Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

A Three-Stage GPU-based FEA Matrix Generation Strategy for Unstructured Meshes

Subhajit Sanfui | Deepak Sharma

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

Correspondence
Deepak Sharma, Department of Mechanical Engineering, Indian Institute of Technology Guwahati, Guwahati, Assam 781039, India.
Email: dsharma@iitg.ac.in

Summary

With the development of parallel computing architectures, larger and more complex Finite Element Analyses (FEA) are being performed with higher accuracy and smaller execution times. Graphics Processing Units (GPUs) are one of the major contributors of this computational breakthrough. This work presents a three-stage GPU-based FEA matrix generation strategy with the key idea of decoupling the computation of global matrix indices and values by use of a novel data structure referred to as the neighbor matrix. The first stage computes the neighbor matrix on the GPU based on the unstructured mesh. Using this neighbor matrix, the indices and values of the global matrix are computed separately in the second and third stages. The neighbor matrix is computed for three different element types. Two versions for performing numerical integration and assembly in the same or separate kernels are implemented and simulations are run for different mesh sizes having up to 3 million degrees of freedom on a single GPU. Comparison to GPU-based parallel implementation from the literature reveals speedup ranging from 4× to 6× for the proposed workload division strategy. Furthermore, the same kernel implementation is found to outperform the separate kernel implementation by 70% to 150% for different element types.

KEYWORDS:
Finite Element Analysis, Matrix Generation, Unstructured Mesh, GPU, Kernel Division

1 Introduction

A large array of physical phenomena such as elasticity, fluid dynamics, sound, heat, electrostatics or quantum mechanics can be described accurately by partial differential equations (PDEs). These supposedly distinct phenomena are formalized using PDEs and solved to get the response of a system by using a numerical method. Finite Element Method (FEM)[1] is one of the most popular numerical methods for solutions to PDEs. Several inherent advantages over other methods have made it an elemental part of a broad array of specializations both in academia and in industries like automotive, aviation and construction. It is also extensively utilized in several engineering specializations like mechanical, civil and electrical engineering.

The practice of applying concepts of FEM for solving practical problems is named as "Finite Element Analysis (FEA)", which is made up of three basic steps, that are, the *Pre-processor step*, the *Solver step* and the *Post-processor step*. The problem domain is established by defining the material constants, element order and type, and mesh information during the *Pre-processor step*. This is followed by the *Solver step*, where the elemental matrices are computed and assembled into a global stiffness matrix. This matrix, after application of proper boundary conditions, gives a set of algebraic equations. This system of equations upon

solving yields the response of the physical system under the applied loading conditions. The final step is the *Post-processor step*, where the results obtained in the *solver step* are utilized to gain more insight about the problem like stresses and strains. Plotting and visualization are also done during this stage.

It is reported in the literature[2,3,4] that several steps of FEA such as the elemental matrix generation, assembly into global stiffness matrix and solution of the linear system are time-consuming in nature for large and complex domains. Although a single FE analysis, in itself, is often capable to be perfectly handled by traditional CPU implementations, challenges arise in cases involving complex geometries and ultra-fine meshes, or cases where the FEA is coupled with another computational algorithm such as in topology optimization[5]. One common strategy for reducing time is to perform FEA computations in parallel by exploiting the data-parallel nature of the FEA algorithm[2]. Due to the single-instruction-multiple-data (SIMD) structure of FEA, it is typically suited for shared-memory parallel architectures such as GPUs. As a result, a lot of research work has focused on GPU-based parallel implementations of different stages of FEA over the last decade. Starting with computing elemental stiffness matrices, it is often computationally expensive due to repeated calculations for a high number of elements or high order of elements[6]. Owing to this fact, several studies focused explicitly on the numerical integration aspect[7,8]. This generation of elemental data can be done independently for all the elements, which makes this stage *embarassingly parallel* as reported by several researchers[9,10]. Another challenge is the assembly of the elemental stiffness matrices into a global stiffness matrix, which can also become computationally expensive[2] for a large mesh or in case of non-linear FEA where assembly needs to be performed repeatedly. After the assembly is performed, the implementation of boundary conditions and solution of linear equations have also been implemented on GPU[11,9]. Some implementations in the literature have investigated matrix-free methods for GPU-based FEA[12,13]. These methods work on the key principle of reducing memory access and storage by introducing redundant computation. Unlike the elemental stiffness matrix generation stage, the other stages of FEA cannot be broken into small independent chunks of computation due to dependencies among the nodes and elements. This calls for more sophisticated implementations from the programmers' side. In general, the *data-parallel* and *throughput-intensive* nature of different stages of FEA makes it a highly suitable candidate for GPU implementation, especially for large-scale problems. However, the traditional algorithms, being originally developed for traditional sequential hardware, are often rendered *infeasible* on the massively parallel architectures. New and innovative approaches that are often radically different are required to reflect the target architecture[14,11,15] and to extract a good performance.

Among very early studies on GPU-based implementation of FEA matrix generation, Bolz et al.[16], RodrÃŋguez-Navarro and SusÃŋn SÃnchez[17] used simplified parallel implementations where expressions were derived for each non-zero (NZ) entry in the global stiffness matrix separately and the computations were performed in parallel on the GPU. Although this implementation provided some performance gain, deriving such expressions was not possible for many real-world applications. After the launch of Compute Unified Device Architecture (CUDA) from NVIDIA in 2007, more sophisticated and generalized implementations were being reported with significant performance gains. Komatitsch et al.[6]implemented numerical simulation of seismic wave propagation caused by earthquakes on an NVIDIA GPU. In this implementation, each thread block was assigned to one 125-noded element of the finite element mesh. The inherent data race was countered by coloring the elements and computing the colors in sequence. Similar coloring strategies were also reported by Cecka et al.[3] targeting assembly on GPU, where a thorough study of different ways to perform an assembly for unstructured 2D mesh was presented. An elemental subroutine was used for calculating the elemental matrices, where each thread was designated different tasks for different implementation strategies. The data race associated with assembly was handled by allocating each NZ entry to one compute thread or by using a coloring approach identical to the implementation by Komatitsch et al.[6]. The coloring approach was found to be under-performing as a result of excessive accesses to the global memory of GPU. For lower order elements, the authors recommended usage of shared memory to store the elemental data and assigning each thread to compute an NZ entry of the global stiffness matrix. For higher orders, a single thread per element strategy was suggested. Fu et al.[9] discussed acceleration of FEM on the GPU for higher order elements. Again the thread allocation was done by assigning a single thread to one element of the FE mesh. This implementation, however, involved redundant computation because of the elemental stiffness matrices being calculated on the fly and not being stored on the GPU memory. Markall et al.[14] discussed assembly by segmentation into two separate approaches, which were named as the `addto` method and the local matrix approach (LMA). The standard assembly approach followed on the CPU was termed as the `addto` method of assembly. LMA, on the other hand, is a matrix-free method with the key idea of alleviating GPU-related implementational difficulties. The idea of these implementations originated from the methods followed in higher order FEM[18], where similar *scatter and multiply* methods were implemented for multiplication to the global stiffness matrix. A conclusion was drawn that LMA is suitable for GPU implementations whereas `addto` method is more suitable for CPU implementations. The study by Kiss et al.[15] took this method of assembly even another step further where

