$$

$$\mathbf{F} = \underset{e\in\mathcal{E}}{\mathcal{A}}\, \mathbf{F}^e, \tag{3}$$

where $\mathbf{K}^e$ is the elemental tangent matrix, $\mathbf{F}^e$ is the elemental force vector, $\mathcal{E}$ is the set of all elements in the mesh and $\mathcal{A}$ is the assembly operator. The iterative approach to solve system of equations (Eq. (1)) requires the multiplication of sparse global stiffness matrix $\mathbf{K}$ with a given vector $\mathbf{y}$, in each iteration step. There are three major strategies by which a matrix-free solver can be implemented on a GPU.

1. Node-by-Node (NbN)
2. Degrees-of-Freedom -by- Degrees-of-Freedom (DbD)
3. Element-by-Element (EbE)

2.2 NbN strategy

In the NbN strategy, the computation of matrix-vector product is performed by moving through each node of the mesh. Every node has its corresponding rows in the global stiffness matrix. The multiplication of each row is done with the given vector and the result is accumulated into an array. Since the global stiffness matrix is not constructed explicitly, each row corresponding to a node needs to be assembled before multiplication. To generate a row in a global stiffness matrix, contributions from all neighboring elements are needed. In practice, first, the required entries in elemental matrices of neighboring elements are multiplied with corresponding vector entries and then the result is assembled [8,27], which is given as

$$\mathbf{p}^{(n)} = \sum_{e\in\mathcal{E}^{(n)}} (\mathbf{K}_n^e \mathbf{y}^e), \tag{4}$$

where $\mathcal{E}^{(n)}$ is the set of elements connected to node n, $\mathbf{K}_n^e$ is the contribution of stiffness matrix toward node n, $\mathbf{p}$ is the resultant vector and $\mathbf{y}^e$ is the elemental sub-vector of vector $\mathbf{y}$ with which the multiplication has to be done. The NbN strategy for two-dimensional problem having two DOFs per node (N_{dof}) is shown in Algorithm 1. Since the computation is performed by moving through each node, list of elements connected to node n ($\mathcal{E}^{(n)}$) is computed in step 2 in terms of the Node connectivity matrix. For each element in the node connectivity, the elemental connectivity matrix ($\mathbf{E}^{(e)}$) is found along with the local position (q) of the node in the element. The element stiffness matrix $\mathbf{K}^e$ is read on the basis of q. Finally, the required product is calculated in steps 7 and 9 for all DOFs associated with the node.

Algorithm 1 Node-by-Node strategy.

1: **for** Node $n = 1$ to $\mathcal{N}$ **do**
2: Find Node connectivity $\mathcal{E}^{(n)}$
3: **for** element $e \in \mathcal{E}^{(n)}$ **do**
4: $\mathbf{E}^{(e)} \leftarrow$ *ElementConnectivity*() ▷ Extract the element connectivity
5: $\mathbf{y}^{(e)} \leftarrow \mathbf{y}(\mathbf{E}^{(e)})$ ▷ Obtain the sub-vector for multiplication
6: **for** $i = 1$ to DOF-per-element **do**
7: $\mathbf{val}[0] += \mathbf{K}^e[2 * q][i] * \mathbf{y}^e[i]$ ▷ '2' refers to $N_{dof} = 2$
8: $\mathbf{val}[1] += \mathbf{K}^e[2 * q + 1][i] * \mathbf{y}^e[i]$
9: **end for**
10: **end for**
11: **end for**

GPU parallelization is done over the nodes of the mesh. Single thread is assigned to do computation for one node. The node connectivity array and local position array can be reordered to read in a coalesced manner. The elemental stiffness matrices are read from strided locations in the global memory and hence cannot be coalesced. The elemental connectivity matrix is arranged column-wise for minimizing the global memory transactions. The access to vector $\mathbf{y}$ is also not coalesced and it is read through the read-only cache.

It can be observed that each node performs its computation independently and therefore, the problem of data race conditions [12] does not arise. This is a major advantage of the NbN strategy since overhead associated with synchronization mechanism like coloring can be avoided.

For an unstructured mesh, each node can have different number of neighboring elements. This leads to an unequal amount of workload distribution on threads. A GPU warp [12] remains active as long as any of its threads is working. This is not desirable for an efficient utilization of GPU resources that leads to the major disadvantage of the GPU-based NbN strategy.

2.3 DbD strategy

The DbD strategy performs the matrix-vector multiplication by moving through each DOF of the system. Here, computation corresponding to each DOF associated with a node is seen as an independent task. In the DbD strategy, computation of matrix-vector product is implemented as [28]

$$\mathbf{p}^{(u)} = \sum_{e \in \mathcal{E}^{(u)}} \mathbf{K}^e(m^{-1}(u), :)\mathbf{y}^e, \tag{5}$$

where $\mathbf{K}^e$ is the elemental stiffness matrix, m is the local to global mapping and $\mathcal{E}^{(u)}$ is the set of elements connected to DOF u. The DbD strategy is implemented in a way similar to Algorithm 1. Single thread per DOF assignment is used to perform the computation. Input data structure remains identical as Algorithm 1 but now the same data is read by as many threads as the value of N_{dof}. Each thread performs its own computation and accumulates the result into an array in a coalesced manner. This strategy is also known as Row-by-Row solution method [43].

The GPU-based DbD strategy has finer level of granularity than the NbN strategy. However, the input data requirement remains the same as that for the NbN strategy. Moreover, since the same data is required for all the threads associated with a particular node, either it can be read redundantly from the global memory or can be shared among threads using the shared memory. The limited size of the shared memory restricts its use to few cases and generally, data is read redundantly from the global memory. The DbD strategy also suffers from the same load imbalance problem found with the NbN strategy.

2.4 EbE strategy

In the EbE strategy, the computation of matrix-vector product takes place at the elemental level. The obtained result is then assembled to get the final solution. This can be expressed as

$$\mathbf{p} = \underset{e \in \mathcal{E}}{\mathcal{A}} (\mathbf{K}^e \mathbf{y}^e) \tag{6}$$

where $\mathbf{K}^e$ is the elemental stiffness matrix, $\mathbf{y}^e$ is the multiplying vector transformed to the elemental level, $\mathcal{A}$ is the assembly operator, $\mathcal{E}$ is the set of all elements in the mesh and $\mathbf{p}$ is the resultant vector.

Algorithm 2 Element-by-Element strategy.

```
1: for element e = 1 to E do
2:     for i = 1 to DOF-per-element do
3:         for j = 1 to DOF-per-element do
4:             global_id ← D(e, j)
5:             val^e[i] += K^e[i][j] * y[global_id]
6:         end for
7:     end for
8: end for
9: p ← Assembly(val)
```

Algorithm 2 shows the implementation of the EbE strategy. The multiplication of elemental stiffness matrix with vector $\mathbf{y}$ is performed by using the local to global

mapping $\mathcal{D}(e, j)$ in step 4 and the result is stored in vector **val** in step 5. The resultant vector **p** is obtained by assembling **val** in step 9.

In GPU implementation, computation for each element is performed in parallel. There are three prominent ways of distributing workload among threads which are as follows.

1. Single thread per element: In single thread per element approach [21], one thread is responsible for reading the input data, computing elemental matrix-vector product, and accumulating calculated value to the resultant vector. This approach is the simplest to implement. However, each element gets an amount of on-chip memories (shared memory and register) corresponding to a thread only. Therefore, this approach suffers from poor utilization of fast on-chip memories.
2. Single thread per node: The single thread per node approach allocates as many threads to an element as the number of nodes. Each thread performs computation for all DOFs associated with the node.
3. Single thread per DOF: The finest level of granularity is achieved in the single thread per DOF approach. Here, the number of threads equal to DOFs associated with an element is allocated [27,32]. The elemental matrix-vector product is decomposed into several inner-vector products corresponding to each row of the matrix. Each thread is assigned to do computation for one inner-vector product. This approach also provides the highest amount of on-chip memory per element.

