

Accelerating Finite Element Assembly on a GPU



Utpal Kiran, Sachin S. Gautam, and Deepak Sharma

Abstract In this paper, a parallel strategy for assembly of finite element matrices on graphics processing unit (GPU) is presented. Considering the limited memory size of a GPU, the proposed strategy doesn't store the elemental matrices into memory but performs on-the-fly computation and stores the data directly into a global stiffness matrix, reducing memory requirement and preventing overhead due to a separate assembly step. The global stiffness matrix is stored in compressed sparse row (CSR) storage format, commonly used by GPU-accelerated linear solver libraries. However, the assembly of elemental matrices directly into a sparse storage format requires prior knowledge of locations of nonzeros. The current work presents an efficient strategy to pre-compute indices for assembly into CSR sparse storage format. The proposed strategy has been implemented on both CPU and GPU. The performance characteristic of the proposed finite element solver is measured by solving large-scale three-dimensional (3D) elasticity problem involving a maximum of 4.7 million degrees of freedom (DOFs). A comparison is made with the standard assembly implementation in Eigen C++ library, which first stores the nonzero values in the form of triplets and then assembles into CSR sparse format. For the finest mesh with 4.7 million DOFs, the proposed CPU-based assembly strategy achieves $9.3\times$ speedup over Eigen library. The computation of indices for assembly into CSR format takes 15.7 s on CPU and 2.4 s on GPU for 4.7 million DOFs. The computation of elemental matrices and their assembly, implemented on GPU as a single compute kernel, is found to be up to $24.3\times$ faster than optimized CPU implementation. In terms of wall-clock time, the GPU-accelerated finite element solver is found to have up to $4\times$ speedup over CPU solver.

U. Kiran · S. S. Gautam · D. Sharma (✉)
Department of Mechanical Engineering, Indian Institute of Technology Guwahati, Guwahati,
Assam 781039, India
e-mail: dsharma@iitg.ac.in

U. Kiran
e-mail: ukiran@iitg.ac.in

S. S. Gautam
e-mail: ssg@iitg.ac.in

Keywords Finite element method · Parallel computing · Sparse matrix assembly · GPU

1 Introduction

Finite element method (FEM) is a popular numerical approach for finding approximate solutions to a wide range of scientific problems. Originally, developed for structural mechanics problem, FEM has spread to almost every discipline of engineering, including aerospace, bio-mechanics, fluid mechanics, electromagnetics and weather prediction [1]. However, numerical procedure to obtain solutions with FEM is computationally expensive. Depending on the nature of the problem, the time spent in FEM simulation might vary from hours to months. As the computational power of computer processors has increased in recent times, there is an increasing interest in performing realistic simulation with high accuracy. Therefore, it becomes extremely important to address the issue of high simulation time associated with the FEM. Parallel computers are often employed to handle the high computation cost associated with FEM-based simulation. A recently developed hardware to perform parallel computing is GPU. The hardware like GPUs has become quite popular due to high computational power, massive parallelism and high memory bandwidth. The current work attempts to find an efficient strategy to implement FEM on a GPU.

FEM solver consists of three expensive steps: computation of the elemental matrices, assembly of global stiffness matrix and solution of a system of linear equations given by the global stiffness matrix. Among these, the solution of the system of equations is often the most time-consuming step in FEM. Consequently, GPU implementations of linear solvers have been studied the most [2–5]. The overhead in computation of elemental matrices and their assembly can be significantly high for large-scale problems involving complex differential operators and higher-order finite elements. In nonlinear problems, the computation of elemental matrices and assembly is required multiple times, and therefore, it can contribute to high simulation timings [6].

The global stiffness matrix obtained in FEM is sparse as elements are locally connected to each other in discretized domain. The sparse matrix contains a mix of zero and nonzero values. Since zero values are not useful for computation, only nonzero values are stored in memory. For this purpose, different sparse storage formats like coordinate format (COO), ELLPACK format (ELL), CSR, etc. [7] have been used in the past implying different levels of memory utilization. However, working with a sparse storage format is not a trivial task as each format introduces a unique data structure, which might not be efficient for other computational steps of FEM. The assembly into global stiffness matrix must take into account the data structure of underlying sparse storage format and accordingly adopt an efficient strategy. The most popular way of constructing a sparse matrix in FEM is a two-step procedure. The first step assembles the elemental contribution in COO format where three arrays are used to store nonzero values, row indices and column indices of a matrix.