the elemental matrices were computed whenever required instead of storing on the GPU. This low-storage compute intensive implementation was shown to be better suited to many-core systems for large data sets. However, Fu et al.[9] later demonstrated through comparative studies that using this approach, the results were worse than the traditional assembly into complete global stiffness matrix for 3D grids. The authors presented a novel algorithm for assembly using a patch-based division of the entire mesh that aims to alleviate several difficulties of GPU-based assembly. Dziekonski et al.[19,20] presented an efficient method for generating the global stiffness matrices in FEA in the field of computational electromagnetic using both single and multi-GPU setups. The key idea behind the strategy was to assemble the sparse global stiffness matrix into the coordinate format (COO) in the first step and to transform it into compressed sparse row (CSR) format by removing the duplicate entries during the second step. Among more recent studies, Dinh and Marechal[21] studied a real-time FEM implementation on GPU, where a sorting-based implementation of parallel global assembly was performed. In this implementation, after executing the elemental routine, the data was accumulated into an unsorted COO format with duplicate entries. Following this, the entries were sorted using parallel radix sort and reduced to form the global stiffness matrix. An implementation based on the principle of dividing the GPU assembly with standard sparse formats was presented by Sanfui and Sharma[22]. The authors used structured meshes with brick elements to demonstrate the advantage of workload division at the assembly stage. The implementation divided the assembly operation into a separate symbolic kernel and a numeric kernel. Later, Zayer et al.[23] accelerated assembly of sparse matrices by modifying the assembly stage as a matrix-matrix multiplication with the aim to remove any CPU or GPU-based preprocessing. This approach enabled them to reduce storage and movement of data on the GPU. Among more recent works, Kiran et al.[24] presented a warp-based assembly approach for hexahedral elements in single precision where the numerical integration and assembly were performed in the same kernel. An implicit finite element model with cohesive zones and collision response was accelerated using CUDA by Gribanov et al.[25]. For handling the race condition in assembly, instead of coloring the elements, `atomicAdd` function of CUDA toolkit was used to resolve it at the hardware level. Recently, Sanfui et al.[26] implemented FEA on GPU by utilizing the symmetry of the elemental and global stiffness matrix. Numerical integration and assembly operations were implemented on the GPU to achieve a speedup of approximately 2× over the standard implementation that computed the entire matrix on the GPU. It should be mentioned here that apart from traditional FEM implementations, GPU implementations of matrix generation for isogeometric analysis (IGA) have also been reported in the literature[27,28].

After assembly, proper boundary conditions are applied to the global stiffness matrix, which results in a non-singular system of equations to be solved using either an iterative or a direct method. For matrices of large size, Conjugate Gradient and other Krylov subspace methods are preferred[3].

It is observed from the literature survey that while assembling into a significantly large global stiffness matrix, a lot of the methods discussed in the literature fail to perform. One of the primary reasons is the usage of sparse storage formats that create complications in the assembly operation. This challenge becomes more prominent for assembly into 3D unstructured meshes for the irregular pattern of the NZ entries in the global stiffness matrix. For assembly of an unstructured mesh directly into a specialized sparse storage format on GPUs, two important challenges are observed. The first challenge is to calculate the memory required for the specific sparse storage format beforehand[20,22]. The second challenge is the search operation that is needed to find the location for writing NZ entries[22]. The search operations need to be performed for locating the target location in the specific sparse format. These operations are in general considered undesirable on the GPU due to their effect on the performance of the application by causing branching. It is also observed that the index computation of the entries of the global stiffness matrix depends only on the mesh of the domain and can be separated from the value computation for the NZ entries. The present work aims to counter these challenges in assembly for 3D unstructured mesh by a novel divide-and-conquer implementation that breaks the assembly into three distinct stages. In the first stage called mesh preprocessor, the mesh is processed to generate the neighbor matrix. By use of the neighbor matrix, the storage requirements are computed for the sparse storage format. This matrix also facilitates decoupling the assembly operation into separate index and values computation. In the second stage, the indices of the sparse storage format are calculated. Finally, in the third stage, the values of the NZ entries are calculated. All of the existing literature has performed assembly on the GPU by using the same thread assignment scheme throughout the application[3,6,9]. In the present work, the index and values computations are performed by assigning a thread to one node and one element of the unstructured mesh respectively. The mesh preprocessor is performed as a collection of kernels that incorporate a mixture of per-node and per-element thread assignment. Since all the kernels follow either a one-thread-per-node or one-thread-per-element strategy, there is no sharing of data among the threads. Due to this reason, shared memory has not been used in any implementation. Only the global and local memories are used. It can also be seen from the literature that the implementations and algorithms are designed for specific types of element and thus, cannot be used with different types of element such as TET4, TET10 and HEX20. However, the proposed strategy can easily be modified for different types of elements.

In the literature, the computation of elemental matrices and assembling them into global stiffness matrix have been done either by developing one kernel[9,10,24] or separate kernels[3,22] approaches. The former approach obviates the need of storing matrices in the global memory, and the later approach performs computing and assembly in separate kernels without overburdening individual compute threads. Since these approaches have not been compared in the literature, the present work aims to compare them on unstructured meshes.

The remaining paper is organized in five sections. In Section 2, preliminaries are discussed including details about CUDA and FEM formulations for elasticity problem. In Section 3.1, the mesh preprocessor is discussed followed by Sections 3.2 and 3.3, where the index computation and values computation for NZ entries are discussed respectively. In Section 4, the same and separate kernel implementations are discussed. Section 5 discusses the results and the conclusions are presented in Section 6.

2 Preliminaries

2.1 Graphics Processing Units for Computation

GPUs are single-chip processors which were initially developed for dedicated graphics rendering purposes. Later, researchers started using them for accelerating various research, scientific and analytical applications which were computationally expensive. To make life easier for the GPGPU programmers, platforms like OpenCL and CUDA were introduced that provide more and more control on the GPU hardware. We use CUDA, a parallel programming API from NVIDIA, for the present application. CUDA supplies a programming paradigm that encompasses both the GPU and the CPU simultaneously. The GPU functions are written by using a specific set of extensions to the standard C/C++ language, inside special functions that are referred to as kernels. On invoking, a kernel can generate large grids of threads to parallelize a solution. Furthermore, unlike traditional CPU-based implementations, the programmer is at liberty to make use of the highly efficient memory hierarchy of the GPUs along with several other features to design a sophisticated kernel that can harness the computational power that a modern GPU is able to provide.

2.2 Linear Elasticity Problem: FEM Formulation

A wide variety of physical phenomenon including the response of mechanical systems under specific loadings can be described by PDEs obtained from the basic physics of the problem. These PDEs, however, in most real-life applications, can not be solved analytically owing to factors such as non-linearity and complexity of the domain. Such cases necessitate the use of numerical methods like FEM. By using FEA, the solution of the concerned PDE is obtained by breaking up the problem domain into smaller parts called finite elements as presented in figure 1. Specific functions are derived which are called shape functions to approximate the values of the primary unknown over the entire domain. Elemental contributions for each of the elements are computed, which are assembled thereafter to form the global stiffness matrix. This is called the *Assembly* operation in the context of FEA.

2.2.1 Linear Elasticity

The study of stresses and elastic deformations in solid bodies subjected to prescribed loading conditions is known as Linear Elasticity. The governing PDEs of a linear elasticity problem are usually stated as boundary value problems. The governing equations include[1] the strain-displacement equations, equilibrium relations and constitutive relations given by,

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

Here, b_i are the components of the body forces per unit volume, ρ is the mass density, u_i are the displacements, σ_{ij} are the components of Cauchy stress tensor, ϵ_{kl} are the strains and C_{ijkl} is the material elasticity tensor. In the present work, the material is considered isotropic and homogeneous.

Equation (1) is subjected to the Dirichlet and Neumann boundary conditions given by,

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

where Γ_u and Γ_t are the portions of the boundary Γ, where the displacements and the boundary traction forces are specified.

