Symbolic and Numeric Kernel Division for GPU-based FEA Assembly of Regular Meshes with Modified Sparse Storage Formats

Subhajit Sanfui
Department of Mechanical Engineering
Indian Institute of Technology, Guwahati
Assam, India
Email: s.sanfui@iitg.ac.in

Deepak Sharma
Department of Mechanical Engineering
Indian Institute of Technology, Guwahati
Assam, India
Email: dsharma@iitg.ac.in

This paper presents an efficient strategy to perform the assembly stage of finite element analysis (FEA) on general-purpose graphics processing units (GPU). This strategy involves dividing the assembly task by using symbolic and numeric kernels, and thereby reducing the complexity of the standard single-kernel assembly approach. Two sparse storage formats based on the proposed strategy are also developed by modifying the existing sparse storage formats with the intention of removing the degrees of freedom-based redundancies in the global matrix. The inherent problem of race condition is resolved through the implementation of coloring and atomics. The proposed strategy is compared with the state-of-the-art GPU-based and CPU-based assembly techniques. These comparisons reveal a significant number of benefits in terms of reducing storage space requirements and execution time and increasing performance (GFLOPS). Moreover, using the proposed strategy, it is found that the coloring method is more effective compared to the atomics-based method for the existing as well as the modified storage formats.

1 Introduction

Finite element method (FEM) ([1]) is a numerical technique that is used for solving partial differential equations (PDEs), which arise in various fields of science and engineering. Finite element analysis (FEA) is the process in which real-world problems and physical phenomenon are simulated using FEM or alike in order to obtain the response of the systems. Typically, an FEA consists of pre-processing, solver, and post-processing stages. In the pre-processing stage, the domain of a given problem is decomposed into finite elements for which their order and type are defined, along with the material constants. Next, in the solver stage, the elemental matrix for every finite element is calculated and assembled to form the global stiffness matrix. This generates a linear system of equations ($Ax = b$), which is solved after application of specified boundary conditions. Lastly, the post-processing stage helps understand and visualize the outcomes of the analysis.

FEA can be computationally expensive when dealing with complex problem domains and very fine meshes; this is also true for some applications such as topology optimization ([2, 3, 4]). In particular, among all the stages in FEA, the solver stage is the most time-consuming one ([5, 6, 7]). The common remedy to this challenge is performing FEA computation in parallel ([5]). Among many available techniques, GPU-based parallelization studies consist a significant part of these FEA implementations in the literature, owing to the low cost to performance ratio of GPUs and the success achieved by many researchers in GPU parallelization of applications ranging from CFD simulations ([8]) to molecular dynamics ([9]). The computation of elemental stiffness matrices in FEA can be performed independently and in parallel ([10]). Thus, many researchers have found the calculation of elemental stiffness matrices to be highly suitable for parallel computing ([11, 12, 13, 14]). Besides this, the assembly stage is another area in FEA that can be computationally expensive for a number of real-life applications ([6, 10, 15, 16]). Furthermore, the position of the non-zero entries in the assembled sparse global matrix can often be highly irregular, which is another crucial factor in the efficient parallel implementations of FEA on GPU.

Assembly of finite element matrices on CPU is a sequential process where all the elemental stiffness matrices

are assembled one-by-one. This approach, when applied on the GPU in parallel is termed as `addto` method ([6, 17]). The approach, although straightforward, involves complications such as race condition, the problem of handling large amounts of data on GPU and more. These issues are alleviated in another approach termed as the local matrix approach (LMA) ([18]) by obviating the need to explicitly assemble the entire global stiffness matrix at any point of the analysis. However, the advantages of LMA come at the additional cost of redundant computations resulting from on-the-fly calculations. This, inadvertently increases the computation time of the LMA and other matrix-free methods. Thus, a trade-off can be noted between the inherent complications of the `addto` method and the redundant computations of the LMA method.

It is seen from the literature that assembling the global stiffness matrix on the GPU poses several challenges. The primary challenge in the `addto` method is efficient handling of the sparse data for the global stiffness matrix on GPU. Here, handling the data involves efficient storage of the non-zero entries using a sparse storage scheme and efficient access of the non-zero entries through kernel design while taking care of the inherent race condition. This key challenge is addressed in the present work by incorporating a division-based assembly scheme along with a new sparse storage system based on the state-of-the-art sparse formats. Splitting the assembly operation into two different parts reduces the complexity of a single-kernel approach for `addto` method. Furthermore, the new formats facilitate efficient access to the matrix data while significantly reducing the storage requirements on the GPU by exploiting the structure of a FE matrix to reduce the memory footprint of the resulting matrix. The key contributions of this paper are summarized as

- Developing novel symbolic and numeric GPU kernels for assembly.
- Implementing three new sparse storage formats by modifying COO and ELL formats.
- Comparative analysis of the modified sparse storage formats with the standard formats using the proposed assembly strategy.
- Comparative analysis of coloring and atomics-based implementations with the proposed kernel division strategy and the new sparse formats.

The paper is organized in six sections. Section 2 presents the relevant literature survey of assembly on GPU. Preliminaries to this paper are discussed in Section 3 including details about GPU architecture and CUDA, FEM for elasticity and sparse storage formats. In Section 4, the proposed assembly strategy and its implementation details are presented. The results are discussed in section 5 and the paper is concluded in section 6.

2 Relevant Studies

Among the early studies on GPU-based assembly implementations, some researchers have computed each non-zero entry of the global matrix in parallel using mathematical expressions ([15, 16]). These were completely application-specific implementations that helped reduce execution times. As the GPU architecture and the programming APIs (CUDA, OpenCL) kept evolving, more intricate applications began emerging. For example, the study by Komatitsch et al. [10] simulated the phenomenon of seismic wave propagation resulting from earthquakes using NVIDIA GPUs for very high order spectral elements. In this study, instead of using simple mathematical expressions for global stiffness entries like in the early studies, the authors carried out the generation of local matrices and assembly for all the elements in parallel. Additionally, they handled the inherent problem of race condition by assigning a color to each of the element and performing the computation for each color sequentially; this strategy ensured that no two elements with a shared DOF were assembled simultaneously. A similar coloring strategy for assembly on GPU was reported by Cecka et al. [6], along with other assembly strategies, for unstructured 2D meshes. The authors used different techniques for overcoming the race condition (coloring, non-zero-wise computation), among which the coloring method incurred excessive memory accesses, thereby reducing the performance.

Later, the study by Markall et al. [18] compared assembly by using the local matrix approach (LMA) with the standard assembly. LMA is a matrix-free assembly technique that was used to overcome challenges arising from GPU-based assembly implementations. In another study by Kiss et al. [19], the LMA method was used, for which the elemental stiffness matrices were computed on the GPU, when necessary. Using large datasets, it was found that the low-storage compute-intensive implementation was suitable for many-core systems. However, afterward, Fu et al. [11] demonstrated that the LMA method on GPU for 3D grids is inferior to the traditional assembly method in terms of performance. In this work, an assembly method was proposed which is based on patch-based division of the problem domain.

Authors in [17, 20] presented a reduction-based implementation for assembly on single and multi-GPU systems. The global stiffness matrix was first assembled into coordinate sparse storage format (COO) and afterward transformed into compressed sparse row (CSR) storage format by performing reduction in parallel. Carrion et al. [21] accelerated a coupled BEM-FEM procedure on the GPU for a dynamic soil-structure interaction problem. The assembled mass and stiffness matrices were parallelized on the GPU, both row and column-wise, to perform linear algebra operations. Among more recent works, Dinh and Marechal [22] reported an FEM implementation for real-time applications that is similar to the approach taken by Dziekonski et al. [17, 20]. After computing the elemental stiffness matrices, the resultant non-zero entries were arranged in unsorted coordinate format with multiple non-zero entries for the same stiffness matrix location. Following this, parallel radix sort and a reduction algorithm is applied to the COO arrays to produce the final global stiffness matrix. Sanfui and Sharma [23] presented an implementation of assembly where the key idea is to divide the workload into different

stages for performing only the assembly stage on GPU.