However, the value array usually contains multiple entries for the same values of row and column indices. In second step, each of the arrays are sorted and entries having repeated values in row and column arrays are consolidated. The global stiffness matrix obtained in COO format can now be converted into any other sparse format, if required. This approach works well for CPU and has been implemented by libraries like [8, 9]. However, this approach does not suit well to the GPU architecture. GPUs have limited memory, and performance of GPU-based code is highly dictated by the amount of memory access. The first step of assembly into COO format allocates more memory than required by the actual number of nonzeros. This puts pressure on GPU memory and limits the size of the problem. The second step of assembly into COO format involves sorting and accumulation of repeated entries. These operations are memory intensive and require a lot of data movement, which is not favorable for the best performance on GPU. Additionally, the COO format is not considered efficient for iterative linear solvers and most of the GPU accelerated libraries recommend other sparse formats like CSR for better performance. There are few related works in literature that attempts to implement finite element assembly on GPU through various approaches. In [10], assembly into COO format is demonstrated on GPU and compared with direct assembly into CSR storage format. It was reported that direct assembly into CSR format is slower than assembly into COO format. However, some recent literatures [11–13] demonstrate the excellent GPU performance for direct assembly into CSR sparse formats. In [11], the assembly process for structured mesh is decomposed into two steps where the location into CSR storage format is computed by performing some pre-processing with mesh connectivity. In [13], the assembly for unstructured mesh is performed over GPU in three compute kernels which includes computation of elemental matrices. The direct assembly into CSR format is presented in [12], where simultaneous computation of elemental matrices and assembly is performed in a single compute kernel.

The current work presents a GPU-based parallel strategy to assemble elemental matrices directly into CSR sparse storage format. The assembly is performed by pre-computing the locations into value array of CSR format for each nonzero values using mesh connectivity. The proposed strategy does not require any additional memory in GPU and provides global stiffness matrix in CSR format to use in linear solvers. The elemental matrices are computed on-the-fly without storing explicitly in GPU memory. The proposed strategy is applicable to unstructured mesh.

The paper is organized as follows. Section 2 provides a brief overview of problem formulations. Section 3 discusses the GPU strategy for finite element assembly. Section 4 presents the results of computational experiments and discusses the parallel performance. Finally, the conclusion is given in Sect. 5.

2 Problem Formulation

Consider an elastic body Ω subjected to body force \mathbf{b} and surface traction \mathbf{t} . The governing differential equation over the body is given as

$$\nabla \cdot \boldsymbol{\sigma} + \mathbf{b} = 0, \quad \forall \mathbf{x} \in \Omega, \quad (1)$$

where $\boldsymbol{\sigma}$ is the Cauchy stress. The governing equation must satisfy the given values of displacement and traction at the boundary, which is given as

$$\begin{aligned} \mathbf{u} &= \mathbf{u}_0, \quad \forall \mathbf{x} \in \Gamma_u, \\ \mathbf{t} &= \mathbf{t}_0, \quad \forall \mathbf{x} \in \Gamma_f, \end{aligned}$$

where \mathbf{u}_0 is the given displacement on boundary Γ_u and \mathbf{t}_0 is the given traction on surface Γ_f . The Cauchy stress is related to strain $\boldsymbol{\epsilon}$ through the constitutive relation, which is given as

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon}. \quad (2)$$

The relationship between strain and displacement is given by,

$$\boldsymbol{\epsilon} = \frac{1}{2}[\nabla \mathbf{u} + \nabla \mathbf{u}^T]. \quad (3)$$