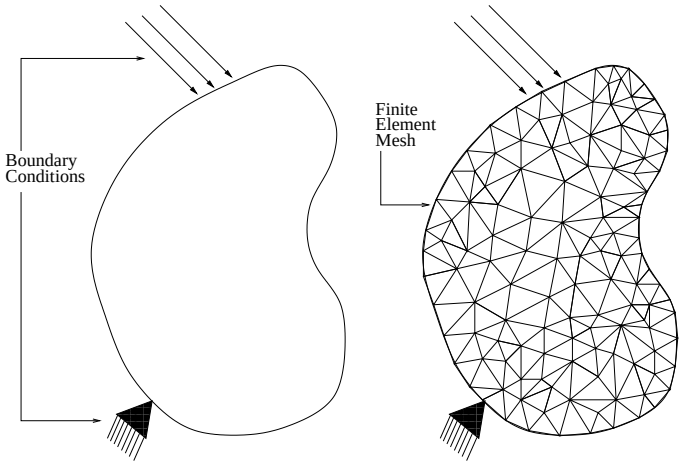


FIGURE 1 Typical finite element mesh

2.2.2 Finite Element Formulation

In order to avoid the difficulties of handling the strong form of the PDE directly, the continuity requirements are reduced to obtain a weak form, which is solved over a discretized domain [1].

For constructing the weak form of the governing equation, equation (1) is multiplied by an arbitrary vector and integrate over the domain Ω. We get virtual work as weak form in which the arbitrary function is the virtual displacement δu_i. Application of Gauss Divergence Theorem and traction boundary conditions with Cauchy's rule, we get,

$$\int_{\Omega} \delta u_i \rho \ddot{u}_i d\Omega + \int_{\Omega} \delta \varepsilon_{ij}(\underline{u}) \sigma_{ij} d\Omega - \int_{\Omega} \delta u_i b_i d\Omega - \int_{\Gamma_t} \delta u_i \bar{t}_i d\Gamma = 0 \,, \tag{3}$$

where the four terms on the left hand side of the equation denote the virtual work of the inertial forces, internal forces, body forces and traction forces respectively. Equation (3) calculated over all the elements of the discretized domain, written in matrix form gives,

$$\sum_{e=1}^{n_e} [\int_{\Omega} \delta\{u\}^T \rho \{\ddot{u}\} d\Omega + \int_{\Omega} \delta([S]\{u\})^T \{\sigma\} d\Omega - \int_{\Omega} \delta\{u\}^T \{b\} \delta\Omega - \int_{\Gamma_t} \delta\{u\}^T \{\bar{t}\} d\Gamma] = 0 \,. \tag{4}$$

The displacements within an element of the mesh can be approximated as,

$$\{u\}^e \approx [N]^e \{\tilde{u}\}^e, \tag{5}$$

where $[N]^e$ are the shape functions used for approximation. Substituting (5) into (4) and performing assembly over all the elements, we get,

$$\sum_{e=1}^{n_e} [[M]^e \{\ddot{u}\}^e + [K]^e \{\tilde{u}\}^e - \{f\}^e_{ext}] = 0, \tag{6}$$

where $[M]^e, [K]^e$ and $\{f\}^e_{ext}$ are the elemental mass matrix, stiffness matrix and external force vector respectively. The inertia term in (6) becomes zero for a static problem.

2.2.3 Assembly

The present work focuses on FEA of three-dimensional elastic structures, which are discretized using 20-noded hexahedral elements (HEX20) and 4-noded tetrahedral elements (TET4). For both element types, equation (7) can be used to compute the elemental stiffness matrices [1]. In equation (7), partial derivatives of the shape functions are stored in $\boldsymbol{B}$ with respect to x, y and z directions. ξ, η and ζ constitute the local coordinate system for each individual element. The Jacobian and constitutive matrix

are represented by J and D respectively. The Gauss Quadrature (GQ) weights for integration are denoted by w_{ξ_i}, w_{η_j} and w_{ζ_k}. n_g is the number of gauss points required. The assembly of all the elemental $[K]^e$ results in the global stiffness $[K]$ matrix.

$$\begin{aligned}
[K]^e &= \int_V [B]^T [D][B]\, dx dy dz \\
&= \int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} B(\xi,\eta,\zeta)^T D B(\xi,\eta,\zeta) \mid J(\xi,\eta,\zeta) \mid d\xi d\eta d\zeta \\
&= \sum_{i=1}^{n_g}\sum_{j=1}^{n_g}\sum_{k=1}^{n_g} w_{\xi_i} w_{\eta_j} w_{\zeta_k} B(\xi_i,\eta_j,\zeta_k)^T D B(\xi_i,\eta_j,\zeta_k) \mid J(\xi_i,\eta_j,\zeta_k) \mid
\end{aligned} \tag{7}$$

3 Three-stage FEA Matrix Generation for Unstructured Mesh

During FEA, the global stiffness matrix is generated by performing numerical integration and assembly of the elemental matrices. This matrix, after incorporating proper boundary conditions, is solved to generate the solution to the problem. In the traditional global stiffness matrix generation approach for both CPU-based and parallel applications, the NZ entries are computed and their indices and values are stored in different arrays depending on the sparse storage format used[3]. Sanfui and Sharma[22] implemented an approach of assembly by splitting into a symbolic kernel and a numeric kernel using structured mesh for a simple cantilever beam with eight-noded hexahedral elements. Structured meshes work well for simpler geometries and are easier to implement. However, in many real-world problems, unstructured grids become a necessity. Unlike a structured grid, the connectivity pattern of an unstructured grid is not fixed. This creates the need for the neighbor matrix computation. In this work, a similar idea of kernel division is extended to an unstructured mesh by dividing the matrix generation into three distinct stages. In the first stage, a new data structure called the *neighbor matrix* is introduced. This neighbor matrix is computed from the connectivity information of the unstructured mesh in the form of two one-dimensional arrays. During the second stage, the indices of NZ entries in the sparse storage format are computed and stored in parallel. This computation of indices can be separated from the value computation of NZ entries because it depends only on the connectivity information of the mesh. Finally, in the third stage, the NZ values are computed and stored based on the indices computed in the second stage. Thus, the index and value computation for NZ entries are decoupled for unstructured meshes with the help of the mesh preprocessor. This, although straightforward in case of structured meshes, becomes challenging for unstructured meshes where the data is scattered irregularly over the mesh. Furthermore, the study is extended to include different type of elements and their effect on the speedup. In the study by Sanfui and Sharma[22], due to the usage of identical elements, the elemental calculations were performed only once on the CPU and the elemental stiffness matrix was sent to the GPU. In this work, the effect of computing the elemental stiffness matrix for each element inside and out of the assembly kernel on GPU in connection to the kernel division strategy is investigated.

3.1 Mesh Preprocessing: Neighbor Matrix Computing

Due to restrictions of the memory hierarchy on the GPU hardware, the entire data cannot be made available to all the compute threads at the same time. A common solution to this problem is to process the data in a manner such that it can be segregated easily for each thread and only the part of the data required by a thread can be made available exclusively. Furthermore, since the FEA matrices are large and sparse in nature, specialized sparse storage formats such as COO, CSR, and ELL are required to store the values. This creates more challenges for writing and accessing the global stiffness matrix due to compaction of the NZ entries while storing into the sparse format. For example, in figure 2, a one-dimensional domain is taken with four elements marked as '1', '2', '3', and '4'. The four elemental matrices considering two degrees of freedom (DOF) per node are shown in the figure. If assembly is to be performed in the standard dense format, as shown by K^g in the figure, it can be done directly from the connectivity mapping of the mesh. However, if the global stiffness matrix is to be assembled directly into ELL format as shown by K^g_{ELL} in the same figure, the target locations of NZ entries (marked by red) get distorted, and the target indices for the elemental matrices cannot be found directly from the mesh connectivity information alone. This can be handled on the GPU by precomputing the column indices and performing a bisection search on the column array when a value is to be added