Later, the study by Zayer et al. [24] used the assembly stage in the form of matrix-matrix multiplication. This approach enabled the authors to reduce the memory footprint of the application as well as the amount of pre-processing required for the assembly operation. A recent study by Kiran et al. [25] implemented the warp-based assembly method using eight-noded hexahedral elements with the aim of efficient utilization of on-chip memory resources. Coloring method was used for race condition and speedups of up to $8\times$ were achieved over standard assembly by element strategy on GPU. The study by Gribanov et al. [26] implemented an implicit finite element model with cohesive zones and collision response on GPU. During assembly, the race condition was handled using atomic operations of the CUDA toolkit. An implementation of FEA that calculated and stored only the symmetric half of the elemental and global stiffness matrix was presented in the study by Sanfui and Sharma [27]. Local matrix computation and assembly were both accelerated on GPU resulting in speedups of up to $2\times$ over an implementation that computed and stored the entire matrices. The study by Kiran et al. [28] presented an implementation of matrix-free finite element solver, where only the symmetric half of the local stiffness matrices are used. By using a novel data structure for storing the symmetric matrices, speedups of up to $5\times$ were achieved. In the study by Sanfui and Sharma [29], a multi-stage GPU-based assembly method was presented for unstructured meshes. This strategy used a new data structure called the *neighbor matrix* to assemble the global stiffness matrix. Furthermore, two approaches were compared with each other where the elemental computation and assembly were performed in the same as well as in separate kernels. Speedups of up to $6\times$ were achieved for the proposed strategy and the same kernel approach was found to outperform the separate kernel approach for all the tested problems.

Due to the large and sparse nature of the resulting global stiffness matrix, specific sparse storage formats need to be used when storing data on the GPU. A plethora of different sparse storage formats have been used in the literature, including the standard ones such as COO, CSR ([17]), ELL ([12]) and the ones that have been developed for specific purposes and applications on the GPU such as ELL-WARP ([30]), pJDS ([31]). The idea behind the new sparse formats is to modify the standard sparse storage formats by making specific assumptions about the structure of the sparse matrix. These modifications help by reducing the execution time and/or the memory footprint of the application. Several GPU-friendly sparse storage formats have also been developed in the literature ([32]) and these make no assumptions on the matrix structure or the type of application.

From the literature review presented, it can be seen that the primary challenge to performing `addto` assembly lies in efficient storage and access of the global matrix. As a result, the research effort in this area has been focused on kernel design of assembly and efficient sparse storage schemes based on GPU. Furthermore, several remedies have been suggested in the literature for handling the race condition. Although, the LMA and others matrix-free methods do alleviate some of the difficulties with `addto` assembly, the benefits come at a cost of increased computation.

3 Preliminaries

3.1 GPU architecture and CUDA

GPUs are many-core processors that were initially developed for rendering graphics in the early '90s especially for video games. Due to the inherent parallel structure, GPUs are much more capable of handling large blocks of data that can be processed in parallel than the general purpose CPUs. The throughput-oriented nature of GPU also makes it suitable for performing general purpose computation involving large amount of data. Being a many-core processing unit, GPU architecture varies significantly from the traditional CPU architecture, that are multi-core in nature. The key difference between the two is that CPUs put the emphasis on low latency making sequential processing faster, whereas GPUs put the emphasis on a high throughput that makes highly parallel applications run efficiently.

The GPU used in this paper is from NVIDIA. Compute Unified Device Architecture (CUDA) is a parallel programming API from NVIDIA, which enables developers to program GPUs for general purpose computing. CUDA is designed to work with programming languages such as C, C++, and Fortran. Using specific set of extensions to these languages, a developer is able to write kernels that generate very high number of threads to divide a workload for parallel processing. CUDA also provides the flexibility to the developer to use the highly sophisticated memory hierarchy of NVIDIA GPUs and a host of other features for writing efficient parallel applications.

3.2 Details of FEM for Linear Elasticity

A linear elasticity problem is a boundary value problem in which the governing PDEs include the strong form for linear elastic material, strain-displacement equations, and constitutive relations from Hooke's law ([1]), which are given as

$$\begin{aligned} \sigma_{ij,j} + b_i &= \rho \ddot{u}_i, \qquad i,j = 1,2,3, \\ \varepsilon_{ij} &= \tfrac{1}{2}(u_{j,i} + u_{i,j}), \\ \sigma_{ij} &= C_{ijkl}\varepsilon_{kl}. \end{aligned} \tag{1}$$

Here, b_i and u_i are the body forces per unit volume and displacement, σ_{ij} is the component of Cauchy stress tensor, ε_{kl} is the strain, C_{ijkl} is the material elasticity tensor, and ρ is the density. It is noted that an isotropic and homogeneous material is considered in this paper. This strong form is subjected to the displacement and traction boundary conditions that are given as

$$u_i = \bar{u_i} \text{ on } \Gamma_u, \qquad t_i = \bar{t_i} \text{ on } \Gamma_t, \tag{2}$$

where $\bar{u_i}$ is the boundary displacement specified on boundary Γ_u and $\bar{t_i}$ is the boundary traction specified on boundary Γ_t.

FE formulation is made by converting the governing PDEs into their weak form using the Galerkin weighted residual approach and solved over a discretization of the problem domain consisting of finite elements ([1]).

The displacement within an element using shape functions $[N]^e$ is approximated as

$$\{u\}^e = [N]^e\{\tilde{u}\}^e, \tag{3}$$

where $\{\tilde{u}\}$ is the nodal displacement vector. in the three directions. Substituting equation (3) into the weak form resulting from equation (1) and performing assembly over all the elements, we get

$$\sum_e [[M]^e\{\ddot{u}\}^e + [K]^e\{\tilde{u}\}^e - \{f\}^e_{ext}] = 0, \tag{4}$$

where $[M]^e$ is the local mass matrix, $[K]^e$ is the local stiffness matrix, and $\{f\}^e_{ext}$ is the local force vector.

In this paper eight-noded hexahedral elements are used for discretization. The eight shape functions for this element are given as $[N_i]^e = \frac{1}{8}(1+\xi\xi_i)(1+\eta\eta_i)(1+\zeta\zeta_i)$, where ξ_i, η_i and ζ_i are the coordinates of the i^{th} node. The expression for the elemental stiffness matrix $[K]^e$ in equation 4 is given by

$$\begin{aligned}[K]^e &= \int_V [B]^T[D][B]\, dxdydz \\ &= \int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} B(\xi,\eta,\zeta)^T DB(\xi,\eta,\zeta) \mid J(\xi,\eta,\zeta) \mid d\xi d\eta d\zeta \\ &= \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} w_{\xi_i} w_{\eta_j} w_{\zeta_k} B(\xi_i,\eta_j,\zeta_k)^T DB(\xi_i,\eta_j,\zeta_k) \mid J(\xi_i,\eta_j,\zeta_k) \mid \end{aligned} \tag{5}$$

$[B]$ is called the strain-displacement matrix and contains the partial derivatives of the shape functions. D is constitutive matrix and J is the Jacobian matrix. w_{ξ_i}, w_{η_j} and w_{ζ_k} are the Gauss Quadrature weights for numerical integration. The elemental stiffness matrix given in equation (5) is then assembled using the connectivity of the mesh into a global $[K]$. The details of assembly is given later in Section 4.

3.3 Sparse Storage Formats

The global stiffness matrix $[K]$ is generally sparse. The non-zero entries of the matrix are stored in the specialized storage formats (For example, COO, CSR and ELL). These formats reduce the storage requirement of keeping the non-zero entries of $[K]$, while solving a system of equations ([6, 17]).