After the elemental matrix-vector product is obtained, it needs to be assembled into a final global vector as shown in Algorithm 2. Each non-zero entry in the global vector corresponds to a DOF of the system. Each boundary DOF is shared among multiple elements of the mesh. During parallel assembly of elemental resultant vector, multiple elements tend to put their calculated value to the same location in global vector simultaneously. Such kind of memory operation leads to the problem of data race condition. In the EbE strategy, the problem of race conditions must be addressed by the use of suitable synchronization mechanism like coloring, atomics or using a separate assembly kernel [27].

The EbE strategy, having single thread per DOF allocation, is found to have the best performance [27]. Apart from providing the finest level of parallelism, the strategy has simple data access pattern, balanced load distribution, and can provide better utilization of on-chip memory. Along with the coloring method to handle data race conditions, this strategy is used as a reference in this work.

3 Proposed matrix-free strategy

3.1 Matrix-free solver exploiting symmetry of elemental matrices

The major steps involved in the GPU-based computation of matrix-vector product **Ky** in a matrix-free manner can be seen in Fig. 1. The matrix-vector product can be computed in three steps: (1) transformation of multiplying vector **y** to elemental vector $\mathbf{y}^e$, (2) computation of dense matrix-vector product with elemental matrices, and (3) assembly of computed results. The steps 1 and 3 primarily involve data scatter and

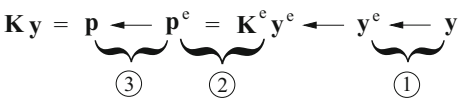


Fig. 1 Major steps involved in the matrix-free computation of elemental matrix-vector product

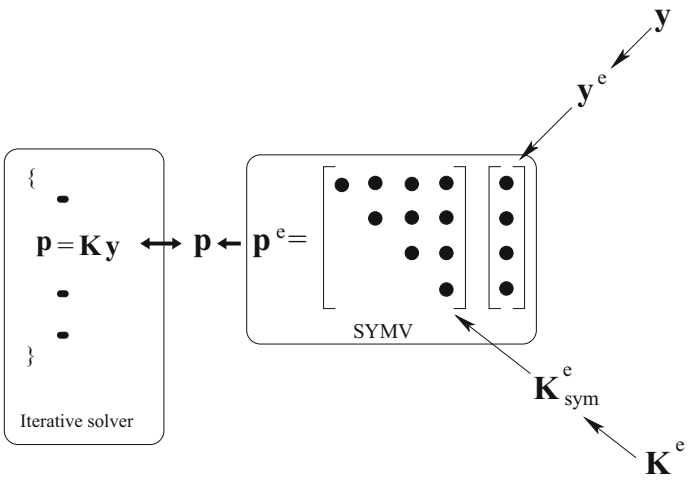


Fig. 2 Illustration of the proposed EbE_{sym} strategy

gather type operation with little arithmetic load. In step 2, the computation of dense matrix-vector is performed for all elements which makes it computationally expensive as compared to the other two steps. The efficient implementation of step 2 is therefore crucial for better performance of the matrix-free solver.

In all existing matrix-free strategies, the computation in step 2 of Fig. 1 is performed by using full elemental stiffness matrices. The proposed strategy makes use of the fact that the elemental matrices in linear FEM is symmetric in most of the cases. This property can be used to perform the elemental dense matrix-vector product using only the lower or upper triangular part of the matrix. The matrix-vector product in the proposed strategy is implemented using only the symmetric part of the elemental matrices as

$$\mathbf{p} = \underset{e \in \mathcal{E}}{\mathcal{A}} (\mathbf{K}^e_{sym} \mathbf{y}^e), \tag{7}$$

where $\mathbf{K}^e_{sym}$ is the symmetric part of elemental stiffness matrix. The proposed matrix-free strategy for FEM is referred to as EbE_{sym} and can be represented in the graphical form as shown in Fig. 2. It can be seen that the dense matrix-vector product required in the matrix-free solver is replaced by a dense symmetric matrix-vector product (SYMV) which requires only the symmetric part of the elemental matrices. In the proposed strategy, all steps mentioned in Fig. 1 is implemented as a single computational kernel with the coloring method to avoid data race conditions.

The computation of a SYMV operation can be performed by following Algorithm 3. The multiplication of each row with vector $\mathbf{y}^e$ is performed by looping over the total

number of rows K_{size}, where the computation is performed first (step 3) for the upper triangular part of the matrix. In step 7 of Algorithm 3, the computation of missing symmetric part (lower triangular part) is performed where indices k and i to the matrix $\mathbf{K}^e$ always refer to the values in the upper triangular part. This shows that the SYMV computation can be implemented by storing only the upper or the lower triangular part of the symmetric matrix.

Algorithm 3 Computation of symmetric matrix-vector product (SYMV)

1: **for** $i = 1$ to K_{size} **do**
2: **for** $j =$ i to K_{size} **do** ▷ Computation for symmetric part
3: $\mathbf{p}^e[i] += \mathbf{K}^e[i][j] * \mathbf{y}^e[j]$
4: **end for**
5: **if** $(i \neq 1)$ **then** ▷ Computation for missing symmetric part
6: **for** $k = (i - 1)$ to 1 **do**
7: $\mathbf{p}^e[i] += \mathbf{K}^e[k][i] * \mathbf{y}^e[k]$
8: **end for**
9: **end if**
10: **end for**

For GPU implementation, the storage of the symmetric part of the matrix can be done in any suitable format which facilitates uniform memory access pattern during computation. However, the same storage format may not be suitable for the computation of missing symmetric part. As shown in step 7 of Algorithm 3, values of matrix $\mathbf{K}^e$ are accessed from the stored symmetric part in a strided and nonuniform manner. Such kind of access pattern wastes the memory bandwidth of GPU and consequently degrades the performance. The optimization of memory access pattern for a SYMV operation on GPU is suggested by many authors [1,11,30]. These studies discuss the strategies to compute the SYMV operation for moderate to large size matrices. However, the approaches available in the literature are either not applicable or not optimal for small size matrices (less than 50) generally found in low-order FEM. Moreover, in the current work, computation has to be performed in a batch for millions of elements. Since matrix-vector product has very low arithmetic intensity, efficient handling of memory overhead becomes indispensable for batch implementation of SYMV. For better performance on GPU, it, thus, becomes extremely important to minimize the data transfer and use coalesced and localized memory access pattern. The proposed EbE_{sym} strategy addresses all these issues by adopting a novel data structure which ensures coalesced memory access while storing only the symmetric part of the elemental matrices. In order to obtain the best performance, the EbE_{sym} strategy seeks to make an efficient use of register cache by using CUDA shuffle instruction. This not only helps in relaxing the shared memory size restrictions but also avoids the data movement to-and-fro from the shared memory. Single thread per node assignment similar to EbE strategy (Sect. 2.4) is used to achieve balanced workload distribution. Also, each thread performs its task independently so that no synchronization barrier is required. In order to demonstrate the performance of the proposed strategy, quadrilateral element with linear basis function is considered.