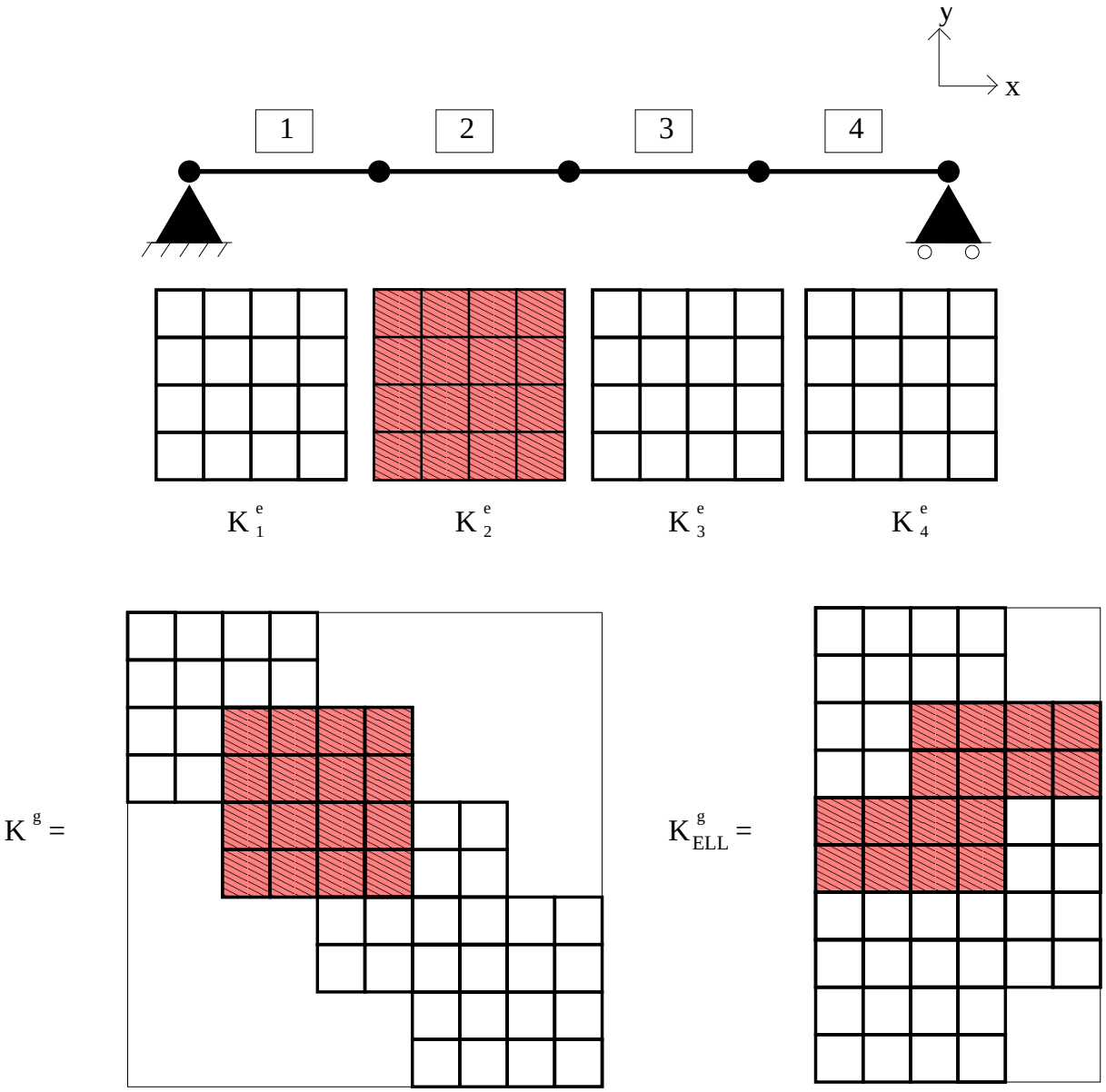


FIGURE 2 Assembly into the standard format versus the sparse storage formats

to a particular row. However, this search operation is undesired on the GPU and usually results in a performance loss. In the proposed implementation, the search is instead performed at the node-level with the pre-performed neighbor matrix to reduce the effect of the search.

Due to these reasons, for all the steps after the numerical integration, a data-structure is introduced which is named as neighbor matrix, along with the connectivity and coordinate matrices. This neighbor matrix is helpful on the GPU in the following ways.

- For assembly into a sparse matrix such as COO, CSR, and ELL, the neighbor matrix can reduce the search span for writing each entry.
- Using this matrix, the assembly can be broken into two parts where the indices and values of the sparse global stiffness matrix can be written in two different kernels.
- This matrix can provide the memory sizes to be allocated to the sparse formats on GPU such as COO and CSR and ELL, which are not known a priori for unstructured FE grid.

3.1.1 Neighbor Matrix Computation

The implementation details of the neighbor matrix computation are outlined in this section. It is computed using a combination of custom kernels and functions available in the *thrust* library of the NVIDIA CUDA Toolkit. Thrust is a software library for CUDA based on the C++ Standard Template Library (STL) that allows the user to implement high-performance parallel applications through a high-level interface. The neighbor matrix is designed to store a sorted list of nodes connected to a particular node of the unstructured finite element mesh. This matrix will later be required to allocate memory and to store the NZ entries of the global stiffness matrix in a sparse storage format. The neighbor matrix is demonstrated in figure 3, where a mesh of triangular elements is taken. The example mesh has 7 nodes marked as '1', '2', … , '7' and 6 elements marked as '*a*', '*b*', … , '*f*'. The neighbor matrix consists of two one-dimensional arrays called *nNum[]* and *nInd[]* that are also shown in the figure. It is to be noted that *nNum[]* is shown as two-dimensional in the figure for clarity. However, it is flattened and stored as a one-dimensional array on the GPU. In the two-dimensional format, it contains the same number of rows as the number of nodes in the mesh. Each row contains a sorted list of nodes connected to the corresponding node including itself. For example, in the figure node '5' is connected to node '4', '6' and '7'. Hence, the 5^{th} row of *nNum[]* array lists '4', '5', '6' and '7'. To identify the entries in the flattened array for each node, the *nInd[]* array is introduced in the figure. The $(i-1)^{th}$ and i^{th} entries of the *nInd[]* array gives the start and end indices of *nNum[]* array for the i^{th} node. For example, the start and end indices for node 5 in the figure is given by the 4th and 5th entry of *nInd[]* (16 and 20) respectively. Furthermore, the *nInd[]* array will be used to compute the memory allocation size for sparse storage formats on the GPU. It is to be noted here that since the neighbor matrix computation is dependent only on the connectivity information of the mesh, it can be performed for any type and order of elements with small modifications in the implementation.

Algorithm 1 lists the functions called for the neighbor matrix computation. These functions are described in details with the help of figure 4. In the figure, the neighbor matrix is computed step-by-step for the triangular mesh shown in figure 3. The steps in the neighbor matrix computation are demonstrated using arrays [I], [II], … [XI] in the same figure. The flattened arrays are also shown in the figure with their respective indices ([V′] and [IX′]).

Algorithm 1 Neighbor matrix computing

Input: $nElem$: number of elements; $nNode$: number of nodes; n_e: nodes per element; $C[nElem, n_e]$: connectivity matrix;
Output: $nNum[]$: Array [IX′]; $nInd[]$: Array [XI];

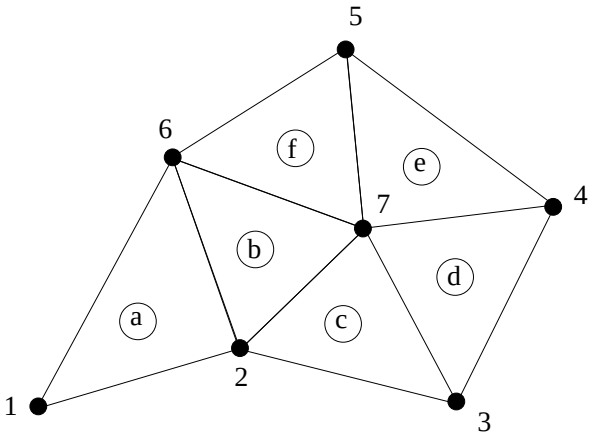
1: $countElemNum <<< ((nElem-1)/256)+1, 256 >>> (C[])$
2: $thrust::inclusive\_scan()$
3: $fillElemNum <<< ((nElem-1)/256)+1, 256 >>> (C[])$
4: $thrust::inclusive\_scan()$
5: $fillNodeNum <<< ((nNode-1)/256)+1, 256 >>> (C[])$
6: $thrustDynamicSort <<< ((nNode-1)/128)+1, 128 >>> (C[])$
7: $thrust::inclusive\_scan()$

3.1.1.1 countElemNum Kernel

The purpose of this kernel is to compute the number of elements connected to each node of the mesh as shown in figure 4. The array [I] shows the node numbers of the given triangular mesh in figure 3. It can be seen that node 1 is connected to element '*a*' only. Therefore, one is stored at the first place of array [II]. Similarly, node 2 is connected to elements '*a*', '*b*' and '*c*', therefore, three is stored at the second place of array [II]. By following the same procedure, the other places of array [II] are filled with the numbers of elements connected to the nodes. The above procedure for *CountElemNum kernel* is presented Algo. 2, which is launched with threads equal to the total number of elements in the mesh. It can be seen from the algorithm that *atomicAdd* function is used and the values of array [II] are stored in *neighborCount[]* array.