COO is the simplest format for storing sparse matrices. It is found to be an efficient format ([17]) for GPU-based assembly operations. The COO format is made up of two arrays for the indices and one array for storing the values of the non-zero entries, as shown in figure 1. Another format

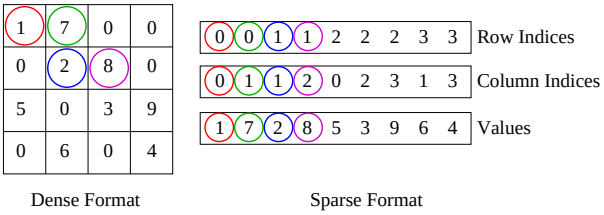

Fig. 1. COO sparse storage format.

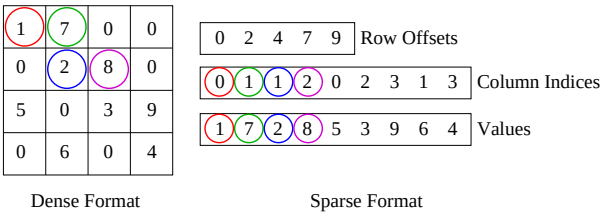

Fig. 2. CSR sparse storage format.

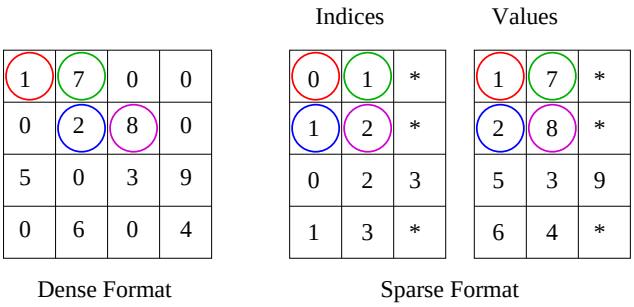

Fig. 3. ELL sparse storage format.

similar to COO format is the CSR format. In this format, the row offsets are saved instead of the row indices, as shown in Figure 2. As a result, this format further reduces the storage requirement. Another important format is the ELL format. In this format, two 2D matrices are used that store the values and indices of the non-zero entries. Each of these two matrices have as many rows as in $[K]$ but only have as many columns as the highest number of non-zero in any particular row of $[K]$. Figure 3 shows the example of ELL format.

4 Assembly on GPU

For performing the assembly, the local stiffness matrix K^e is calculated by using equation (5). In our parallel implementation of K^e calculation, we adopt a similar approach described by [33], wherein one thread is allocated to compute each elemental stiffness matrix. The procedure is presented in algorithm 1. In order to compute K^e, the connectivity matrix C, nodal coordinate matrix C_d, constitutive matrix D are required. Furthermore, we adopt the same strategy of precomputing the shape functions (N_i) and its derivatives (dN_i) as done by [33] to avoid repetitive calculations. Each CUDA thread computes J, $\mid J \mid$ and J^{-1}, uses them to compute B and subsequently K^e, which is then stored in a one-dimensional array containing the elemental matrices of all the elements.

Algorithm 1 Parallel numerical integration for K^e calculation

Input: E: Number of elements; I: Total number of nodes; C: Connectivity matrix; C_d: Nodal coordinate matrix; D: Constitutive matrix;
Output: K^e: Elemental stiffness matrix;
int blockSize = 512;
int gridSize =((E - 1) / blockSize) + 1;
__global__ void numIt(K^e, C, C_d, D**)** ▷ Kernel Definition
int tid = blockIdx.x × blockDim.x +threadIdx.x; ▷ Unique thread ID
Read N_i and dN_i ▷ Pre-computed shape functions and derivatives
if $tid < E$ **then**
  for $j \leftarrow 8$ **do** ▷ Loop over Gauss points
    Compute $J, |J|, J^{-1}$;
    Compute B;
    Store K^e in a column-major order;
  end for
end if

4.1 Assembly

Algorithm 2 `addto` algorithm for assembly

Input: $ndof$: DOFs per node; $nnodes$: nodes per element; N: Total DOFs; E: Number of elements; $C[E, nnodes \times ndof]$: Connectivity matrix storing DOFs; $K^e[nnodes \times ndof, nnodes \times ndof]$: Elemental stiffness matrix; $f^e[nnodes \times ndof]$: Element load vector;
Output: K: Global stiffness matrix; F: Global load vector
$F[N] = 0$; $K[N,N] = 0$; ▷ Initialization
for $e \leftarrow 1 : E$ **do** ▷ Outer loop over elements
  Compute K^e; ▷ Compute elemental stiffness matrix
  Compute f^e; ▷ Compute element force vector
  for $i \leftarrow 1 : ndof$ **do** ▷ Inner loop over DOF
    Compute row index (row_ind);
    $F[C[e, row\_ind]] += f[i]$; ▷ Vector Assembly
    for $j \leftarrow 1 : ndof$ **do** ▷ Inner loop over DOF
      Compute column index (col_ind);
      $K[C[e, row\_ind], C[e, col\_ind]] += K^e[i][j]$; ▷ Matrix Assembly
    end for
  end for
end for

For performing assembly operation, the elemental entries are added to the global matrix based on the connectivity information. For GPU implementations of FEA assembly, different strategies have been used in the literature ([6, 18]). The traditional method is the `addto` method in which the assembly of each element into the global stiffness matrix is done sequentially. The `addto` method as shown in algorithm 2 is massively data-parallel in nature. This is due to the fact that the outer-loop over each element of the FE mesh can be performed independently (refer step 4). However, when it comes to GPU implementation, one major issue with the `addto` method is the possibility of multiple simultaneous threads trying to write to the same memory location. This is well known as the race condition or data race in parallel computing. This issue is handled in this paper by use of atomic operations and element coloring methods.

In the traditional assembly operation for FEA applications, the values and locations of the non-zero entries of the global matrix are filled in the sparse format ([6]). We follow a novel strategy in which the workload is divided into two distinct parts, henceforth mentioned as the *symbolic* and the *numeric* part of the computation. The *symbolic kernel* computes only the locations of the non-zero entries (row and column indices) in the sparse storage format in parallel. Owing to the fact that this computation is dependent solely on the mesh data, it can be executed independently. After calculating the row and column indices, they are stored onto the global memory in their respective locations. Thereafter, the *numeric kernel* is launched to fill the values of the non-zero entries based on the pre-filled sparse indices. This division of the assembly task reduces the complexity of the algorithm significantly and yields several benefits over the standard implementation on GPU.

An important aspect in the GPU implementation of assembly is to determine the exact number of non-zero entries for the sparse storage formats at the start of the kernel. This number is necessary for memory allocation and also for determining kernel launch parameters. In the present work, we derive an expression analytically using the elemental connectivity of the mesh for this purpose. The expression in equation (6) gives the number of non-zero entries in the sparse global stiffness matrix for a rectangular cuboid shaped structured mesh containing eight noded brick elements.

$$NZ = 108(n_x+n_y+n_z) - 162(n_xn_y+n_yn_z+n_zn_x) + 243n_xn_yn_z - 72. \quad (6)$$

Here n_x, n_y and n_z are the number of nodes in the three directions of the cuboid mesh. For this expression, the number of repeated write operations during assembly is subtracted from the total writes performed. A similar expression can be obtained for a different domain shape by following the same principle. It is noted that such algebraic expression may be cumbersome to derive for certain cases. In those cases, a simple code can be written that takes the connectivity information of the mesh as input and output the number of non-zero entries for the given type of elements.

Before the launch of the *symbolic* part, memory is allocated and the kernel launch parameters are fixed based on the number of non-zero entries calculated using equation (6). Thereafter, the matrices are assembled in the following two steps.

4.1.1 The Symbolic Kernel: *Node-by-Node* Implementation

The purpose of the symbolic kernel is to compute the indices of the non-zero entries in the sparse global stiffness matrix. This kernel works on the principle that the exact count of non-zeroes in any one row of the global matrix can be obtained beforehand by multiplication of DOFs per node to the no. of immediate neighbors to the node. If we take a node having *nng* number of neighbors, the rows corresponding to that particular node in the global stiffness matrix will have $(\text{DOF} \times (nng+1))$ non-zero entries. It can be seen from figure 4 that a node can have different numbers of neighboring nodes. For example in the figure, the number of neighbors to a *corner* node, an *edge* node, a *face* node and an *interior* node is 7, 11, 17, and 26, respectively. Total number of threads in the grid is equal to the number of nodes in the entire mesh, where each thread is responsible for computing one node exactly.