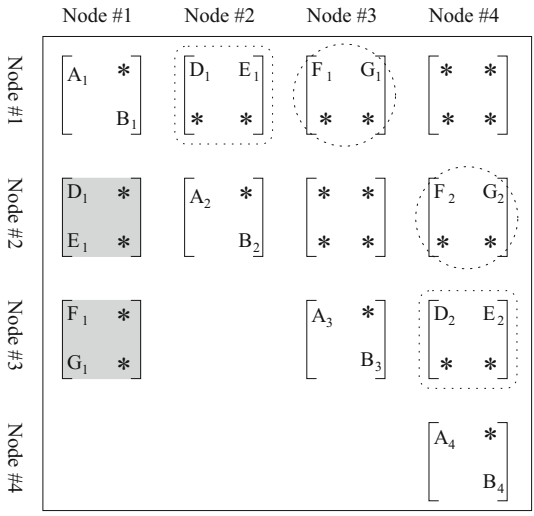


Fig. 3 Organization of the elemental stiffness matrix for a 4-noded quadrilateral element with two DOFs per node

3.2 Kernel design and data structure for *EbE*$_{sym}$ strategy

In the proposed strategy, the elemental stiffness matrix is divided into a number of sub-matrices as shown in Fig. 3. The figure shows nonzero entries in the upper triangular part of the elemental stiffness matrix for 4-noded quadrilateral element with $N_{dof} = 2$. Each node is associated with as many rows and columns in the matrix as the value of N_{dof}. The size of sub-matrices is kept equal to N_{dof} and it contains the values corresponding to one node in row and one node in column. For example, the following sub-matrix contains all the computed values between node 1 in row and node 2 in column.

$$\begin{bmatrix} D_1 & E_1 \\ * & * \end{bmatrix}$$

Depending on the position in the matrix the sub-matrices are categorized into two groups: diagonal and off-diagonal. The diagonal group contains all sub-matrices lying on the diagonal of the elemental matrix, such as

$$\begin{bmatrix} A_1 & * \\ * & B_1 \end{bmatrix}, \begin{bmatrix} A_2 & * \\ * & B_2 \end{bmatrix}, \text{etc.}$$

The off-diagonal group contains sub-matrices that are not unique. These sub-matrices appear in the symmetric part also. In case of the diagonal group, all sub-matrices are associated with only one node number (the rows and column nodes are same). In case of the off-diagonal group, two such numbers exist, that is, one associated with the row and other with the column. Therefore, each sub-matrix is identified with a nodal index of the form $K^e\{n, m\}$ which represents sub-matrix at n^{th} node in row and m^{th}

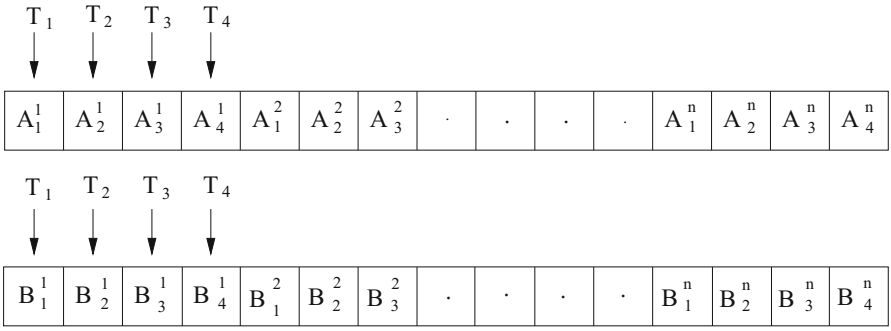


Fig. 4 Data access pattern for diagonal group

node in column. The row node number is used to find global DOF to store result of multiplication, whereas the column node number is used to find global column indices of vector **y** to perform multiplication.

The computation of symmetric matrix-vector product is divided into two stages. In the first stage, the computation for the diagonal group is performed whereas in the second stage, multiplication for the off-diagonal group is done.

The sub-matrices in the diagonal group are unique. Since computation is performed in a node-wise manner, these sub-matrices contribute to the matrix-vector multiplication results for their respective node number. Thread assigned to each node reads all the unique entries from its sub-matrix and performs multiplication with **y** vector. The data access pattern for the diagonal group is shown in Fig. 4. It shows four threads accessing the values marked as A and B (also shown in Fig. 3) from four sub-matrices of the diagonal group. Here, in Fig. 4, superscript represents the element number and the subscript denotes the sub-matrix position in the diagonal group. The other entries of a sub-matrix are accessed in a similar way. In order to achieve coalesced access for a warp, data for other elements are stored side by side. Once these values are read, they get multiplied by the given vector and stored in the shared memory. Here, each thread uses its global node number to read values from vector **y** through the read-only cache. The read from **y** vector is not coalesced.

The off-diagonal entries in the symmetric matrix are not unique. The transpose of sub-matrices in the off-diagonal group can be obtained if the row and column nodes are interchanged, as shown in Fig. 3. It can be seen that the sub-matrix located at $K^e\{1, 2\}$ appears in its transposed form at $K^e\{2, 1\}$. This implies that the same sub-matrix can be used to perform the computation for both node 1 and node 2. Similarly, the sub-matrix at position $K^e\{1, 3\}$ can be used for both node 1 and node 3. Thus, for a 4-noded quadrilateral element, the computation for two sub-matrices can be performed simultaneously. The computation for the off-diagonal group is implemented such that each thread is assigned with an equal workload. Therefore, the sub-matrices for simultaneous computation must be chosen judiciously. As shown in Fig. 3, the sub-matrices having the same type of enclosing can be processed at the same time. If chosen otherwise, any one thread can remain idle and others may have to do their task.

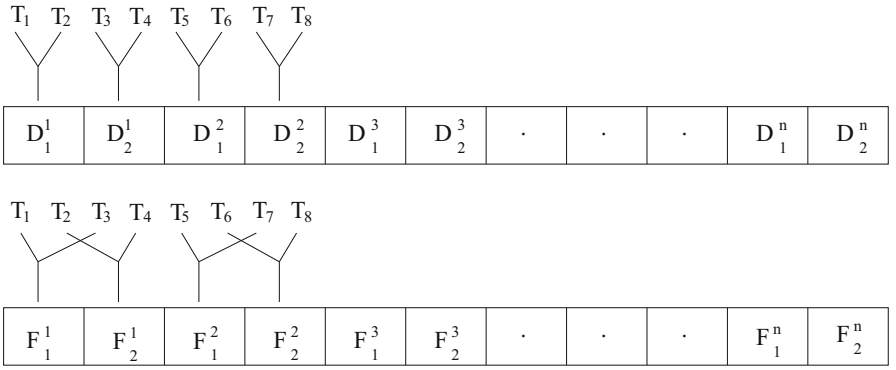


Fig. 5 Data access pattern for off-diagonal group

Since the computation of matrix-vector product uses only the symmetric part of the matrix, the sub-matrices in the missing part (lower triangular part in Fig. 3) must be obtained separately or shared between two threads. Due to the limited size of the shared memory, values in sub-matrices are read redundantly from the global memory. However, data is arranged such that it results into a broadcast. The broadcast from the global memory is although slower than the shared memory, it has lower overhead than reading values separately. The data access pattern for the off-diagonal entries is shown in Fig. 5. Here, D denotes the corresponding value in sub-matrices with same type of enclosing (refer to Fig. 3) in which subscript indicates the sub-matrix within an element and superscript indicates the element number. It can be seen that threads T_1 and T_2, assigned to node 1 and 2, read the same D_1^1 value whereas threads T_3 and T_4 read D_2^1 value which is required to perform computation of node 3 and node 4. The F values are stored and read in a similar way, except the values are now required by different set of threads. The data for all elements are kept beside each other to enable coalesced access for a warp.

Once the data is read, it is multiplied with the corresponding values of vector $\mathbf{y}^e$. The $\mathbf{y}^e$ vector is extracted from $\mathbf{y}$ vector by using column indices of stiffness matrix entries. In FEM, column indices of an elemental stiffness matrix can be obtained by the global node numbers and N_{dof}. In particular, the column node numbers of each sub-matrix can be used to obtain column indices of its entries. It can be observed from Fig. 3 that threads working over a sub-matrix either contain row node number or column node number of the sub-matrix. The row node number becomes column node number for a sub-matrix after it gets transposed. Thus, the global node number of two threads can be interchanged to get column node number of the sub-matrix. This is achieved in the proposed strategy by using the warp-shuffle instruction. Using the warp shuffle feature, the proposed strategy prevents the use of the shared memory as well as the global memory access. The warp shuffle feature is found to be more faster than the shared memory and leads to better utilization of register cache [20].