3.1.1.2 Inclusive Scan

An inclusive scan is performed to find the size of memory allocation to the GPU using *thrust* library. A cumulative sum of the numbers shown in the array [II] is stored in the array [III]. The number on the last position of the array [III] (15, in the example



	nNum[]						*nInd[]*
Node 1	1	2	6				3
Node 2	1	2	3	6	7		8
Node 3	2	3	4	7			12
Node 4	3	4	5	7			16
Node 5	4	5	6	7			20
Node 6	1	2	5	6	7		25
Node 7	2	3	4	5	6	7	31

FIGURE 3 Neighbor matrix for a triangular mesh

Algorithm 2 countElemNum Kernel

Input: $nElem$: number of elements; $nNode$: number of nodes; n_e: nodes per element; $C[nElem, n_e]$: connectivity matrix;
Output: $neighborCount[]$: Array [II];

1: $countElemNum <<< ((nElem - 1)/256) + 1, 256 >>> ()$
2: __global__ void countElemNum(int *C, int *neighborCount, int nElem)
3: $int\ tx = blockIdx.x * blockDim.x + threadIdx.x$
4: **if** $tx < nElem$ **then**
5: **for** $i \leftarrow 1 : n_e$ **do**
6: $atomicAdd(\&neighborCount[C[n_e * tx + i]], 1)$;
7: **end for**
8: **end if**

figure) shows the memory size required on GPU for the given mesh. For clarity, the array [IV] is shown in figure 4 which shows the number of empty cells equivalent to the number shown in the array [II] for every node. Array [III] is also utilized to identify entries for each node in array [V].

3.1.1.3 fillElemNum Kernel

The memory allocated on the GPU after the inclusive scan is now used for storing the element numbers connected to each node. The array [V] in figure 4 represents those connected elements. It is noted that the array [V] is stored as a one-dimensional array [V′] on GPU which can be seen in the inset of the same figure. It can be seen that element 'a' is stored at the first place of the array [V′] followed by the elements of node 2, that are, 'a', 'b', and 'c'. Similarly, the elements connected to other nodes are stored. The indices of the array [V′] are the same as shown in the array [III] in order to locate the connected elements with the nodes. The procedure of this kernel is shown in Algo. 3, which is launched with threads equal to the total number of elements in the mesh. It is noted that the one-dimensional array [V′] is stored in *elemNum[]* array after executing this kernel.

3.1.1.4 Inclusive Scan

After storing the number of elements in the array [II], the total numbers of nodes connected to those elements are stored in the array [VI]. In the given mesh, every element is connected to $n_e = 3$ nodes. Therefore, the number '3' is stored at the first position of the array [VI]. For the second position of the array [VI], three elements 'a', 'b', and 'c' are stored, therefore $3 \times n_e =$ '9' is stored. Similarly, the numbers are stored for other positions of the array [VI]. It can be noted that the number stored in the array [VI] can be directly found from the array [III] after multiplying every number with n_e. The numbers of array [VI] are stored in $nodeCount[]$ array. Another inclusive scan is performed on the array [VI] and the cumulative sum of the numbers in the array [VI] is stored in the array [VII]. The number on the last position of the array [VII] gives the size of memory allocation on GPU.

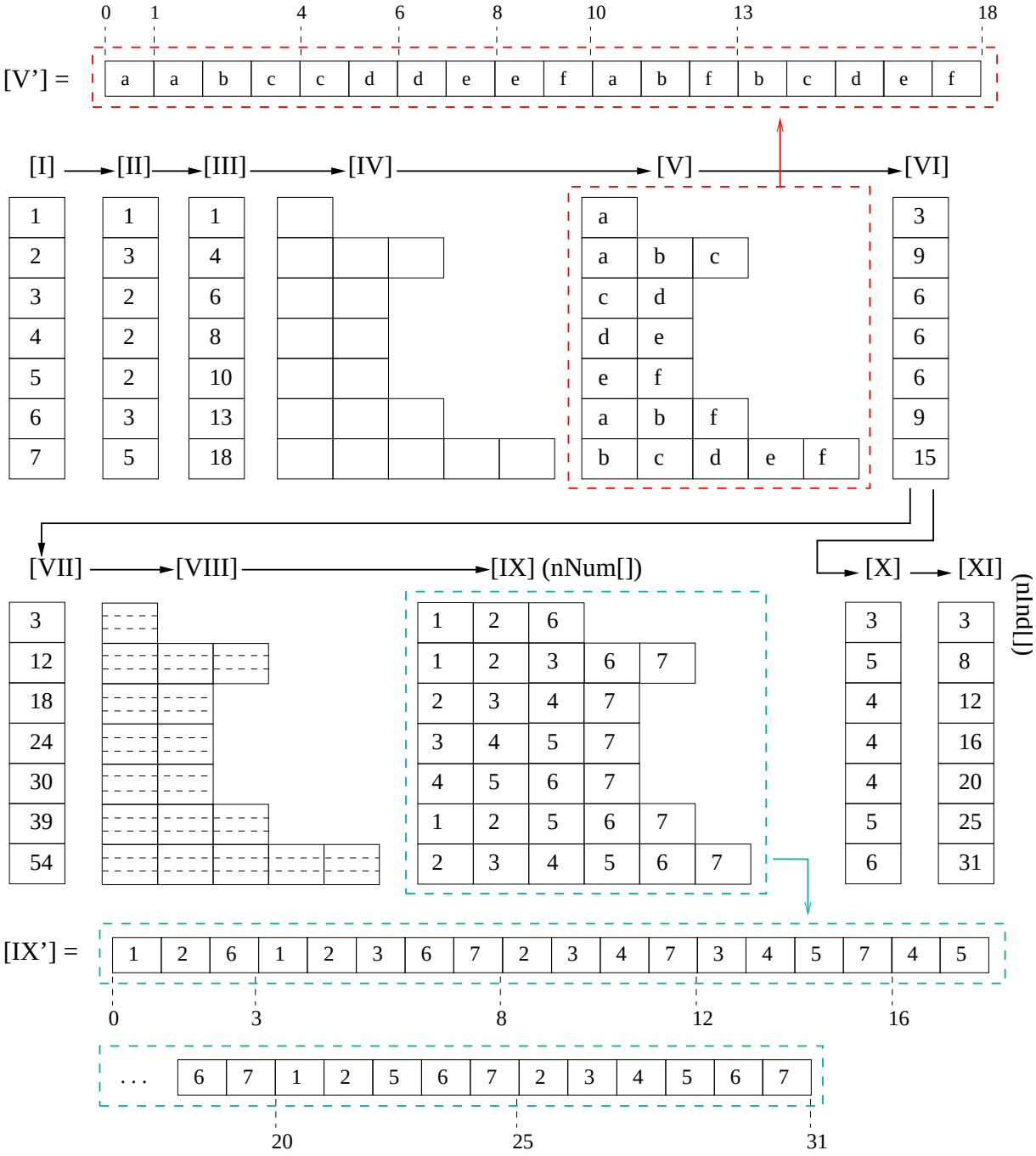


FIGURE 4 Pre-processing the mesh data for assembly

Similar to the approach described in Section 3.1.1.2, the array [VIII] is shown in figure 4 for clarity, which has the number of empty cells equivalent to the numbers shown in the array [VII]. After the inclusive scan, the required memory is allocated to *nodeNum[]* array on GPU.