The geometry of the problem is subdivided into a number of smaller entities called as elements. The displacement over an element is given by

$$\mathbf{u}(\mathbf{x}) = \mathbf{u}_i N_i(\mathbf{x}),$$

where \mathbf{u}_i is the vector of nodal displacement and N_i is the interpolation function associated with i^{th} node. The above equation can be rewritten as

$$\mathbf{u} = \mathbf{N}\mathbf{u}. \quad (4)$$

where \mathbf{N} is the shape function matrix and \mathbf{u} is the elemental displacement vector (refer to [1] for more details). The substitution of discrete variables into weak form gives finite element equation as

$$\mathbf{K}\mathbf{U} = \mathbf{F}, \quad (5)$$

where $\mathbf{K} = \sum_e \mathbf{K}^e$, $\mathbf{F} = \sum_e \mathbf{F}^e$, and \mathbf{U} is the global vector of unknown nodal displacements. The elemental quantities are given as

$$\mathbf{K}^e = \int \mathbf{B}^T \mathbf{D} \mathbf{B} dV, \quad \mathbf{F}^e = \int \mathbf{N}^T \mathbf{b} dV + \int \mathbf{N}^T \mathbf{t} ds.$$

where \mathbf{B} is the strain–displacement matrix and \mathbf{D} is the elastic constitutive matrix. The computation of elemental matrices is done by computing the integral numerically using Gauss quadrature rule. Please refer to [1] for further details.

3 GPU Implementation of Finite Element Assembly

3.1 *Parallel Computing on a GPU*

GPUs are specialized processor architecture that provides massive parallelism, high memory bandwidth and massive threading capability. Compared to CPUs, which are designed to reduce latency of an operation, GPUs are built to handle data parallel task and provide high computational throughput.

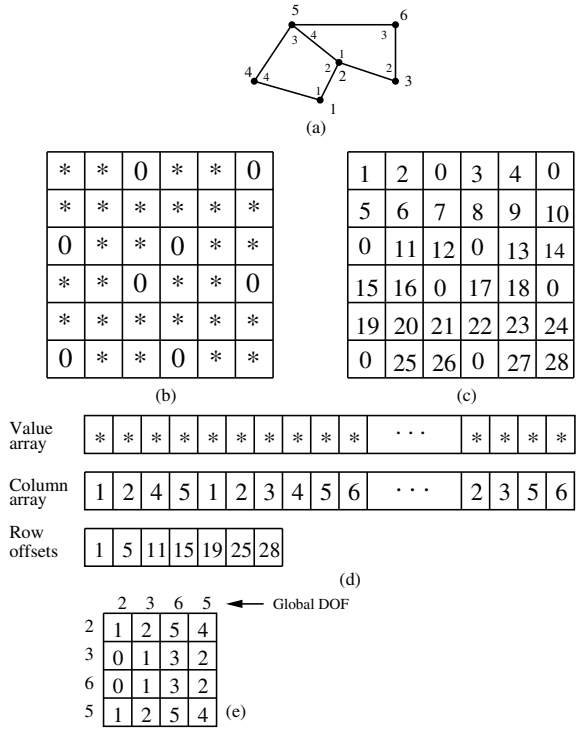
Compute Unified Device Architecture (CUDA) is a parallel programming model for general-purpose computation on NVIDIA GPUs. A typical CUDA program consists of CPU code with desired number of function calls to CUDA kernels. A kernel in CUDA is a function that uses GPU resources to perform computation in parallel. CUDA provides an elegant way to access different types of memory available in a GPU. The global memory is an off-chip memory in a GPU and has the highest latency and the largest size. The on-chip memories named as shared memory and registers are smaller in size and have the lowest latency. Readers are referred to [14] for more details on the CUDA programming model.

3.2 *Pre-Computing Indices into CSR Matrix*

The CSR sparse storage format uses three arrays to store a sparse matrix. These arrays are: value array to store nonzero values, column indices array to store column indices of nonzero values and row offsets array to store the locations of beginning of each row. The row offset array has size equal to size of the matrix incremented by one, the last entry contains the total number of nonzero values. Figure 1a shows an example mesh consisting of quadrilateral elements with a single degree of freedom per node. The full global stiffness matrix corresponding to Fig. 1a is displayed in Fig. 1b, where * denotes nonzero values, numbered row-wise (as stored in CSR format) as shown in Fig. 1c. The global matrix has a total of 28 nonzero values, and therefore, the value and the column arrays are allotted space for 28 values. The row offsets array is assigned the size of seven values. Figure 1d shows the global stiffness matrix in CSR format where the values of column array are displayed for first, second and last rows.