4 Results and discussion

The efficiency and performance of the proposed EbE_{sym} strategy are evaluated by solving elasticity and steady-state heat conduction equations in two dimensions (2D). The elasticity equation is solved over cantilever and L-shaped beam, and the steady-state heat equation is solved over a plate with multiple holes. Further, the performance of the proposed strategy is compared with the existing matrix-free strategies discussed in Sect. 2. In all the numerical problems, a symmetric positive-definite system of equations is obtained due to finite element discretization. Since the conjugate gradient (CG) solver is the most efficient and widely used iterative solver for the symmetric positive-definite system [36,40], it is chosen as a solver in this work. It is important to note that the proposed strategy is equally applicable to other iterative solvers including multigrid [17]. Coloring method is used to handle data race conditions with the EbE and EbE_{sym} strategies. However, any data race condition is not observed with the NbN and DbD strategies.

The geometry of cantilever beam and L-shaped beam problems is relatively simple, and hence, structured mesh is generated through an FEM software package called as ABAQUS. However, the data structure generated by ABAQUS is unstructured and the same is used in this work. All the strategies have been implemented by considering the mesh as unstructured and do not use any simplification of the mesh to alter the performance. The plate with multiple holes problem is solved with unstructured mesh which is also generated through ABAQUS.

The hardware used consists of NVIDIA Tesla K40 GPU and Intel Xeon (R) E5-2650 CPU. The CPU consists of 12 physical cores clocked at 2.2 GHz and the GPU consists of 2880 cores clocked at 745 MHz. The CUDA runtime version 9.2 is used. All the numerical results are obtained using the double precision floating point arithmetic.

4.1 Elasticity problem

The following elasticity equation is considered over the domain Ω,

$$\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{b} = 0, \quad \forall\, \boldsymbol{x} \in \Omega, \tag{8}$$

which is subjected to the following boundary conditions,

$$\begin{aligned} \boldsymbol{u}(\boldsymbol{x}) &= \boldsymbol{u}_0, \quad \boldsymbol{x} \in \Gamma_u, \\ \boldsymbol{t}(\boldsymbol{x}) &= \bar{\boldsymbol{t}}, \quad \boldsymbol{x} \in \Gamma_t, \end{aligned}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor, $\boldsymbol{b}$ is the body force per unit volume, $\boldsymbol{u}$ is the unknown displacement variable, $\boldsymbol{u}_0$ is the specified displacement on the boundary Γ_u and $\bar{\boldsymbol{t}}$ is the given traction on the boundary Γ_t. The elasticity equation is solved for cantilever beam and L-shaped beam under plane stress condition and linear strain-displacement relation. The problem geometry along with the dimensions and boundary conditions are shown in Figs. 6 and 7 , respectively. The material properties are taken as: Young's modulus (E) = 210 GPa and Poisson's ratio (ν) = 0.3. The domain is discretized

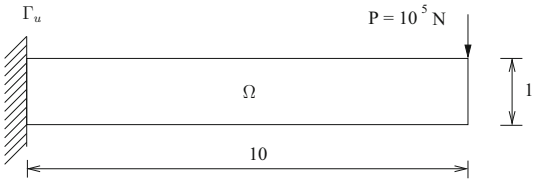


Fig. 6 A 2D cantilever beam with end load. All dimensions are in meters (m)

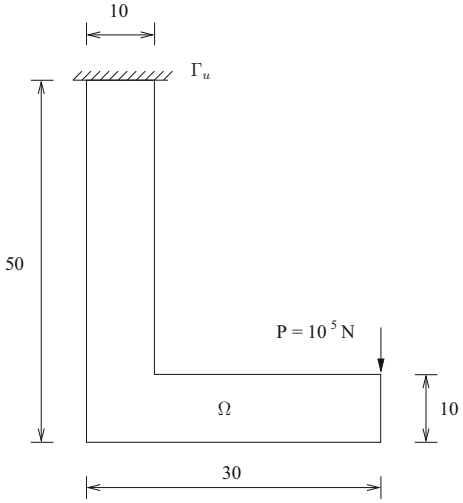


Fig. 7 L-shaped beam. All dimensions are in meters (m)

Table 1 Finite element mesh for 2D cantilever beam

Mesh	Elements	Nodes	Degrees of freedom
C1	100,000	101,101	202,202
C2	400,000	402,201	804,402
C3	900,000	903,301	1,806,602
C4	1,600,000	1,604,401	3,208,802
C5	2,500,000	2,505,551	5,011,002

Table 2 Finite element mesh for L-shaped beam

Mesh	Elements	Nodes	Degrees of freedom
L1	1,750,000	1,754,001	3,508,002
L2	3,112,889	3,118,224	6,236,448
L3	4,480,000	4,486,401	8,972,802
L4	5,783,967	5,791,240	11,582,480
L5	7,000,000	7,008,001	14,016,002

using 4-noded quadrilateral elements having two DOFs per node. The problems are solved for different level of mesh refinement to evaluate the performance at various workload. Tables 1 and 2 present the mesh with different number of elements and corresponding DOFs for 2D cantilever beam and L-shaped beam, respectively.

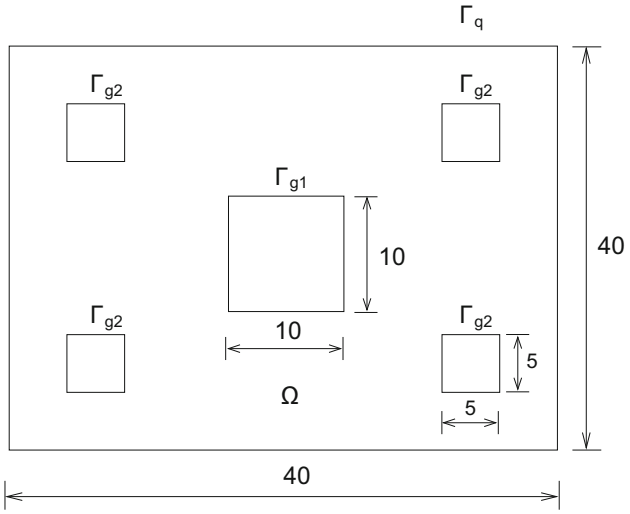


Fig. 8 A plate with multiple holes. All dimensions are in meters (m)

Table 3 Finite element mesh for steady-state heat conduction problem

Mesh	Elements	Degrees of freedom
H1	9,84,681	9,88,734
H2	1,938,537	1,944,226
H3	3,048,540	3,055,672
H4	4,215,044	4,223,429
H5	6,240,237	6,250,435

4.2 Steady-state heat conduction problem

The following steady-state heat conduction equation is solved over a plate with multiple holes as shown in Fig. 8.

$$\begin{aligned} \nabla \cdot (\boldsymbol{\kappa} \cdot \nabla T(\boldsymbol{x})) &= f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega, \\ T(\boldsymbol{x}) &= g(\boldsymbol{x}), \quad \boldsymbol{x} \in \Gamma_g, \\ \boldsymbol{n}(\boldsymbol{x}) \cdot \boldsymbol{\kappa} \cdot \nabla T(\boldsymbol{x}) &= 0, \quad \boldsymbol{x} \in \Gamma_q, \\ \Gamma_g &= \Gamma_{g1} \cup \Gamma_{g2}, \end{aligned} \tag{9}$$

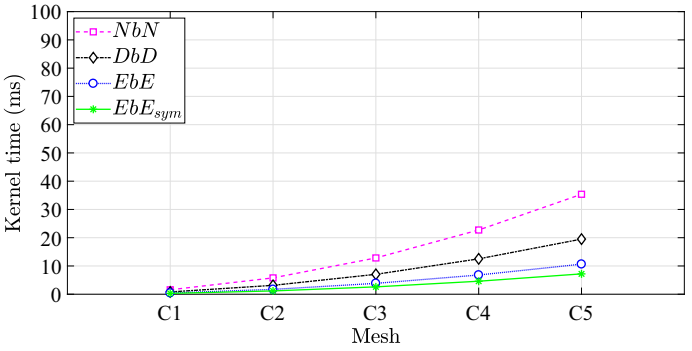
Here, $T(\boldsymbol{x})$ is the unknown temperature field, $f(\boldsymbol{x}) = 0$, $g(\boldsymbol{x}) = 200$ on Γ_{g2} and 10 on Γ_{g1} and $\boldsymbol{\kappa}$ is the thermal conductivity matrix which is taken as identity. Table 3 lists the mesh with various level of refinement used in this analysis. The domain is discretized with 4-noded quadrilateral element with single DOF per node.