Algorithm 3 outlines the symbolic kernel, where the row and column indices are computed for all non-zero entry in parallel. d_i^j and S^j denote the DOFs and neighbor nodes to all the nodes j respectively. These are stored on the GPU global memory in step 7. The kernel is called in step 8 and a grid is launched on the GPU with number of threads equal to the total number of nodes in the FE mesh. Two sets R^j and C^j are declared for storing the row and column indices respectively for each node j. In step 12, R^j is filled by copying DOF d_i^j into it $(nng \times nd)$ times. From figure 5, it can be seen that R^j is filled with DOFs of node j, one after another. Hence, $(nng \times nd^2)$ entries are stored in R^j for node j. After filling R^j, they are appended to the set of row indices R. As shown in step 18, C^j is filled with d_i^k DOF of node $k \in S^j$, set of neighbor nodes. After filling the individual C^j for node j, they are appended (nd) times to the set of column indices C. Similar to R, $(nng \times nd^2)$ entries are stored in C^j for node j. D^j, S^j, R^j, C^j, R, and C for node j are shown in figure 5 for a 2D domain with 2 DOFs per node.

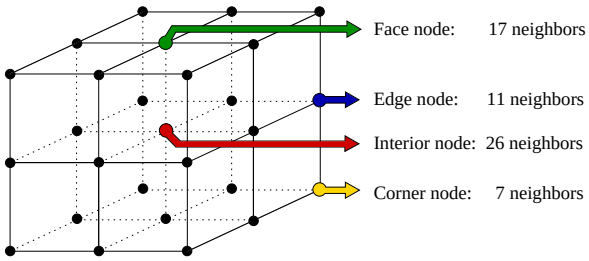

Fig. 4. Neighbors of different nodes in the mesh based on location

4.1.2 The Numeric Kernel: *Element-by-Element* Implementation

After the locations of the non-zeroes in the global matrix are determined using the symbolic kernel, the numeric kernel is launched to store the values of the non-zero entries into the sparse storage format based on the indices stored in (R,C).

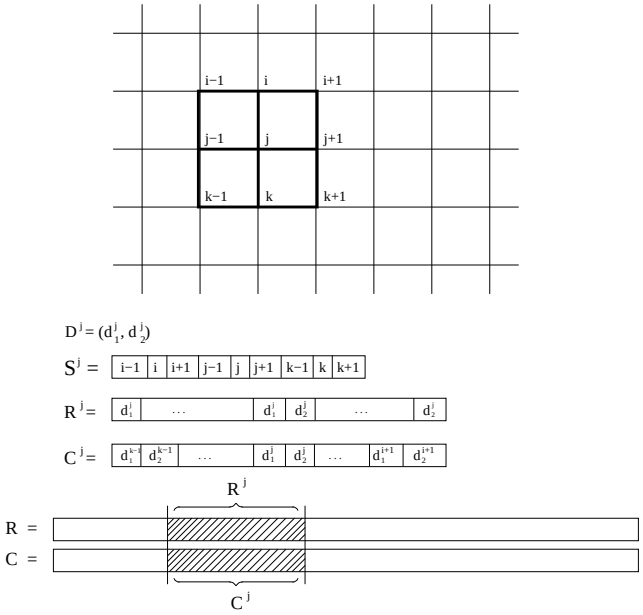

Fig. 5. D^j, S^j, R^j, C^j, R, and C for node j are shown for a 2D domain with 2 DOFs per node

Algorithm 3 Symbolic kernel for indices of non-zero entries

1: **INPUT:** N: total number of nodes; nd: DOFs per node; d_i^j: i^{th} global DOF of node j; S^j: Set containing neighboring nodes of node j; nng: Number of neighboring nodes of a node j; n_k^j: global node number of node k;
2: **OUTPUT:** R : Set of row indices; C : Set of column indices;
3: **for** $j \leftarrow 1 : N$ **do**
4: $D^j = (d_1^j, \ldots, d_{nd}^j)$; ▷ Pre-processing
5: $S^j = (n_1^j, \ldots, n_j^j, \ldots, n_{nng}^j)$;
6: **end for**
7: Copy D^j and S^j to the global memory of GPU
8: **for** $\forall j \in N$ **do** ▷ Kernel launch with a grid of N number of threads
9: $R^j = \emptyset, C^j = \emptyset$;
10: **for** $i \leftarrow 1 : nd$ **do**
11: **for** $k \leftarrow 1 : (nng \times nd)$ **do**
12: $R^j = R^j \cup d_i^j$; ▷ Filling individual R^j in parallel
13: **end for**
14: **end for**
15: $R = R \cup R^j$; ▷ Appending R^j to the row index set R
16: **for** $k \leftarrow 1 : nng$ **do**
17: **for** $i \leftarrow nd$ **do**
18: $C^j = C^j \cup d_i^k, k \in S^j$; ▷ Filling individual C^j in parallel
19: **end for**
20: **end for**
21: **for** $i \leftarrow nd$ **do** ▷ Copying nd times
22: $C = C \cup C^j$; ▷ Appending C^j to C
23: **end for**
24: **end for**

This kernel assembles the entries in an element-by-element manner. Race condition becomes an issue here because of the possibility of more than one threads reading or writing the same memory location simultaneously. This is due to sharing nodes among neighboring elements. Two different implementations for countering this issue are presented in Section 4.4.

Since assembly is done in an element-wise manner, every 8-noded hexahedron element can have $8^2 = 64$ *node-to-node* connections, which are shown in figure 6. We refer to a link between any two node in an element as *node-to-node* connection. A connection can be line-type or point-type depending on whether two different nodes are chosen to make the connection. For example, in figure 6, the connection $(c-h)$ represents a line, whereas, the connection $(a-a)$ represent a point. Every node-to-node connection writes a total of DOF^2 number of non-zeroes into the sparse global stiffness matrix.

In algorithm 4, the steps of computing the values of non-zero entries are shown. The algorithm uses the row (R) and column (C) indices from the symbolic kernel. Unlike the symbolic kernel, the computation is performed in an element-by-element manner as shown in step 3. In step 6, the target index (t) for the first DOF d_1^k is searched within each C^i for every connection using connectivity of element j ($connect[j,ne]$). Only the first DOF (d_1^k) of node $k \in connect[j,nnodes]$ needs to be searched. Other DOF can be determined from the storage sequence used for the sparse format in algorithm 3. In step 11, non-zero entries of the global stiffness matrix are written into the sparse format. In the following sections, three existing formats in addition to two proposed formats are analyzed by modifying their corresponding assembly strategies.

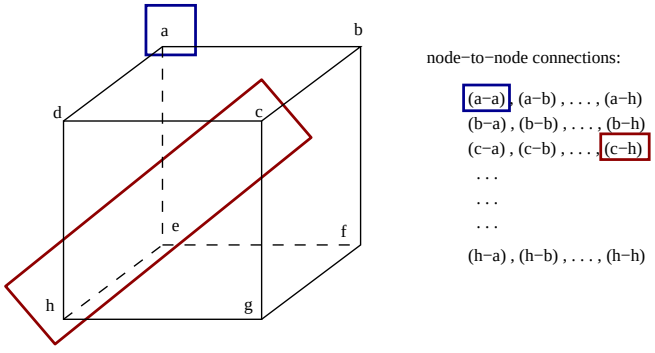

Fig. 6. Node-by-node assembly in the numeric kernel.