The local to global mapping of elemental DOFs only provides rows and column indices for a particular nonzero value. While this information is sufficient to find the exact position in full global matrix, the locations into compressed global matrix cannot be found. The direct assembly of elemental matrices into CSR format requires prior knowledge of locations of nonzero values in value array. For example, let's assume that values corresponding to third node in element two are accumulated in the global matrix. The corresponding global DOF number is six, which means that the values should be modified in sixth row of the global matrix. Figure 1d shows the row offset value for 6th row as 25, indicating that the values corresponding to 6th row

Fig. 1 CSR storage format and pre-computing locations into the value array



lie at locations starting from 25th position in column and value arrays. However, if one wants to modify the value in 5th column of 6th row, the location for 5th column needs to be searched in column array. As shown in column array of Fig. 1d, the 5th column comes at 3rd position in column indices for 6th row. Therefore, to modify a value at 6th row and 5th column in global stiffness matrix, one has to make changes at 27th (25 + 2) position in value array. The exact location is obtained by adding relative position of desired column with respect to the first column in the corresponding row. Whenever the assembly is performed, the relative position of the column index of a nonzero value needs to be searched in the column array, which can be expensive for unstructured meshes in three dimensions (3D). It is to be noted that the relative position of column indices remains fixed as long as a mesh is fixed. This implies that expensive search operations into column array can be done prior to the assembly step, and relative positions of column indices can be pre-computed and stored for later use. Figure 1e shows a 4 × 4 matrix that contains relative positions of column indices for each nodes in element 2 (see Fig. 1a). The relative position of a column index depends on the immediate neighborhood of the node and elements with which it is connected. If a node is associate with multiple elements, the relative position is found for each element. If a node has multiple DOF associated with it, the same relative position can be used for all DOFs. For example, the same 4 × 4 matrix as shown in Fig. 1e is sufficient if quadrilateral elements in Fig. 1a had two DOFs per

node. The relative positions of column indices are referred as CSR indices in the following discussions.

The flowchart for the computation of CSR indices is shown in Fig. 2. In the first step, for each node in the finite element mesh, a list of immediate neighboring elements is obtained. This list is transformed into a neighboring nodes list by including the connectivity of each element. The neighboring nodes list is then sorted in ascending order, and duplicate entries are removed. This procedure is repeated for all nodes, and the final sorted node list is stored in an array. For each node, its neighboring nodes list denotes the position of nonzero values in corresponding rows of global stiffness matrix. The neighboring nodes list can be easily used to generate column indices of nonzero values for each row of global matrix in CSR format. Now, the position of any nonzero value can be obtained by performing a search into the neighboring nodes list instead of column indices. As shown in Fig. 2, the CSR indices are computed for each element in a loop by retrieving neighboring node list for each of its node. The position of all other nodes in the connectivity list is searched, and relative position is stored in an array. In GPU implementation, loop over nodes and loop over elements are parallelized by allocating one thread to each node and element, respectively.

3.3 Computation of Elemental Matrices and Assembly

The computation of elemental matrices is an embarrassingly parallel operation as each element is independent of others in terms of input data and computation. Each GPU thread is assigned the task to compute the elemental matrix of one element. The respective thread reads the input data for each element such that collective memory access for a warp is coalesced. The Gauss quadrature rule is implemented for numerical integration of fully integrated 8-noded hexahedral element. Since, on-chip memories are limited for a GPU thread, each thread makes extensive use of local memory to store intermediate variables and final output.

The assembly in FEM is not amenable for parallelization due to data read–write conflict among threads. Each DOF in the finite element mesh has a corresponding row in the global stiffness matrix. Since a DOF is shared by multiple elements, each row in global stiffness matrix receives contribution from multiple elemental matrices. When assembly is done in parallel, threads assigned to different elements may try to read or write values simultaneously to the same memory locations. Such operations are conflicting and create an issue of data race condition.