4.3 Performance results

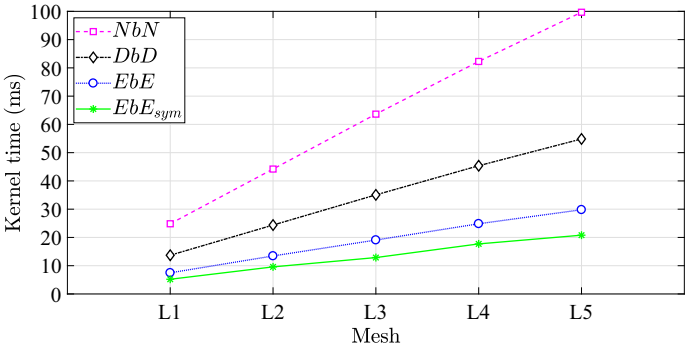
The performance of the EbE_{sym} strategy is assessed on the basis of the kernel time, arithmetic throughput (GFLOP/s) and memory bandwidth. The kernel time is evaluated using the CUDA event function, whereas the GFLOP/s and bandwidth are calculated by the metrics given by `nvprof` profiler. Here, the kernel time is referred to as the execution time of CUDA kernel in one iteration of the CG solver. For the NbN and DbD strategies only one kernel is launched per iteration but for the EbE and EbE_{sym} strategies separate kernel is launched for each color. Therefore, the kernel time in case of the EbE and EbE_{sym} strategies includes execution time for all the colors. It is noted that the NbN and DbD strategies are equivalent in case of steady-state heat transfer problem since each node has only one DOF. Hence, results for the DbD strategy are not presented for the heat transfer problem.

Figure 9 shows the comparison of kernel time for different matrix-free strategies for the elasticity and heat transfer problems. The NbN strategy takes the highest amount of kernel time in all the test problems. It also has the highest amount of data requirement as compared to all the other strategies. With the same data structure the DbD strategy achieves better timings than the NbN strategy by just increasing the granularity of computation. The redundant access of data in case of the DbD strategy does not seem to have much overhead as the values are broadcasted to N_{dof} threads from the global memory. Also, access to the elemental matrix requires lesser number of transactions as compared to the NbN strategy as more number of threads now accesses the same matrix. However, in both the NbN and DbD strategies, gather operation is performed to read the elemental matrices in an uncoalesced manner. The elemental matrices constitute the largest amount of data that a matrix-free solver needs to access. As evident from less kernel time of the EbE strategy (Fig. 9) compared to the NbN and DbD strategies, the uncoalesced access to the elemental matrices has a large impact on the performance. In the EbE strategy, the elemental matrices are accessed in a coalesced manner. Apart from better memory access pattern, the EbE strategy has the finest level of granularity, less data requirement, and equally distributed workload on each of the computational threads. With all these characteristics, the EbE strategy overcome the overhead associated with race conditions handling and achieves the least kernel time among the existing matrix-free strategies.

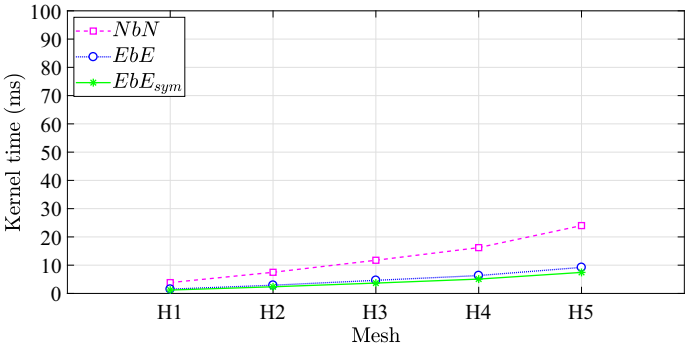
The proposed EbE_{sym} strategy outperforms all the other strategies in every test problem (refer to Fig. 9). Since the EbE_{sym} strategy inherits the major characteristic of the EbE strategy, a better performance compared to the NbN and DbD strategies is expected. However, the superior performance compared to the EbE strategy can be mainly attributed to the reduction in data movement due to use of only the symmetric part of the elemental matrices. In elasticity problem, 4-noded quadrilateral elements with two DOFs per node is used which gives the elemental matrix of size 8×8. The implementation and optimization of matrix-vector product for such a smaller size matrix in a batch mode is extremely challenging as it involves very low arithmetic load compared to the required amount of data movement. The proposed EbE_{sym} strategy achieves better kernel time compared to the EbE strategy due to a unique data structure that ensures localized and coalesced access pattern using only the symmetric part of the



(a) Kernel time for the 2D cantilever beam problem.



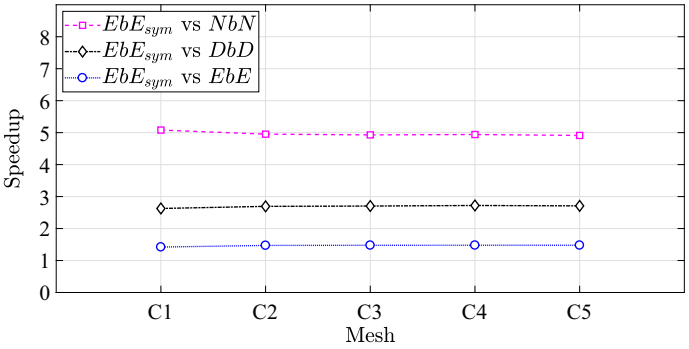
(b) Kernel time for the L-shaped beam problem.



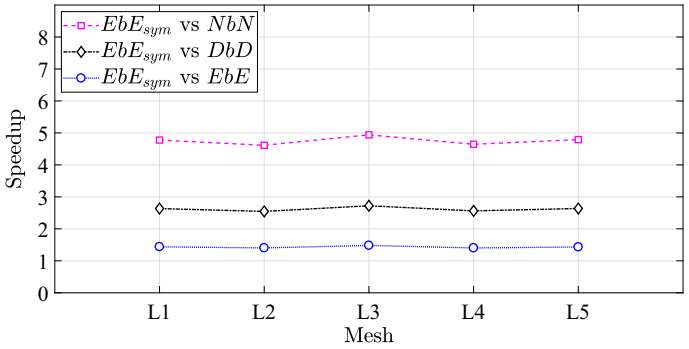
(c) Kernel time for the steady-state heat conduction problem over a plate.

Fig. 9 Comparison of kernel time for test problems

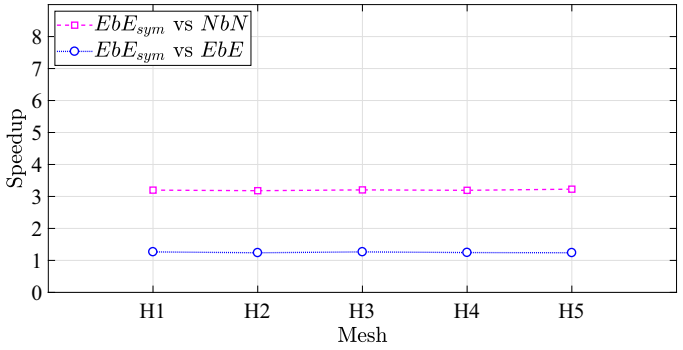
elemental matrices. The reduction in data requirement helps in maintaining a higher computation to data movement ratio than the EbE strategy which is favorable for GPU implementation. Moreover, the EbE_{sym} strategy also achieves better execution times for the heat transfer problem (Fig. 9c) which uses the elemental matrices of size 4×4.