4.2 Assembly into Standard Sparse Storage Formats

The symbolic and numeric kernel are described in algorithms 3 and 4 in a generic fashion. These can easily be used with any sparse format with little modifications. In essence, both of the algorithms are presented for the COO format. Therefore, the set of row indices (R) and the set of column indices (C) are the same length as the total no. of non-zero

Algorithm 4 Numeric kernel for values of non-zero entries

1: **INPUT:** N_e: total elements; nd: DOFs per node; ne: nodes per element; $connect[N_e, ne]$: Connectivity Matrix; R, C;
2: **OUTPUT:** Values of global stiffness matrix K
3: **for** $\forall j \in N_e$ **do** ▷ Each element j gets one thread
4: **for** $i \leftarrow\mid connect[j:] \mid$ **do** ▷ connect[j:]: Set of all nodes of element j
5: **for** $k \leftarrow\mid connect[j:] \mid$ **do**
6: Bisection search target index (t) for d_1^k in the set $C^i \in C$ ▷ d_1^k is the first DOF of node k
7: **for** $m \leftarrow nd$ **do** ▷ For assembling K^e into K
8: $r_i = t + nd \times nng \times (m-1)$; ▷ r_i: row index
9: **for** $n \leftarrow nd$ **do**
10: $c_i = t + nd \times nng \times (m-1) + (n-1)$; ▷ c_i: column index
11: $K[R(r_i), C(c_i)] += K^e[m,n]$ ▷ Assembly of non-zero entry via *i-to-j* connection
12: **end for**
13: **end for**
14: **end for**
15: **end for**
16: **end for**

entries in the final matrix. The pattern used for storage of row and column indices in (R,C) remains the same as shown in figure 1. Therefore, algorithm 3 is executed to get (R,C) for storing row and column indices of COO. Next, algorithm 4 is executed to obtain the `Value` array of COO. Step 11 of algo 4 is updated as

$$\texttt{Value}[c_i] += K^e[m,n]$$

Both algorithms 3 and 4 remain same for the CSR format as shown for the COO format. However, at step 12 of algorithm 3, instead of storing row indices (R), row offsets are stored. The storage of non-zero entries is identical to the COO format.

Since the ELL sparse format (referring figure 3) has a column-major ordering, all accesses to the global memory are coalesced. Due to the structured nature of this format, the need for row indices set (R) is removed. Therefore, the steps 10 through 15 of algorithm 3 are no longer required. This results in a reduction of the shared memory and register requirements by approximately 30%. Due to this reason, a higher performance is observed later in Section 5.

As shown in figure 3, the ELL sparse storage format uses one matrix to store the column indices and another matrix for the values of the non-zeroes. Instead of two dimensional matrix in the traditional form, one dimensional arrays are used to store the indices and values on the GPU. The length of C^j for every j in step 18 of algorithm 3 is equal to length of the row of the global stiffness matrix having maximum number of entries. The value of this length can be precomputed using the connectivity matrix ($connect[N_e, ne]$) for the given prob-

lem. It can be noted in figure 3 that the rows, which have less number of non-zero entries, are filled with '$*$'. Similar approach is used for C^i in which DOF of all neighboring nodes are stored. Moreover, step 22 and the corresponding loop are no longer required and can be removed.

In algorithm 4, the step 11 is updated for storing non-zero entries as

$$\texttt{Value}[c_i] += K^e[m,n]$$

Rest of the algorithm 4 is the same for both ELL and COO.

4.3 Assembly using Modified Sparse Storage Formats

We present two new modified sparse storage formats. The primary is to further reduce storage requirement by exploiting specific properties of the global stiffness matrix. As mentioned before, in an element-by-element assembly, the assembly is performed by *node-to-node* connections of each element as shown in figure 4. Each of these connections writes a 3×3 (DOF $\times$ DOF in the generalized case) dense matrix into the global stiffness matrix. While all of these nine non-zero entries need to be stored, all nine indices need not to be stored explicitly. In other words, assembly is always performed in blocks of DOF^2 entries, which stay adjacent to each other even after assembly. Only one index could be stored for all the DOF^2 non-zero entries. For further operations (for example SpMV) on these formats, either modified strategies need to be devised, or these can be converted to any standard sparse format for further processing. The proposed assembly strategy is customized in the context of these two formats. Details of these storage formats as well as the strategy of assembling into them are discussed in the following sections.

4.3.1 COO Modified (COOM) Sparse Storage Format

The proposed COOM format is made up of three one-dimensional arrays same as the COO storage format. The size for storing row indices in (Φ) set is now reduced by keeping the first DOF of every node i as shown in step 10 of algorithm 5. It means that steps 10 and 15 of algorithm 3 are removed for the COOM format. With this modification, the overall size of Φ for the COOM format is now I, instead of $(\theta \times ndof^2 \times I)$ for the COO format. For the column indices, the first DOF for all neighboring nodes to node i are stored as shown at step 12 of algorithm 5. This also reduces size of Γ to $(\theta \times I)$, instead of $(\theta \times ndof^2 \times I)$ for the COO format.

The `Value` array for the COOM format has the size equal to the total number of non-zero entries. Algorithm 4 remains the same for the COOM format. It can be seen at step 6 of the same algorithm that searching of index (p) is reduced in the range of (θ) for $\Gamma^i \in \Gamma$. The assembly through `Value` array remains the same as the COO format as described in Section 4.2. In the present study, we did not include a modified CSR format separately, because it would simply be identical to the COOM format. This is because the CSR format already uses a similar concept of row offsets and indices as in the COOM format.

Algorithm 5 Symbolic kernel for COOM format

1: **INPUT:** N: total number of nodes; nd: DOFs per node; d_i^j: i^{th} global DOF of node j; S^j: Set containing neighboring nodes of node j; nng: Number of neighboring nodes of a node j; n_k^j: global node number of node k;
2: **OUTPUT:** R : Set of row indices; C : Set of column indices;
3: **for** $j \leftarrow 1 : N$ **do**
4: $\quad D^j = (d_1^j, \ldots, d_{nd}^j)$; ▷ Pre-processing
5: $\quad S^j = (n_1^j, \ldots, n_j^j, \ldots, n_{nng}^j)$;
6: **end for**
7: Copy D^j and S^j to the global memory of GPU
8: **for** $\forall j \in N$ **do** ▷ Kernel launch with a grid of N number of threads
9: $\quad R^j = \emptyset, C^j = \emptyset$;
10: $\quad R^j = R^j \cup d_1^j$; ▷ Filling individual R^j in parallel
11: $\quad$ **for** $k \leftarrow 1 : nng$ **do**
12: $\quad\quad C^j = C^j \cup d_1^k, k \in S^j$; ▷ Filling individual C^j in parallel
13: $\quad$ **end for**
14: $\quad C = C \cup C^j$
15: **end for**

4.3.2 ELL Modified (ELLM) Sparse Storage Format

As described in Section 4.2 that the indices and non-zero entries are stored in two matrices as per the format shown in figure 3. However, we store the column indices in a one-dimensional array (Γ) and non-zero entries in the `Value` array, similar to the ELL format. Since the row indices are not required, step 10 can be removed for the ELLM format for the symbolic kernel in algorithm 5. The Γ set remains the same as shown in algorithm 5, which reduces its size for the ELLM format to $(\theta \times I)$ as compared to $(\theta \times ndof \times I)$ with the ELL format. The `Value` array for the ELLM format stores non-zero entries as explained in algorithm 4. For the ELLM format also, searching range for index (p) is reduced at step 6 of the same algorithm. The assembly through `Value` array remains the same as the ELL format described in Section 4.2. In figures 7 and 8 the space required in bytes to store the global stiffness matrix are plotted against node numbers and DOF per node for different storage formats. These storage requirements are calculated using expressions derived for both the standard and the modified storage formats as presented in table 1. In the table, n_x, n_y and n_z represent the node numbers in the x, y and z directions respectively for a cuboid domain using hexahedron elements. The figure shows similar storage requirements for the ELL and CSR formats. Both the proposed formats (COOM and ELLM) take significantly less amount of storage space compared to the other three formats.

Table 1. Storage space requirement for different sparse formats.