3.1.1.5 fillNodeNum Kernel

The memory allocated in the inclusive scan in Section 3.1.1.4 is filled with the node numbers in the array [VIII]. The first position of the array [VIII] stores '1', '2', and '6' node numbers because these nodes are connected to the element '*a*' which was stored at the first position of the array [V]. For the second position of the array [VIII], the elements stored at the second

Algorithm 3 fillElemNum Kernel

Input: $nElem$: number of elements; $nNode$: number of nodes; n_e: nodes per element; $C[nElem, n_e]$: connectivity matrix; $neighborCount$[];

Output: $elemNum$[]: Array [V′];

1: $fillElemNum <<< ((nElem - 1)/256) + 1, 256 >>> ()$
2: $\_\_global\_\_$ void fillElemNum(int *elemNum, int *C, int *neighborCount, int nElem)
3: $int\ tx = blockIdx.x * blockDim.x + threadIdx.x$
4: **if** $tx < nElem$ **then**
5: **for** $i \leftarrow 1 : n_e$ **do**
6: int temp
7: $temp = atomicSub(\&neighborCount[C[n_e * tx + i]], 1)$
8: $elemNum$[temp - 1] $= tx$
9: **end for**
10: **end if**

place of the array [V] are used that are 'a', 'b', and 'c'. The nodes connected to element 'a' are '1', '2', '3', to element 'b' are '2', '6', '7', and to element 'c' are '2', '3', and '7'. These node numbers are stored at the second place of array [VIII]. Similarly, the remaining positions of the array [VIII] (*nodeNum[]* array in Algo. 4) store the nodes. This procedure is shown in Algo 4 which is launched with threads equal to the number of nodes in the mesh.

3.1.1.6 thrustDynamicSort Kernel

This kernel is designed to sort the nodes at every row of the array [VIII] and to remove the duplicate nodes. At the first position of the array [VIII], the nodes '1', '2', and '6' are already sorted with no duplicates. These nodes are copied to the array [XI]. At the second place of the array [VIII], the nodes are unsorted and the duplicate nodes are also available. At this stage, the kernel sorts them and duplicates are removed. Thereafter, the nodes '1', '2', '3', '6', and '7' are stored at the second position of the array [IX]. By following the same procedure, the other positions of the array [IX] are filled. It is noted that the two-dimensional array [IX] is a graphical representation for clarity, however, the data is stored in a one-dimensional array [IX′] in the same figure with the required indices. This kernel uses the *Dynamic Parallelism* feature of the Kepler microarchitecture. With this feature, grids can be launched from inside a GPU kernel. Two *thrust* calls are made inside the kernel for each node as shown in line 7 and line 8 of Algo. 5 for first sorting the nodes and then removing the duplicates. The kernel is launched with threads equal to the number of nodes in the mesh and the array [IX] is stored in *nNum[]* array. At the end of this kernel, the *nodeCount[]* array (array [VI]) is updated in line 9 by subtracting the number of repeated entries from each corresponding row. The updated values are shown in array [X] in figure 4. In essence, this array simply lists the number of neighboring nodes of each node.

3.1.1.7 Inclusive Scan

Similar to Sections 3.1.1.2 and 3.1.1.4, another inclusive scan is performed on *nodeCount[]* array and the values are stored in array [XI]. This array is the *nInd[]* array of neighbor matrix.

After performing all the steps of Algo. 1, the neighbor matrix is generated in the form of two one-dimensional arrays *nNum[]* and *nInd[]* on the GPU. The neighbor matrix is used to compute the GPU memory allocation size for the global stiffness matrix in the following ways.

- **ELL Sparse Format:** ELL sparse format consists of two matrices for storing the column indices and the values of the sparse matrix. Both of these matrices have rows equal to the number of degrees of freedom of the system. The number of columns is computed by calculating the maximum number of nodes surrounding any particular node in the mesh and multiplying it by DOF per node. For a general mesh, after calling `thrustDynamicSort` kernel of Algo. 5, `thrust::max_element` is called with the final *nodeCount[]* array (array [X] in figure 4) to give the maximum number of neighboring nodes for any node of the mesh. This number multiplied with the DOF per node gives the column number of ELL format.

Algorithm 4 fillNodeNum Kernel

Input: $nElem$: number of elements; $nNode$: number of nodes; n_e: nodes per element; $C[nElem, n_e]$: connectivity matrix; $neighborCount$[];

Output: $nodeNum$[]: Array [VIII];

1: $fillNodeNum <<< ((nNode - 1)/256) + 1, 256 >>> ()$
2: __global__ void fillNodeNum(int *elemNum, int *C, int *neighborCount, int nNode)
3: $int\ tx = blockIdx.x * blockDim.x + threadIdx.x$
4: **if** $tx < nNode$ **then**
5: **for** $i \leftarrow 1 : neighborCount[tx]$ **do**
6: **for** $j \leftarrow 1 : n$ **do**
7: $int\ p = C[n * elemNum[i] + j]$
8: $temp = atomicSub(\&nodeCount[tx], 1)$
9: $nodeNum[temp - 1] = p$
10: **end for**
11: **end for**
12: **end if**

Algorithm 5 thrustDynamicSort Kernel

Input: $nElem$: number of elements; $nNode$: number of nodes; n_e: nodes per element; $C[nElem, n_e]$: connectivity matrix; $neighborCount$[];

Output: $nNum$[]: Array [IX′];

1: $thrustDynamicSort <<< ((nNode - 1)/128) + 1, 128 >>> ()$
2: __global__ void thrustDynamicSort(int *elemNum, int *C, int *neighborCount, int nNode, int *nodeCount)
3: $int\ tx = blockIdx.x * blockDim.x + threadIdx.x$
4: **if** $tx < nNode$ **then**
5: $p1 =$ starting index in $nodeNum$ for tx
6: $p2 =$ length in $nodeNum$ for tx
7: thrust::sort(thrust::seq, $nodeNum + p1$, $nodeNum + (p1 + p2)$)
8: thrust::unique_copy(thrust::seq, $nodeNum + p1$, $nodeNum + (p1 + p2)$, $nNum + p1$)
9: $nodeCount[tx] = nodeCount[tx]-$ no. of repeated entries
10: **end if**

- **COO/CSR Sparse Format:** COO format consists of three one-dimensional arrays for storing the row indices, column indices, and values of the NZ entries. For this format, the last entry of *nInd[]* array is used. This value multiplied by the DOF per node gives the length of the three arrays of the COO format. For storing in CSR format, the value array and the column array remains of the same size. The size of the row offsets array is the number of rows in the global stiffness matrix plus one.

In the present work, assembly with ELL format using HEX20 (20-noded hexahedral elements) and TET4 (4-noded tetrahedral elements) element types is implemented on unstructured meshes. The next section outlines the details of the index computation stage.

3.2 Index Computation for NZ Entries

In this stage, the row and column indices of the sparse storage format for storing the NZ entries in the global stiffness matrix are calculated. The idea behind this stage is that the number of NZ entries in a particular row or column of the global stiffness matrix can be calculated by multiplying the number of neighboring nodes to a node with its DOF. For a node i with θ_i number of neighboring nodes, there can be $(\theta_i + 1)ndof$ total NZ entries in the associated rows and columns of the global stiffness matrix, where $ndof$ is the DOF per node. The corresponding indices can be calculated from the neighboring node numbers. In this kernel, each thread is assigned to one node of the finite element mesh as shown in figure 5.