In this paper, mesh coloring method [12] is employed to manage data race among conflicting threads. The mesh coloring is a popular and robust method in which finite element mesh is divided into multiple sets, identified with a color. The elements belonging to a colored set do not have any common node/DOF. The computation for elements belonging to a color can be performed in parallel without any conflict. The computation for all colors is done in sequence.

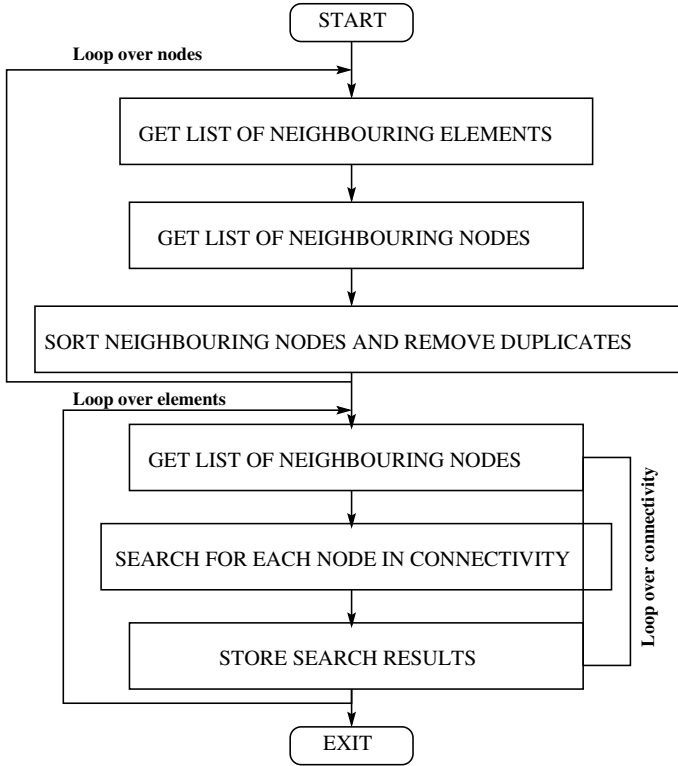


Fig. 2 Flowchart indicating steps in computation of CSR indices

The same CUDA kernel that computes elemental matrix also performs assembly into global stiffness matrix in CSR format. Each thread reads global connectivity, corresponding row offsets and CSR indices to locate the position of nonzero values into the global matrix stored in CSR format. The assembled global matrix in CSR format can be directly used in any linear solver after the application of boundary condition. This assembly strategy is similar to the one presented in [12].

4 Results and Discussion

To assess the performance of the proposed strategy, numerical experiment is conducted over a large-scale 3D linear elasticity problem. The numerical experiment is conducted on a machine with the configuration given as: CPU—Intel Xeon® E5-2650 clocked at 2.6 GHz and 128 GB RAM, GPU—NVIDIA Tesla K40c clocked at 745 MHz with 12 GB of DRAM. The solution of the linear system of equations is done by conjugate gradient iterative solver from GPU-based CUSP library [15]. Since

the focus of the current work is not on acceleration of linear solver step of FEM, the same solver is used in both CPU and GPU implementations. The meshes for computational examples are generated in ABAQUS software package with 8-noded hexahedral elements.

A unit cube is taken with boundary conditions as shown in Fig. 3. All DOFs at the bottom are constrained while a distributed load is applied at the top surface. The problem is solved with the following parameters: Young's modulus (E) = 200 GPa, Poisson's ratio (ν) = 0.3 and Load (P) = 400 MPa. The cube is discretized with different number of elements to get different sizes of mesh, as shown in Table 1.

The computational time for CSR indices on CPU and GPU is presented in Fig. 4. For the coarsest mesh M1, the CPU takes 1.14 s to compute CSR indices which increases almost linearly up to 15.67 s for the finest mesh. The GPU-based implementation achieves speedup in the range $4.98\times$ to $6.48\times$ for all the meshes. On GPU, the computation of CSR indices for the finest mesh is completed in just 2.42 s. In order to assess the proposed strategy for direct assembly into CSR format, a comparison is made with sparse assembly feature of the Eigen library. The functions `tripletList.push_back()` and `setFromTriplets()` from Eigen are used for assembly which collect elemental contributions in COO format and transform into CSR format, respectively. As shown in Fig. 5, the assembly with Eigen consumes large amount of time compared to the proposed strategy on CPU. The proposed strategy achieves approximately $9.3\times$ speedup over Eigen, performing assembly for the finest mesh in 35.08 s, including the time spent in computation for CSR indices. Figure 6 shows the combined computational time of elemental matrices and assembly by the proposed strategy on CPU and GPU. It can be observed that expensive numerical integration adds a significant amount of time to the assembly. The GPU solver achieves speedup