(a) Speedup for the 2D cantilever beam problem.



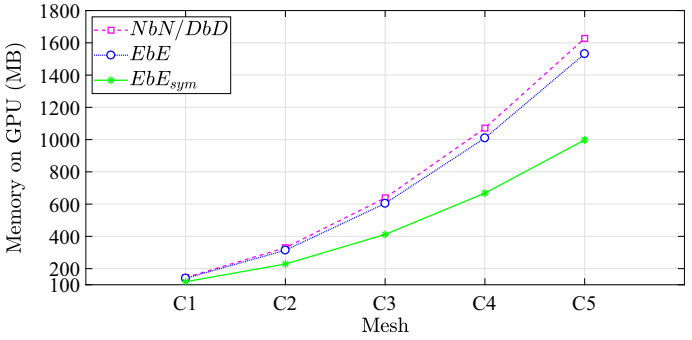
(b) Speedup for the L-shaped beam problem.



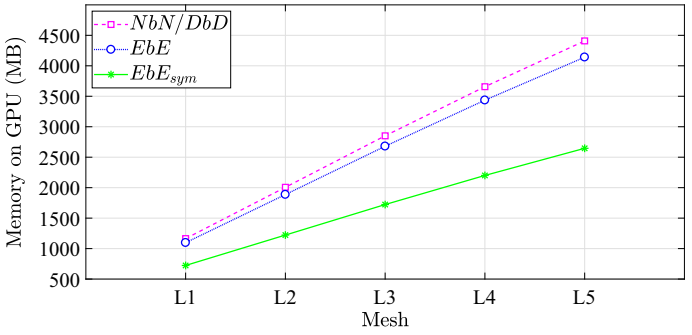
(c) Speedup for the steady-state heat conduction problem.

Fig. 10 Speedup achieved by EbE_{sym} strategy over the other matrix-free strategies

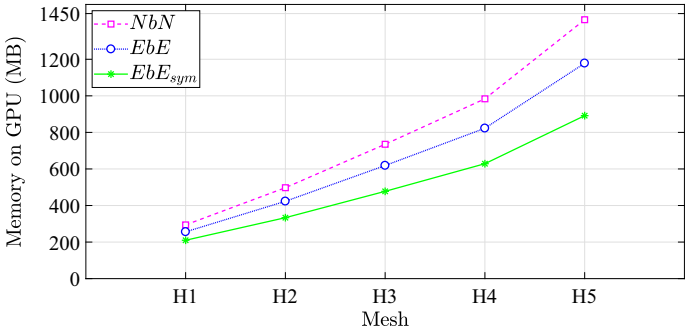
The speedup obtained by the EbE_{sym} strategy over the other strategies for matrix-free solver is shown in Fig. 10. In all the three problems, a consistent speedup is observed which suggests that the EbE_{sym} strategy is able to scale well with increasing problem size. With respect to the NbN strategy, approximately $5\times$ speedup is observed for both cantilever and L-shaped beam problems. In the case of the DbD strategy, approximately $2.8\times$ speedup is observed for elasticity problems. Relatively



(a) GPU memory utilization for the 2D cantilever beam problem.



(b) GPU memory utilization for the L-shaped beam problem.



(c) GPU memory utilization for the steady-state heat conduction problem over a plate.

Fig. 11 GPU memory utilization by various strategies

lower speedup is observed for the case of heat transfer problem because smaller elemental matrices have less memory overhead in the case of the NbN or DbD strategy. The proposed strategy could achieve 1.4× speedup over the EbE strategy for elasticity problems and 1.3× speedup in the case of the heat transfer problem. Here, a similar speedup in both the cases illustrates the suitability of the proposed strategy for extremely small size elemental matrices.

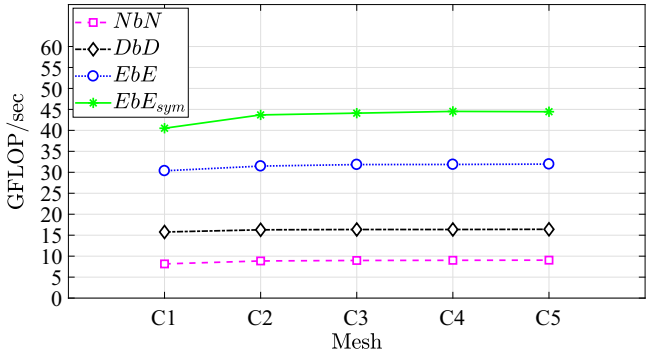
Figure 11 shows the amount of GPU memory occupied by different strategies as a function of the problem size. It can be observed that the proposed EbE_{sym} matrix-free strategy consumes the least amount of GPU memory. This suggests that a much larger problem can be solved by the proposed strategy on a given GPU card in lesser amount of time. The NbN and DbD strategies occupy the highest amount of memory in all the numerical problems. This is due to the dependency of these strategies on arrays like node connectivity and local position of nodes in each element in addition to the elemental connectivity and the elemental matrices. The EbE strategy only stores the elemental connectivity and the elemental matrices on GPU and therefore requires lesser memory than the NbN and DbD strategies. The least amount of memory consumption by the EbE_{sym} strategy is due to the storage of only the symmetric part of elemental matrices. For elemental matrix of size 8×8 and 4×4 the EbE_{sym} strategy uses only 36 and 10 number of entries respectively to perform computation. This leads to $1.7\times$ and $1.4\times$ reduction in data at the elemental level. Overall, the EbE_{sym} strategy requires $1.5\times$ less memory in elasticity problem and $1.3\times$ less memory in heat transfer problem (refer Fig. 11) than the EbE strategy. The ratio of total number of entries to unique entries in a symmetric matrix is given by

$$\frac{2n}{n+1}, \tag{10}$$

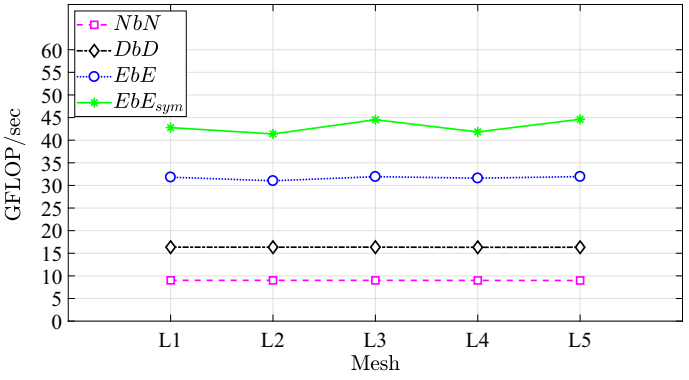
where n is the size of matrix. As the size of matrix increases, this ratio tends to move closer to two. Thus, for higher order finite elements where the elemental matrix size is large use of the symmetric part of the matrix can save up to 50% of the data required in the EbE strategy. Furthermore, the performance of the EbE_{sym} strategy relative to the EbE strategy can also become better for larger matrix size.

A deeper insight into the performance of various matrix-free strategies can be obtained by looking into arithmetic throughput and effective memory bandwidth achieved by them. Figure 12 shows the GFLOP/s achieved by the different matrix-free strategies as a function of mesh size. The arithmetic throughput of the proposed EbE_{sym} strategy having the least kernel time is found to be the highest in all the test problems. The EbE strategy shows significant improvement in GFLOP/s as compared to the NbN and DbD strategies. Compared to the EbE strategy, $1.5\times$ and $1.3\times$ better throughput are observed by the EbE_{sym} strategy in case of elasticity and heat transfer problems. However, GFLOP/s achieved by all the strategies is found to be much lower than the peak value of the device. The maximum value achieved by the EbE_{sym} strategy reaches approximately 3% of the peak value of NVIDIA Tesla K40.