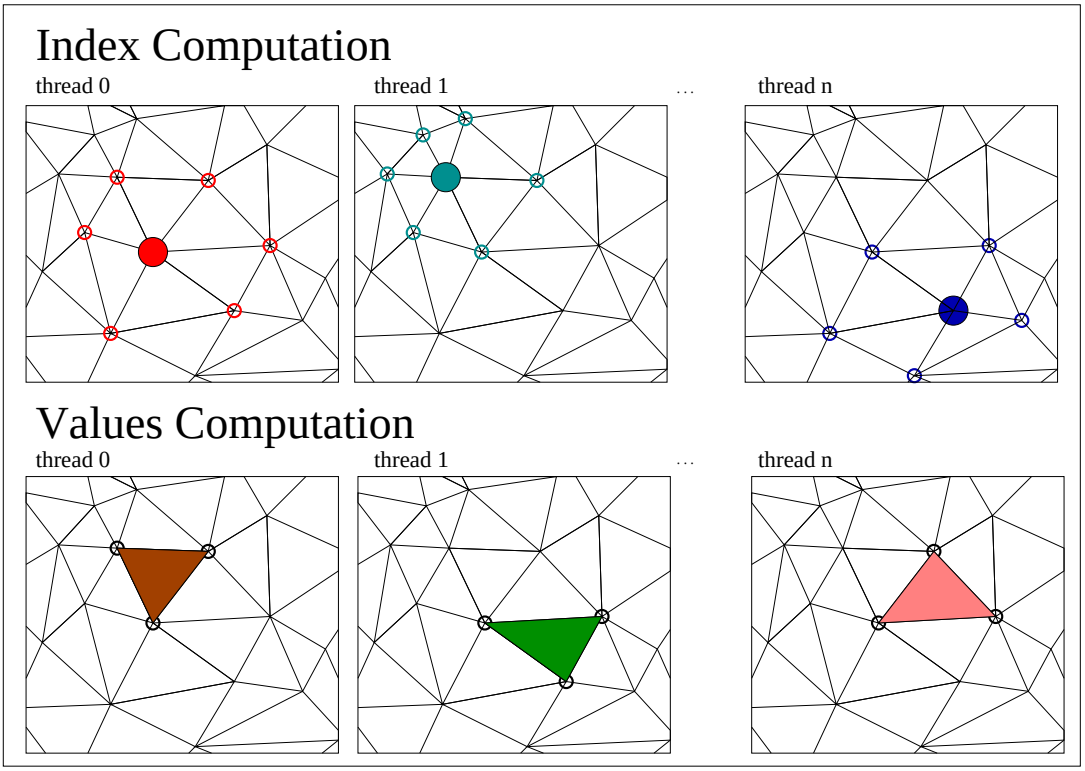


FIGURE 5 Thread allocation scheme for the index computation and values computation kernels.

A generalized algorithm for the index computation is presented in Algo. 6 in which the required data is preprocessed for determining row and column indices for every NZ entry. At step 4, neighboring nodes (S^i) is fetched from the neighbor matrix computed in the mesh preprocessor stage by using Algo 1. At step 6 of Algo 6, one thread is assigned to every node i. Each thread first initializes two null sets (Φ^i, Γ^i) for storing row and column indices. It can be seen that step 10 stores d_k^i DOF of node i in Φ^i that is copied ($\theta_i \times ndof$) times. As can be seen from figure 6, all DOFs of node i are copied one by one in Φ^i. Therefore, Φ^i stores ($\theta_i \times ndof^2$) indices for node i. Every Φ^i is then copied to Φ set of row indices. For the column indices for every node i, d_k^j DOF of node $j \in S^i$ is copied into Γ^i (refer step 16). Once all DOFs of neighboring nodes to node i are copied into Γ^i, the same set is copied ($ndof$) times to Γ. The set Γ also stores ($\theta_i \times ndof^2$) indices for node i. Figure 6 shows D^i, S^i, Φ^i, Γ^i, Φ, and Γ for node i in the two-dimensional domain having two DOF per node.

3.3 Values Computation for NZ Entries

During this stage, the values of the global stiffness matrix entries are computed and stored in a sparse storage format. An element-by-element assembly strategy is adopted for the implementation as shown in figure 5. Since each node is shared by a number of elements in the mesh, a situation may occur where a number of threads try to access the same memory location simultaneously, resulting in race condition or data race. This can cause inaccuracies in the final result. This is handled at the hardware level by using atomic operations.

A *node-to-node* connection, as the name suggests, is simply the connection between two nodes of a finite element. It can be a straight line or a point depending on the choice of nodes to form the connection. During assembly, each of these connections writes $ndof^2$ (in the present implementation, 3^2) entries from K^e into the `Value` array of a sparse storage format to store NZ values.

A generalized algorithm for the value computation of NZ entries is presented in Algo. 7 in which NZ entries are stored based on the row and column indices (Φ, Γ) determined using the computed indices in the previous stage. In this kernel, one thread is assigned to one element at step 1. Using the connectivity matrix for element e ($C[e, nnodes]$), the target index (p) for the first DOF (d_1^j) of node $j \in C[e, nnodes]$ is searched in $\Gamma^i \in \Gamma$ for every *node-to-node* connection (refer step 4). We need to search