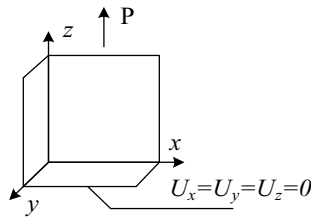


Fig. 3 A unit cube with boundary conditions

Table 1 Mesh for cube problem

Mesh	Elements	Nodes	Degrees of freedom
M1	110,592	117,649	352,947
M2	216,000	226,981	680,943
M3	438,976	456,533	1,369,599
M4	884,736	912,673	2,738,019
M5	1,520,875	1,560,896	4,682,688

in the range from $21.18\times$ to $24.25\times$ for all the mesh sizes. For 4.7 million DOFs, the GPU solver takes only 6.59 s to perform computation of elemental matrices and their assembly (including time for CSR indices). The wall-clock time for the CPU solver, GPU solver and CPU solver with Eigen library is presented in Fig. 7. Due to huge overhead in assembly, the CPU solver using Eigen library performs poorly and consumes the highest amount of time. The proposed CPU solver consumes moderate amount of time and performs linear elastic analysis of cube for the finest mesh in 213.95 s. The proposed GPU solver achieves speedup in the range from $8.13\times$ to $10.14\times$ over CPU solver using Eigen and $3.34\times$ to $4.01\times$ speedup over the proposed CPU solver for all mesh sizes. It should be noted that CPU solvers use GPU-based linear solver.

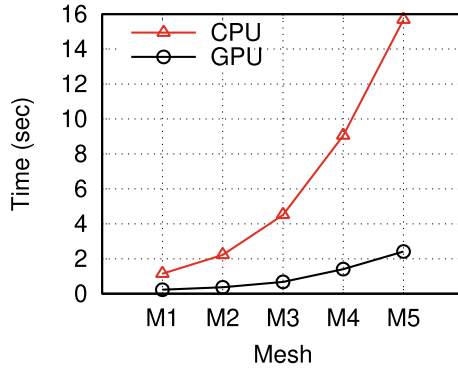


Fig. 4 Computation of CSR indices

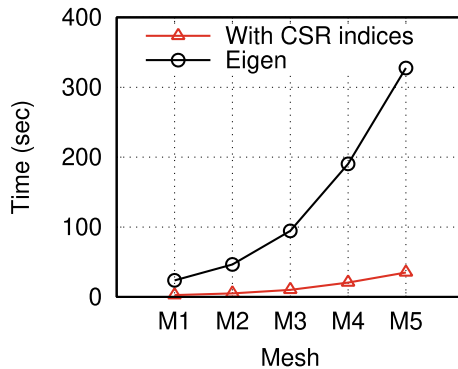


Fig. 5 Assembly with CSR indices on CPU compared with Eigen library

Fig. 6 Numerical integration and assembly on CPU and GPU

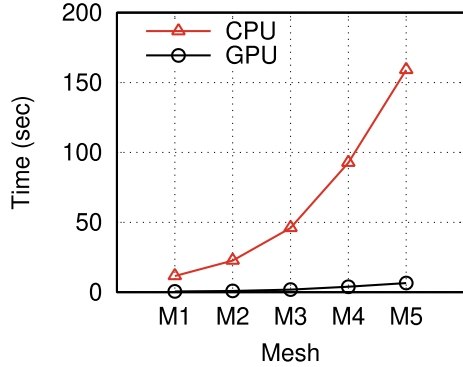
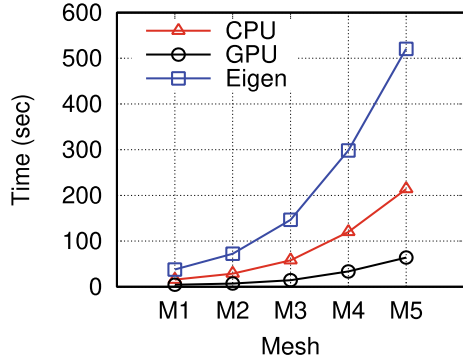