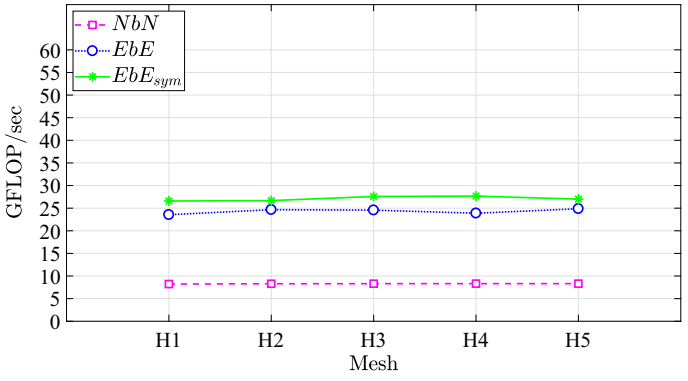
The comparison of effective memory bandwidth achieved by various strategies is shown in Fig. 13. The memory bandwidth by all the strategies is found to be on the higher side indicating memory bound nature of the matrix-free solvers. The NbN and DbD strategies are found to have relatively lesser bandwidth than the EbE strategy. The EbE strategy achieves 212 GB/s for the case of elasticity problems and 208 GB/s in heat transfer problem. The proposed EbE_{sym} strategy shows small improvement over the EbE strategy and achieves bandwidth of 215 GB/s and 209 GB/s in elasticity and heat transfer problems, respectively. The maximum bandwidth achieved by the EbE_{sym} strategy is found to be approximately 74% of the theoretical peak value of



(a) GFLOP/s for the 2D cantilever beam problem.

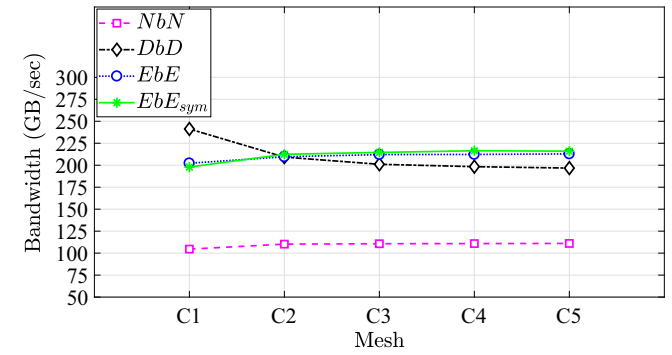


(b) GFLOP/s for the L-shaped beam problem.

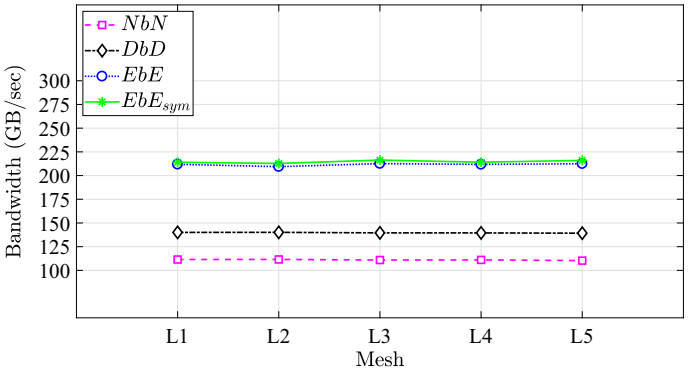


(c) GFLOP/s for the steady-state heat conduction problem over a plate.

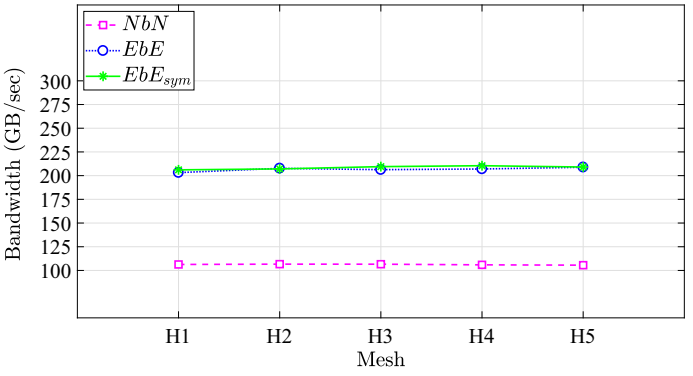
Fig. 12 GFLOP/s achieved by various matrix-free strategies



(a) Memory bandwidth for the 2D cantilever beam problem.



(b) Memory bandwidth for the L-shaped beam problem.



(c) Memory bandwidth for the heat conduction problem.

Fig. 13 Memory bandwidth achieved by various matrix-free strategies

NVIDIA Tesla K40 GPU which is close to the achievable bandwidth of the device [13]. Since the maximum arithmetic throughput is found to be close to 3% of theoretical peak throughput, it can be concluded that the performance of the proposed strategy is limited by memory bandwidth.

5 Conclusions

A new GPU-based matrix-free strategy (EbE_{sym}) for FEM has been proposed. The developed strategy was based on an element-by-element FE solver which replaced the SpMV operation in an iterative solution method by an element level dense matrix-vector product. A new approach to compute the elemental matrix-vector product was developed which used only the symmetric part of the elemental matrices. The performance of the proposed solver was evaluated by solving the elasticity and the heat transfer problems on unstructured mesh using 4-noded quadrilateral element and the comparison was made with the existing GPU-based matrix-free solvers. For the elasticity problems (two DOFs per node), approximately 5× speedup was observed over the node based (NbN), 2.8× over the DOF based (DbD) and 1.4× over the element based (EbE) matrix-free strategies. In heat conduction problem (single DOF per node), 3× speedup over the NbN strategy and 1.3× speedup over the EbE strategy were obtained. As a consequence of using the symmetric part of the elemental matrices the overall memory footprint of the proposed EbE_{sym} strategy was reduced by 1.5× for the elasticity and 1.3× for the heat conduction problems over the state-of-the-art EbE strategy. The obtained results suggest that the proposed strategy can be used to solve problems of bigger sizes on a given GPU card in lesser time. Also, the proposed strategy is applicable where the symmetric elemental matrices are generated by FEM. In future, the proposed strategy can be applied to various element types including higher order and three-dimensional elements to study its performance and identify the limitations, if any. The outcome of the future work is expected to make this strategy more generic and applicable to a broader class of problems in FEM. Moreover, the proposed strategy can also be used to develop kernel for batched symmetric matrix-vector product using small size matrices for linear algebra applications.

Acknowledgements The authors are grateful to the SERB, DST for supporting this research under Project SR/FTP/ETA-0008/2014.

References

1. Abdelfattah A, Dongarra J, Keyes D, Ltaief H (2012) Optimizing memory-bound SYMV kernel on GPU hardware accelerators. In: International conference on high performance computing for computational science. Springer, pp 72–79
2. Ahamed AKC, Magoulès F (2017) Conjugate gradient method with graphics processing unit acceleration: CUDA vs OpenCL. Adv Eng Softw 111:32–42. https://doi.org/10.1016/j.advengsoft.2016.10.002
3. Alexandersen J, Sigmund O, Aage N (2016) Large scale three-dimensional topology optimisation of heat sinks cooled by natural convection. Int J Heat Mass Transf 100:876–891. https://doi.org/10.1016/j.ijheatmasstransfer.2016.05.013
4. Altinkaynak A (2017) An efficient sparse matrix-vector multiplication on CUDA-enabled graphic processing units for finite element method simulations. Int J Numer Methods Eng 110(1):57–78. https://doi.org/10.1002/nme.5346
5. Anzt H, Gates M, Dongarra J, Kreutzer M, Wellein G, Köhler M (2017) Preconditioned Krylov solvers on GPUs. Parallel Comput 68:32–44
6. Bauer S, Drzisga D, Mohr M, Rüde U, Waluga C, Wohlmuth B (2018) A stencil scaling approach for accelerating matrix-free finite element implementations. SIAM J Sci Comput 40(6):C748–C778. https://doi.org/10.1137/17M1148384