Format	Required Sapce in bytes
COO	$1296(n_x+n_y+n_z)-1944(n_xn_y+n_yn_z+n_zn_x)$ $+2916n_xn_yn_z-864$
CSR	$864(n_x+n_y+n_z)-1296(n_xn_y+n_yn_z+n_zn_x)$ $+1956n_xn_yn_z-576$
ELL	$1944n_xn_yn_z$
COOM	$576(n_x+n_y+n_z)-864(n_xn_y+n_yn_z+n_zn_x)$ $+1300n_xn_yn_z-384$
ELLM	$1080n_xn_yn_z$

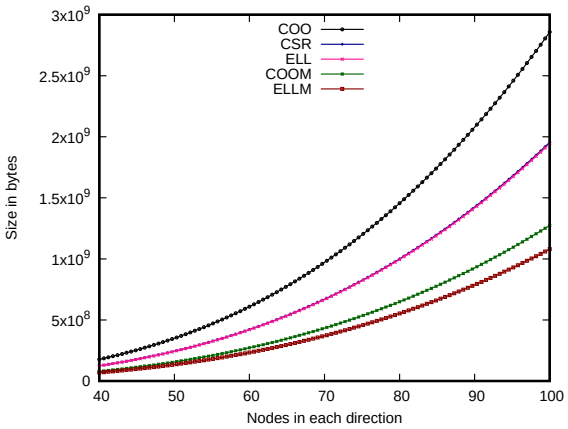

Fig. 7. Storage requirement for different sparse storage formats with increasing number of nodes.

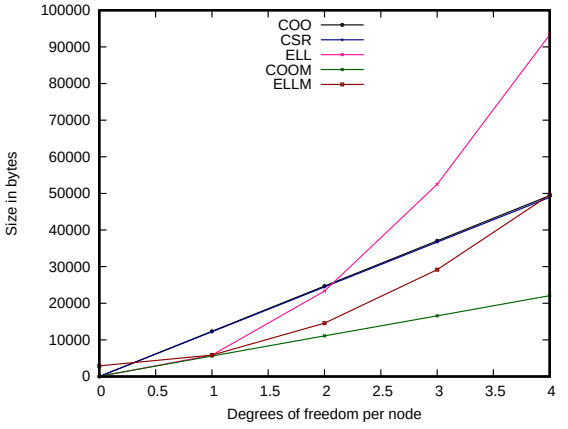

Fig. 8. Storage requirement for different sparse storage formats with increasing DOF/node.

4.4 Race Condition

A race condition or data race in computation occurs when the output of a process becomes dependent on the sequence in which two simultaneous threads access a memory location. The GPU-based implementation of FEA assembly is susceptible to this issue when more than one threads write to the same memory location. In the symbolic kernel, each node being assigned to a single unique thread, the possibility of sharing of nodes among any thread is alleviated. This nodal independence removes any chance of a race condition. This, however, is not true for the numeric kernel, where each element is assigned exactly to one thread. Due to the fact that all the elements in the mesh have shared nodes, the possibility exists for two or more threads writing to the same memory location at the same time. Therefore, we implement atomic operations and element coloring methods to counter the race condition.

When an atomic operation such as `atomicAdd` or `atomicSub` is invoked, it locks the concerned memory location and waits until the requested operation is completed ([34]). This is usually avoided due to the serialization of atomic threads hampering the overall performance. The current work keeps the threads waiting for an atomic lock to a bare minimum. For a dense enough mesh, the amount of serialization caused by an atomic operation becomes smaller and smaller and the performance degradation becomes negligible. In the second implementation, we color the mesh elements with different colors in a way that an element is only allowed to share nodes with elements of different colors. The element coloring method is shown in figure 9 for a structured mesh. After the coloring is done, separate kernels for each of the colors are invoked in a serial manner as shown in algorithm 6.

Algorithm 6 Assembly using Colors

```
1: n ← Number of colors
2: *E[n] ← Set of elements in a color
3: procedure KERNEL CALL USING COLORING
4:     for i ← 1 : n do
5:         numThreads ← size(*E[i]);
6:         AssemblyKernel <<< numThreads >>> (n)
7:     end for
8: end procedure
```

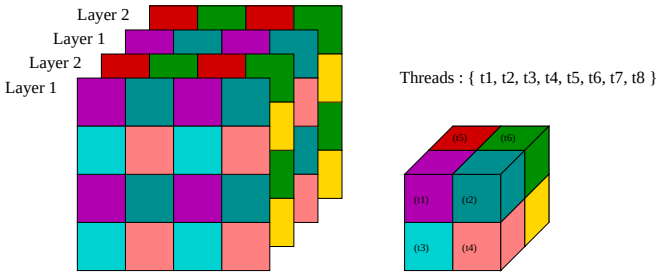

Fig. 9. Coloring Scheme for race condition

5 Results and Discussion

5.1 Test Problem and Hardware Details

The proposed assembly strategy of dividing the task into symbolic and numeric kernels is now tested using the standard problem of the cantilever beam. The beam is discretized with eight-noded hexahedron elements with an end load, whose results are already known to us. The size of the beam is decided by the number of nodes in x, y, and z directions. The material used is isotropic and homogeneous. For performance analysis, the CPU version of the code is run on an Intel Xeon ES1650 Sandy Bridge with 6 cores clocked at 3.2 GHz. The GPU code is run on a Tesla K40c with peak memory bandwidth of 288 gigabytes per second. The GPU has 12 gigabytes of global memory with 2880 cores.

5.2 Performance Analysis

For the performance analysis of the proposed assembly strategies on GPU, it is compared with the *SharedNZ* implementation of study by Cecka et al. [6] on GPU and serial implementation of algorithm 2 of `addto` method. The *SharedNZ* algorithm is implemented using CUDA in which the kernel and memory utilization remain the same as given by Cecka et al. [6]. The serial code is implemented using the C-programming language. Figure 10 shows the execution time of the proposed assembly strategy using different storage formats with the coloring method. The execution time of the *SharedNZ* implementation is shown using the bar and that of the others are shown using the curves. It can be seen from the figure that all implementations using the proposed assembly strategy require less execution time as compared to the *SharedNZ* implementation. Moreover, the modified storage formats with the proposed assembly strategy perform significantly better than the standard counterparts. This reduced execution time can be attributed to the search-span reduction while assembly through efficient distribution of the workload, reduced memory footprint and write operations. Figure 11 shows the execution time of the proposed assembly strategy using atomics with different sparse storage formats. It can be seen from the figure that apart from the COO and CSR formats, all the storage formats including the modified ones outperform the *SharedNZ* implementation. Also, the modified formats outperform the standard formats using atomics as well.

Figure 12 shows the speedup comparison of all the implementations (five sparse storage formats, each for atomics and coloring method) compared to the *sharedNZ* implementation. Clusters of histograms are plotted each for five different mesh sizes. In each of these clusters, the colored bars represent the coloring-based implementations and the patterned bars represent the atomics-based implementations. As can be expected from figures 10 and 11, the speedup values for the coloring-based implementations are significantly higher than the atomics-based implementations. The highest speedup is obtained for the ELLM and COOM formats for all mesh sizes using the coloring method. For the coloring-based modified formats (COOM-coloring and ELLM-coloring) a downward trend in the speedup can be observed for larger

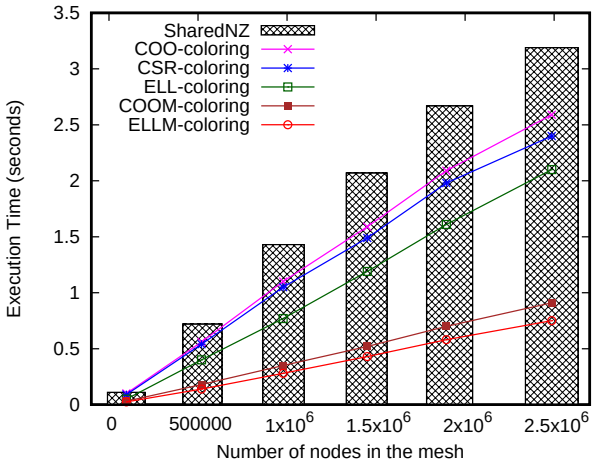

Fig. 10. Comparison of execution time with different meshes having different number of nodes for coloring-based implementations.