Fig. 7 Wall-clock time for CPU, GPU and CPU solver using Eigen for assembly



5 Conclusion

FEM is widely used numerical method to find solution of engineering problems. However, computation in FEM can be expensive and often leads to high execution time for real-world problems. In this paper, we have addressed the issue of high computational time associated with assembly step of FEM and proposed parallel strategy to perform assembly on the GPU. The proposed assembly strategy uses pre-computed indices into the value array of CSR sparse format to assemble elemental matrices into global matrix. The performance characteristic of the proposed assembly strategy is assessed by solving a linear elasticity problem. For the finest mesh with 4.7 million degrees of freedom, the pre-computing of indices into CSR format takes 15.67 s on CPU and 2.42 s on GPU. On CPU, the proposed assembly strategy is found $9.3\times$ faster than the sparse assembly function from Eigen library. The proposed GPU assembly strategy achieves speedup in the range from $21.18\times$ to $24.25\times$ over the proposed CPU strategy for all mesh sizes in computation of elemental matrices and assembly. As a result of GPU acceleration, an overall speedup of $3.34\times$ to $4.01\times$ is obtained with respect to the CPU implementation.

Acknowledgements This work was supported by the Science and Engineering Research Board [IMP/2019/000276, SB/FTP/ETA- 0008/2014].

References

1. Zienkiewicz OC, Taylor RL, Zhu JZ (2005) The finite element method: its basis and fundamentals, 6th edn. Butterworth-Heinemann, Oxford
2. Georgescu S, Chow P, Okuda H (2013) GPU acceleration for FEM-based structural analysis. *Archiv Comput Methods Eng* 20(2):111–121
3. Filippone S, Cardellini V, Barbieri D, Fanfarillo A (2017) Sparse matrix-vector multiplication on GPGPUs. *ACM Trans Math Softw (TOMS)* 43(4):1–49
4. Kiran U, Sanfui S, Ratnakar SK, Gautam SS, Sharma D (2019) Comparative analysis of GPU-based solver libraries for a sparse linear system of equations. In: *Advances in computational methods in manufacturing*. Springer, Singapore, pp 889–897
5. Kiran U, Gautam SS, Sharma D (2020) GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices. *Computing* 102(9):1941–1965
6. Kiran U, Agrawal V, Sharma D, Gautam SS (2019) A GPU based acceleration of finite element and isogeometric analysis. In: Liu GR, Xiangguo GX (eds) *Proceedings at the 10th international conference on computational methods (ICCM2019)*. ScienTech Publisher, Singapore, pp 641–651
7. Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA. Nvidia Technical Report NVR-2008-004, Nvidia Corporation
8. Guennebaud G, Jacob B (2021) Eigen V3, <http://www.eigen.tuxfamily.org>
9. The MathWorks. Inc. (2021) MATLAB version R2021a. Natick, Massachusetts
10. Dziekonski A, Sypek P, Lamecki A, Mrozowski M (2012) Finite element matrix generation on a GPU. *Progress Electromagn Res* 128:249–265
11. Sanfui S, Sharma D (2017) A two-kernel based strategy for performing assembly in FEA on the graphics processing unit. In: *International conference on advances in mechanical, industrial, automation and management systems (AMIAMS)*, pp 1–9. IEEE
12. 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
13. Sanfui S, Sharma D (2020) A three-stage graphics processing unit-based finite element analyses matrix generation strategy for unstructured meshes. *Int J Numer Meth Eng* 121(17):3824–3848
14. NVIDIA Corporation. NVIDIA CUDA C++ programming guide, version 11.6 (2022)
15. Dalton S, Bell N, Olson L, Garland M (2014) Cusp: generic parallel algorithms for sparse matrix and graph computations. version 0.5.0, <http://cusplibrary.github.io>