7. Bell N, Garland M (2009) Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the conference on high performance computing networking, storage and analysis, ACM, p 18
8. Cai Y, Li G, Wang H (2013) A parallel node-based solution scheme for implicit finite element method using GPU. Proc Eng 61:318–324. https://doi.org/10.1016/j.proeng.2013.08.022
9. Carey GF, Jiang BN (1986) Element-by-element linear and nonlinear solution schemes. Int J Numer Methods Biomed Eng 2(2):145–153
10. Cecka C, Lew AJ, Darve E (2011) Assembly of finite element methods on graphics processors. Int J Numer Methods Eng 85(5):640–669
11. Charara A, Keyes D, Ltaief H (2019) Batched triangular dense linear algebra kernels for very small matrix sizes on GPUs. ACM Trans Math Softw TOMS 45(2):15:1–15:28. https://doi.org/10.1145/3267101
12. Corporation NVIDIA (2019) CUDA C programming guide. Version 10
13. Deakin T, McIntosh-Smith S (2015) GPU-STREAM: benchmarking the achievable memory bandwidth of graphics processing units. In: SuperComputing, IEEE/ACM, Austin, USA
14. Fehn N, Wall WA, Kronbichler M (2019) A matrix-free high-order discontinuous Galerkin compressible Navier–Stokes solver: a performance comparison of compressible and incompressible formulations for turbulent incompressible flows. Int J Numer Methods Fluids 89(3):71–102. https://doi.org/10.1002/fld.4683
15. Filippone S, Cardellini V, Barbieri D, Fanfarillo A (2017) Sparse matrix-vector multiplication on GPGPUs. ACM Trans Math Softw TOMS 43(4):30
16. Fu Z, Lewis TJ, Kirby RM, Whitaker RT (2014) Architecting the finite element method pipeline for the GPU. J Comput Appl Math 257:195–211. https://doi.org/10.1016/j.cam.2013.09.001
17. Göddeke D (2011) Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters. Logos Verlag Berlin GmbH
18. Hughes TJR, Levit I, Winget J (1983) An element-by-element solution algorithm for problems of structural and solid mechanics. Comput Methods Appl Mech Eng 36(2):241–254. https://doi.org/10.1016/0045-7825(83)90115-9
19. Joldes GR, Wittek A, Miller K (2010) Real-time nonlinear finite element computations on GPU-application to neurosurgical simulation. Comput Methods Appl Mech Eng 199(49–52):3305–3314
20. Kiran U, Sharma D, Gautam SS (2019) GPU-warp based finite element matrices generation and assembly using coloring method. J Comput Des Eng 6(4):705–718. https://doi.org/10.1016/j.jcde.2018.11.001
21. Kiss I, Gyimothy S, Badics Z, Pavo J (2012) Parallel realization of the element-by-element FEM technique by CUDA. Magn IEEE Trans 48(2):507–510
22. Komatitsch D, Michéa D, Erlebacher G (2009) Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. J Parallel Distrib Comput 69(5):451–460
23. Kronbichler M, Kormann K (2019) Fast matrix-free evaluation of discontinuous Galerkin finite element operators. ACM Trans Math Softw. https://doi.org/10.1145/3325864
24. Li R, Saad Y (2013) GPU-accelerated preconditioned iterative linear solvers. J Supercomput 63(2):443–466
25. Macioł P, Płaszewski P, Banaś K (2010) 3D finite element numerical integration on GPUs. Proc Comput Sci 1(1):1093–1100
26. Markall G, Slemmer A, Ham D, Kelly P, Cantwell C, Sherwin S (2013) Finite element assembly strategies on multi-core and many-core architectures. Int J Numer Methods Fluids 71(1):80–97
27. Martínez-Frutos J, Martínez-Castejón PJ, Herrero-Pérez D (2015) Fine-grained GPU implementation of assembly-free iterative solver for finite element problems. Comput Struct 157:9–18
28. Martínez-Frutos J, Herrero-Pérez D (2015) Efficient matrix-free GPU implementation of fixed grid finite element analysis. Finite Elem Anal Des 104:61–71. https://doi.org/10.1016/j.finel.2015.06.005
29. Müller E, Guo X, Scheichl R, Shi S (2013) Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs. Comput Vis Sci 16(2):41–58. https://doi.org/10.1007/s00791-014-0223-x
30. Nath R, Tomov S, Dong TT, Dongarra J (2011) Optimizing symmetric dense matrix-vector multiplication on GPUs. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis. ACM, New York, NY, USA, SC '11, pp 6:1–6:10. https://doi.org/10.1145/2063384.2063392

31. Ohshima S, Hayashi M, Katagiri T, Nakajima K (2013) Implementation and evaluation of 3D finite element method application for CUDA. In: Daydé M, Marques O, Nakajima K (eds) High performance computing for computational science—VECPAR 2012. Springer, Berlin, Heidelberg, pp 140–148
32. Pikle NK, Sathe SR, Vyavahare AY (2018) High performance iterative elemental product strategy in assembly-free FEM on GPU with improved occupancy. Computing 100(12):1273–1297. https://doi.org/10.1007/s00607-018-0613-x
33. Ram L, Sharma D (2017) Evolutionary and GPU computing for topology optimization of structures. Swarm Evolut Comput 35:1–13
34. Reguly I, Giles M (2013) Finite element algorithms and data structures on graphical processing units. Int J Parallel Progr 43(2):203–239
35. Rupp K, Weinbub J, Jüngel A, Grasser T (2016) Pipelined iterative solvers with kernel fusion for graphics processing units. ACM Trans Math Softw TOMS 43(2):11:1–11:27. https://doi.org/10.1145/2907944
36. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia. https://doi.org/10.1137/1.9780898718003
37. Sanfui S, Sharma D (2017) A two-kernel based strategy for performing assembly in FEA on the graphics processing unit. In: 2017 international conference on advances in mechanical, industrial, automation and management systems (AMIAMS), IEEE, pp 1–9
38. Sanfui S, Sharma D (2019) Exploiting symmetry in elemental computation and assembly stage of GPU-accelerated FEA. In: Proceedings at the 10th international conference on computational methods (ICCM2019). ScienTech Publisher, pp 641–651
39. Sanfui S, Sharma D (2020) A three-stage gpu-based fea matrix generation strategy for unstructured meshes. International Journal of Numerical Methods in Engineering. (**in press**). https://doi.org/10.1002/nme.6383
40. Shewchuk JR (1994) An introduction to the conjugate gradient method without the agonizing pain. Tech. Rep, Pittsburgh
41. Tezduyar T, Aliabadi S, Behr M, Mittal S (1994) Massively parallel finite element simulation of compressible and incompressible flows. Comput Methods Appl Mech Eng 119(1):157–177. https://doi.org/10.1016/0045-7825(94)00082-4
42. Top500 Supercomputers (2019). https://www.top500.org. Accessed 2 Jan 2020
43. van Rietbergen B, Weinans H, Huiskes R, Polman B (1996) Computational strategies for iterative solutions of large FEM applications employing voxel data. Int J Numer Methods Eng 39(16):2743–2767
44. Wong J, Kuhl E, Darve E (2015) A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems. Int J Numer Methods Eng 102(12):1784–1814. https://doi.org/10.1002/nme.4865
45. Yagawa G, Soneda N, Yoshimura S (1991) A large scale finite element analysis using domain decomposition method on a parallel computer. Comput Struct 38(5):615–625. https://doi.org/10.1016/0045-7949(91)90013-C
46. Zhang J, Shen D (2013) GPU-based implementation of finite element method for elasticity using CUDA. In: 2013 IEEE 10th international conference on high performance computing and communications, 2013 IEEE international conference on embedded and ubiquitous computing, pp 1003–1008. https://doi.org/10.1109/HPCC.and.EUC.2013.142