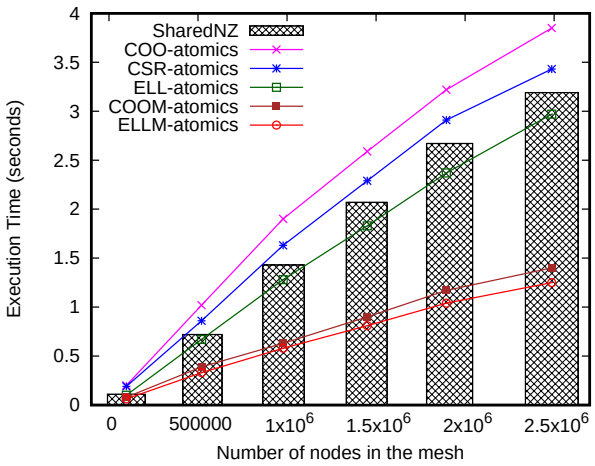

Fig. 11. Comparison of execution time with different meshes having different number of nodes for atomics-based implementations.

mesh sizes. On the other hand, a constant trend can be observed at higher mesh sizes for the atomic-based implementations. The reason behind this observation is that the number of available warps increases for larger mesh sizes. This ensures that despite the number of threads waiting for a lock created by an atomic operation to release, the GPU occupancy does not go down. This, in turn, results in higher relative performance for larger mesh sizes for atomics-based implementations. Figure 13 shows the speedups of all the implementations in comparison to the serial implementation of algorithm 2 on CPU. Unlike the trend shown in figure 12, the modified (COOM and ELLM) and existing storage formats show an increasing trend in the speedups when compared to the CPU version. This is due to the fact that the CPU version consumes increasing amounts of time for larger mesh sizes.

Figure 14 shows the GFLOP/s for all the implementations for two different mesh sizes. Similar to the previous plots, the GFLOP/s values for the coloring-based implementations are significantly higher than the atomics-based implementations. This is because of the fact that although the total

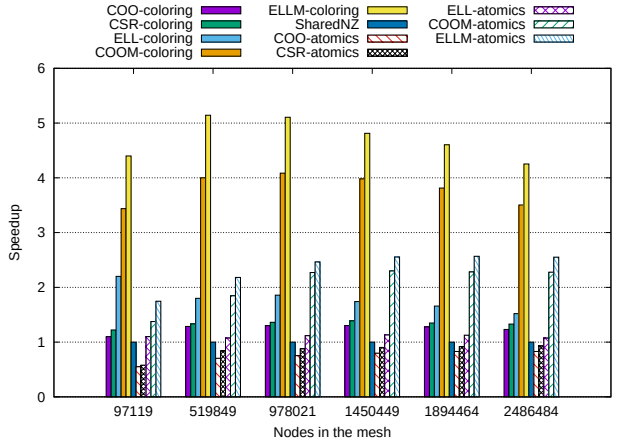

Fig. 12. Obtained speedup for all formats in comparison to SharedNZ implementation on GPU.

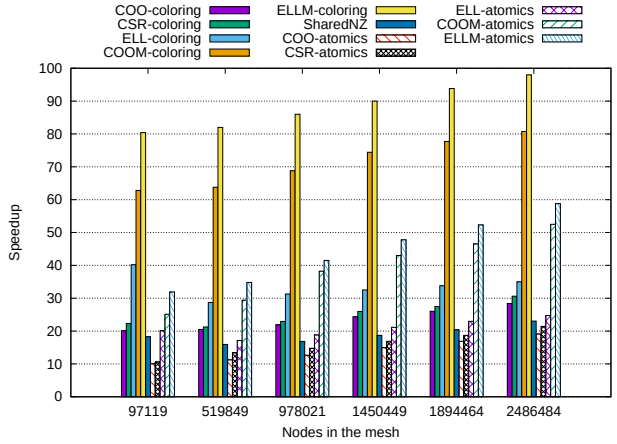

Fig. 13. Obtained speedup for all storage formats in comparison to CPU implementation.

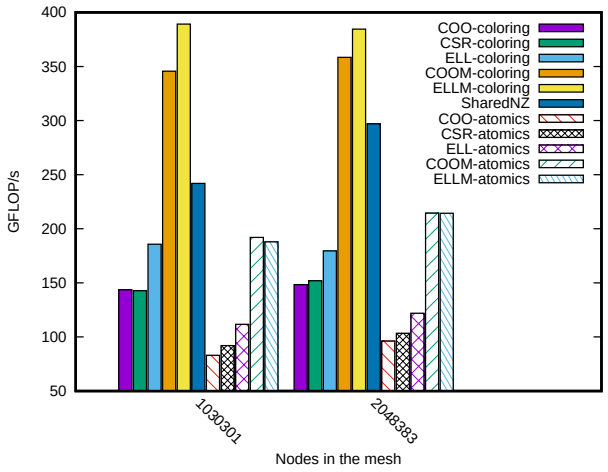

Fig. 14. Obtained GFLOPS for all implementations.

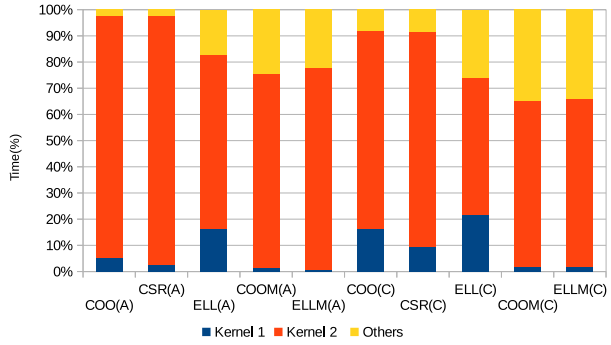

Fig. 15. Execution time percentage of symbolic and numeric kernels with different sparse formats

number of floating point operations is similar for the atomics and coloring-based implementations, the GFLOP/s count becomes low in comparison to the coloring-based implementations due to the total execution time being higher in case of atomics-based implementations. The highest performance is observed for the coloring-based implementations with the COOM and ELLM formats. The COO and CSR formats are seen to have lower values, especially for the atomics-based implementations. Figure 15 shows the percentage of time required by different parts of the application. The COOM and ELLM formats with element coloring outperform all other implementations. For all implementations, the numeric kernel is seen to be the most time consuming part of the entire assembly operation, whereas the symbolic kernel takes only a small fraction of the total time.

6 Conclusion

We presented a number of implementations for performing FEA assembly on GPUs. A strategy was developed for efficient implementation of the `addto` assembly algorithm on GPU with three standard and two proposed sparse storage formats, while decomposing the task into two parts. The strategy was found to perform significantly better when compared to the optimized CPU and GPU implementations for all the sparse storage formats. For countering the race condition, the coloring method outperformed atomics-based method for all sparse storage formats. Both the modified sparse storage formats produced considerably better results than their standard counterparts. Assembly into the ELLM format required the least amount of time compared to other formats. This can be attributed to low usage of registers and shared memory by the kernels. Assembly into the COOM format, on the other hand, required the smallest amount of storage space compared to all other formats for the same mesh size. Since, for structured meshes the neighbor information is readily available using index mappings, efficient ordering can be explored for handling race condition along with the coloring and atomics-based methods presented in this paper. Although only regular meshes have been used in this paper, the modified sparse formats are independent of the type of mesh and can be used with any type of meshes. The kernel division strategy, however needs to be adopted for implementation on an irregular mesh. Since the neighbor information is not readily available for unstructured grids, preprocessing of the mesh connectivity can be performed and

the resulting neighbor information can be utilized for implementing the same kernel division strategy. The GPU used in this paper allowed us to solve a problem with a maximum node number of approximately 4 million. In case of larger mesh sizes, we can use domain decomposition methods using a graph partitioning library such as METIS ([35]) in future. Furthermore, testing of the proposed methodology using a more modern GPU such as the V100 or A100 with the increased number of cores and superior performance can be performed for a more complete analysis.

Acknowledgment

This work was supported by SERB, Department of Science and Technology (DST), India (grant number SB/FTP/ETA-28/2013) and IIT Guwahati (grant number SG/ME/DS/P/01). We would like to express our gratitude to NVIDIA for donating Tesla K40c GPU used in this work.

References

[1] Zienkiewicz, O. C., Taylor, R. L., and Lee, R., 1977. *The finite element method*, Vol. 3. McGraw hill London.

[2] Ram, L., and Sharma, D., 2017. "Evolutionary and gpu computing for topology optimization of structures". *Swarm and Evolutionary Computation*, **35**, pp. 1–13.

[3] Ratnakar, S. K., Sanfui, S., and Sharma, D., 2020. "Gpu based topology optimization using matrix-free conjugate gradient finite element solver with customized nodal connectivity storage". In 2nd International Conference on Future Learning Aspects of Mechanical Engineering (FLAME - 2020), Amity University.

[4] Ratnakar, S. K., Sanfui, S., and Sharma, D., 2020. "Simp-based structural topology optimization using unstructured mesh on gpu". In 2nd International Conference on Future Learning Aspects of Mechanical Engineering (FLAME - 2020), Amity University.

[5] Georgescu, S., Chow, P., and Okuda, H., 2013. "GPU acceleration for FEM-based structural analysis". *Archives of Computational Methods in Engineering*, **20**(2), pp. 111–121.

[6] Cecka, C., Lew, A. J., and Darve, E., 2011. "Assembly of finite element methods on graphics processors". *International Journal for Numerical Methods in Engineering*, **85**(5), pp. 640–669.

[7] Maciol, P., Plaszewski, P., and Banaś, K., 2010. "3D finite element numerical integration on GPUs". *Procedia Computer Science*, **1**(1), pp. 1093 – 1100. ICCS 2010.

[8] Lei, J., Li, D.-l., Zhou, Y.-l., and Liu, W., 2019. "Optimization and acceleration of flow simulations for cfd on cpu/gpu architecture". *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, **41**(7), p. 290.

[9] Liu, W., Schmidt, B., Voss, G., and Müller-Wittig, W., 2008. "Accelerating molecular dynamics simulations using graphics processing units with CUDA". *Computer Physics Communications*, **179**(9), pp. 634 – 641.

[10] Komatitsch, D., Michéa, D., and Erlebacher, G., 2009. "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA". *Journal of Parallel and Distributed Computing*, **69**(5), pp. 451 – 460.

[11] Fu, Z., Lewis, T. J., Kirby, R. M., and Whitaker, R. T., 2014. "Architecting the finite element method pipeline for the gpu". *Journal of Computational and Applied Mathematics*, **257**, pp. 195 – 211.

[12] Reguly, I. Z., and Giles, M. B., 2015. "Finite element algorithms and data structures on graphical processing units". *International Journal of Parallel Programming*, **43**(2), pp. 203–239.

[13] Banaś, K., Kruzel, F., and Bielański, J., 2016. "Finite element numerical integration for first order approximations on multi- and many-core architectures". *Computer Methods in Applied Mechanics and Engineering*, **305**, pp. 827 – 848.

[14] Knepley, M. G., and Terrel, A. R., 2011. "Finite element integration on gpus". *CoRR*, **abs/1103.0066**.

[15] Bolz, J., Farmer, I., Grinspun, E., and Schröoder, P., 2003. "Sparse matrix solvers on the GPU: conjugate gradients and multigrid". *ACM Trans. Graph.*, **22**(3), July, pp. 917–924.

[16] Rodríguez-Navarro, J., and Susín Sánchez, A., 2006. "Non structured meshes for Cloth GPU simulation using FEM". In Vriphys: 3rd Workshop in Virtual Realitiy, Interactions, and Physical Simulation, C. Mendoza and I. Navazo, eds., The Eurographics Association.

[17] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M., 2012. "Finite element matrix generation on a GPU". *Progress In Electromagnetics Research*, **128**, pp. 249–265.

[18] Markall, G., Slemmer, A., Ham, D., Kelly, P., Cantwell, C., and Sherwin, S., 2013. "Finite element assembly strategies on multi-core and many-core architectures". *International Journal for Numerical Methods in Fluids*, **71**(1), pp. 80–97.

[19] Kiss, I., Gyimothy, S., Badics, Z., and Pavo, J., 2012. "Parallel realization of the element-by-element FEM technique by CUDA". *IEEE Transactions on Magnetics*, **48**(2), Feb, pp. 507–510.

[20] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M., 2013. "Generation of large finite-element matrices on multiple graphics processors". *International Journal for Numerical Methods in Engineering*, **94**(2), pp. 204–220.

[21] Carrion, R., Mesquita, E., and Ansoni, J. L., 2015. "Dynamic response of a frame-foundation-soil system: a coupled bem–fem procedure and a gpu implementation". *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, **37**(4), pp. 1055–1063.

[22] Dinh, Q., and Marechal, Y., 2016. "Toward real-time finite-element simulation on gpu". *IEEE Transactions on Magnetics*, **52**(3), March, pp. 1–4.

[23] Sanfui, S., and 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), AMIAMS, ed., pp. 1–9.

[24] Zayer, R., Steinberger, M., and Seidel, H., 2017. "Sparse matrix assembly on the gpu through multiplication patterns". In 2017 IEEE High Performance Extreme Computing Conference (HPEC), HPEC, ed., pp. 1–8.

[25] Kiran, U., Sharma, D., and Gautam, S. S., 2018. "Gpu-warp based finite element matrices generation and assembly using coloring method". *Journal of Computational Design and Engineering*.

[26] Gribanov, I., Taylor, R., and Sarracino, R., 2018. "Parallel implementation of implicit finite element model with cohesive zones and collision response using cuda". *International Journal for Numerical Methods in Engineering*, **115**(7), pp. 771–790.

[27] Sanfui, S., and 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), G. X. G. R. Liu, Fangsen Cui, ed., Vol. 6, ScienTech Publisher, pp. 641–651.

[28] Kiran, U., Gautam, S. S., and Sharma, D., 2020. "Gpu-based matrix-free finite element solver exploiting symmetry of elemental matrices". *Computing*, **102**, p. 19411965.

[29] Sanfui, S., and Sharma, D., 2020. "A three-stage graphics processing unit-based finite element analyses matrix generation strategy for unstructured meshes". *International Journal for Numerical Methods in Engineering*, **121**(17), p. 38243848.

[30] Wong, J., Kuhl, E., and Darve, E., 2015. "A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems". *International Journal for Numerical Methods in Engineering*, **102**(12), pp. 1784–1814.

[31] Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Basermann, A., and Bishop, A. R., 2012. "Sparse matrix-vector multiplication on gpgpu clusters: A new storage format and a scalable implementation". In 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, M. Li, ed., pp. 1696–1702.

[32] Choi, J. W., Singh, A., and Vuduc, R. W., 2010. "Model-driven autotuning of sparse matrix-vector multiply on gpus". In ACM sigplan notices, ACM, ed., Vol. 45, pp. 115–126.

[33] Ramrez-Gil, F. J., Silva, E. C. N., and Montealegre-Rubio, W., 2016. "Topology optimization design of 3d electrothermomechanical actuators by using gpu as a co-processor". *Computer Methods in Applied Mechanics and Engineering*, **302**, pp. 44 – 69.

[34] Kirk, D. B., and Wen-mei, W. H., 2012. *Programming massively parallel processors: a hands-on approach*. Newnes.

[35] Karypis, G., and Kumar, V., 1998. "A fast and high quality multilevel scheme for partitioning irregular graphs". *SIAM Journal on Scientific Computing*, **20**(1), Dec., pp. 359–392.