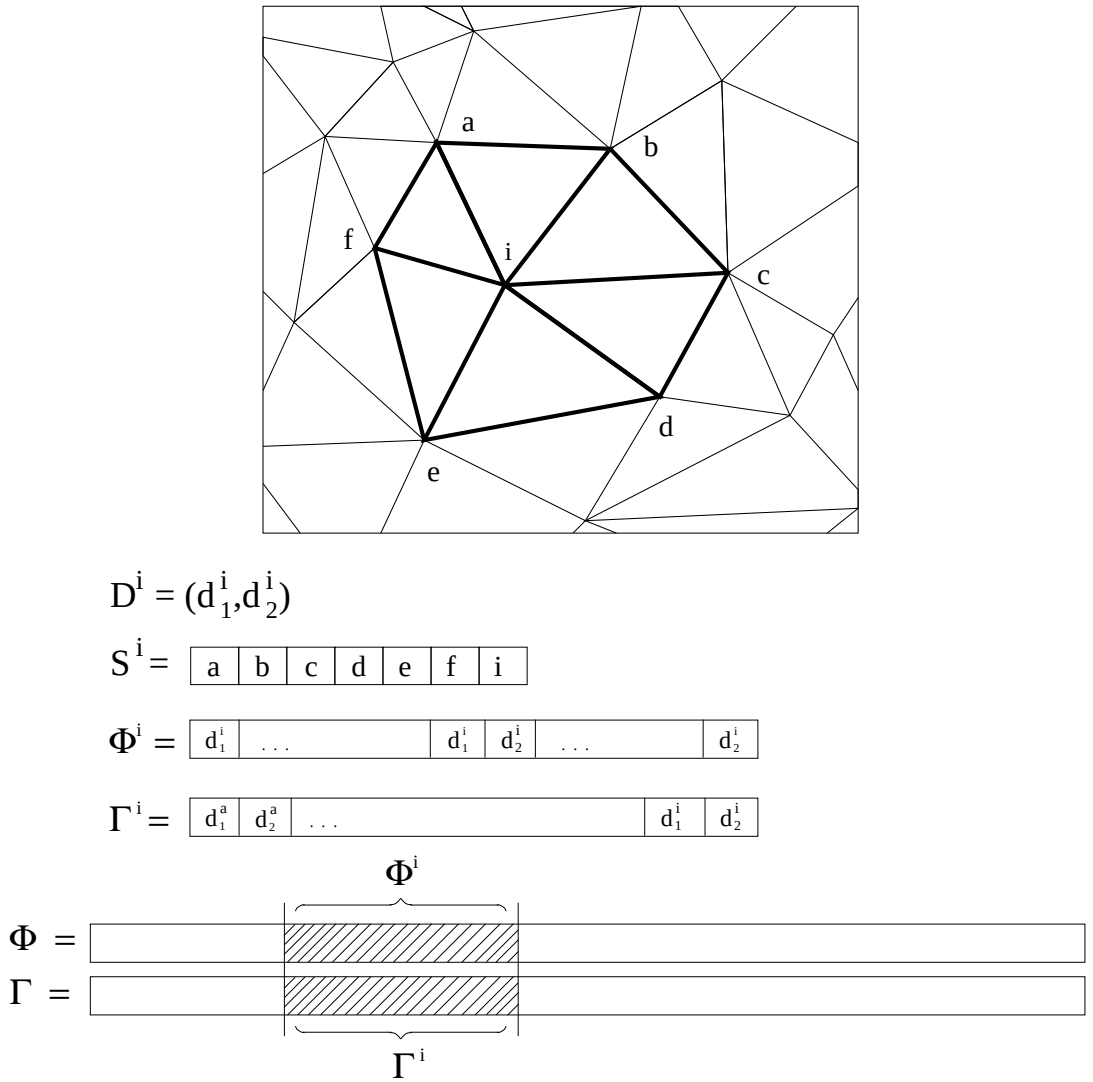


FIGURE 6 Different sets of data for node i are presented for a two-dimensional domain having two DOF per node. D^i represents the DOFs of node i. S^i represents the set of neighboring nodes to node i. Φ^i and Γ^i represent the row and column indices respectively of the NZ entries related to node i.

only the first DOF (d_1^j) of node $j \in C[e, nnodes]$ because other DOF can be determined from the pattern used for storing indices using Algo. 6. Finally, the entries of the elemental stiffness matrix are stored into the sparse storage format as shown in step 9.

3.4 Assembly into COO, CSR and ELL

Algos. 6 and 7 are described in a generic way so that they can be coupled with any sparse storage format. Essentially, these algorithms are described for COO storage format in which row and column indices sets (Φ, Γ) have the same size as the number of non-zeroes in the global stiffness matrix. For storing row and column indices for the COO format, Algo. 6 is executed to get (Φ, Γ), and for storing NZ entries, Algo. 7 is executed to get the `Value` array. The step 9 of Algo. 7 is modified as

$$\texttt{Value}[col\_ind] += K^e[l, k]$$

The index and values computation kernels for CSR storage format remain the same as described in Algos. 6 and 7 for COO format. The only difference can be seen at step 10 of Algo. 6, when the row indices set (Φ) only stores row offsets. The storage of NZ entries in the `Value` array remains the same as COO format.

Algorithm 6 Kernel 1: Computing row and column indices of NZ entries on GPU

Input: E: number of elements; $ndof$: DOFs per node; I: total number of nodes; $nnodes$: nodes per element; $C[E, nnodes]$: Connectivity matrix;

Output: Φ : row indices for NZ entries; Γ : column indices for NZ entries;

1: **for** $i \leftarrow 1 : I$ **do**
2: $D^i = (d^i_1, \dots, d^i_{ndof})$ % d^i_k: k^{th} global DOF of node i
3: $S^i = (nNum[nInd[i-1]], nNum[nInd[i-1]+1], ..., nNum[nInd[i]])$
4: $= (n^i_1, \dots, n^i_i, \dots, n^i_{\theta_i})$ % S^i: Set of neighboring nodes to node i; θ_i: No. of neighboring nodes to node i including itself; n^i_j: global node number of node j
5: **end for**
6: **for** $\forall i \in I$ **do** % On GPU by assigning one thread to every node i
7: $\Phi^i = \emptyset, \Gamma^i = \emptyset$
8: **for** $k \leftarrow 1 : ndof$ **do**
9: **for** $j \leftarrow 1 : (\theta_i \times ndof)$ **do**
10: $\Phi^i = \Phi^i \cup d^i_k$ % Storing DOF of node i
11: **end for**
12: **end for**
13: $\Phi = \Phi \cup \Phi^i$ % Storing DOF into the row index set
14: **for** $j \leftarrow 1 : \theta_i$ **do**
15: **for** $k \leftarrow ndof$ **do**
16: $\Gamma^i = \Gamma^i \cup d^j_k, j \in S^i$ % Storing DOF of all neighboring nodes to node i
17: **end for**
18: **end for**
19: **for** $k \leftarrow ndof$ **do**
20: $\Gamma = \Gamma \cup \Gamma^i$ % Copying $ndof$ times and Storing Γ^i into the column index set
21: **end for**
22: **end for**
23: Store Φ and Γ on the global memory of GPU

ELL sparse storage format has the column-major ordering, due to which memory coalescence for reading and writing into global memory is ensured. This format having a structured nature, the row indices set (Φ) is not required and thus, steps 8 to 13 can be removed from Algo. 6.

ELL storage format consists of two matrices for storing column indices and NZ entries. Indices of the matrix are stored in a one-dimensional array similar to COO format (Γ). In this case, the size of Γ^i for every node i at step 16 of Algo. 6 is same as the maximum number of NZ entries in any row of the global stiffness matrix. This maximum number of NZ entries can be found from *nodeCount[]* array described in Section 3.1.1.6. Step 20 is also not required for ELL storage format and thus, the corresponding `for-loop` can be removed. Rest of the steps remain the same for Algo. 6 for determining and storing column indices in (Γ) set. The NZ entries are stored in a one-dimensional `Value` array which has the same size as Γ set. In Algo. 7, the step 9 is modified to store NZ entries into the `Value` array as

$$\texttt{Value}[col\_ind] += K^e[l, k]$$

Rest of the steps of Algo. 7 for ELL format remain the same as of COO format. In the present work, only ELL format is used for all the implementations.

4 Same and Separate Kernel Approaches

The same kernel and separate kernel approaches are used for assembling and storing the elemental stiffness matrices in the value array of the sparse storage format. As shown in figure 7, the neighbor matrix is computed first followed by index computation in the same kernel approach. In this approach, the generation of elemental stiffness matrix and value computation of the global

Algorithm 7 Kernel 2: Computing values of NZ entries using indices stored in Φ and Γ

Input: $ndof$: DOF per node; E: total elements; $nnodes$: nodes per element; $C[E, nnodes]$: Connectivity Matrix; Φ, Γ;
Output: K: Global stiffness matrix in the sparse storage format;

1: **for** $\forall e \in E$ **do** % Assign a thread to each element e
2: **for** $i \leftarrow\mid C[e:] \mid$ **do** % C[e:] represents all nodes of e
3: **for** $j \leftarrow\mid C[e:] \mid$ **do** % *Node-to-node* connection within an element e
4: Search target index (p) for d_1^j in the set $\Gamma^i \in \Gamma$ % d_1^j is the first DOF of node j
5: **for** $l \leftarrow ndof$ **do** % For assembling K^e into K
6: $row\_ind = p + ndof \times \theta_i(l-1)$ % Row index for NZ entry
7: **for** $k \leftarrow ndof$ **do**
8: $col\_ind = p + ndof \times \theta_i(l-1) + (k-1)$ % Column index for NZ entry
9: $K[\Phi(row\_ind), \Gamma(col\_ind)] += K^e[l,k]$ % Assembly of NZ entry via *node-to-node* connection of nodes i and j
10: **end for**
11: **end for**
12: **end for**
13: **end for**
14: **end for**

stiffness matrix are performed in the same kernel. This obviates the need to store the elemental matrices on the GPU. In the separate kernel approach as shown in figure 7, the elemental matrix generation is performed separately at the beginning and all the elemental stiffness matrices are stored in the global memory of the GPU. After computing the neighbor matrix, the indices of the global stiffness matrix are computed. Following this, the value computation kernel is launched that takes the previously computed elemental stiffness matrices from global memory and assembles them into the value array.

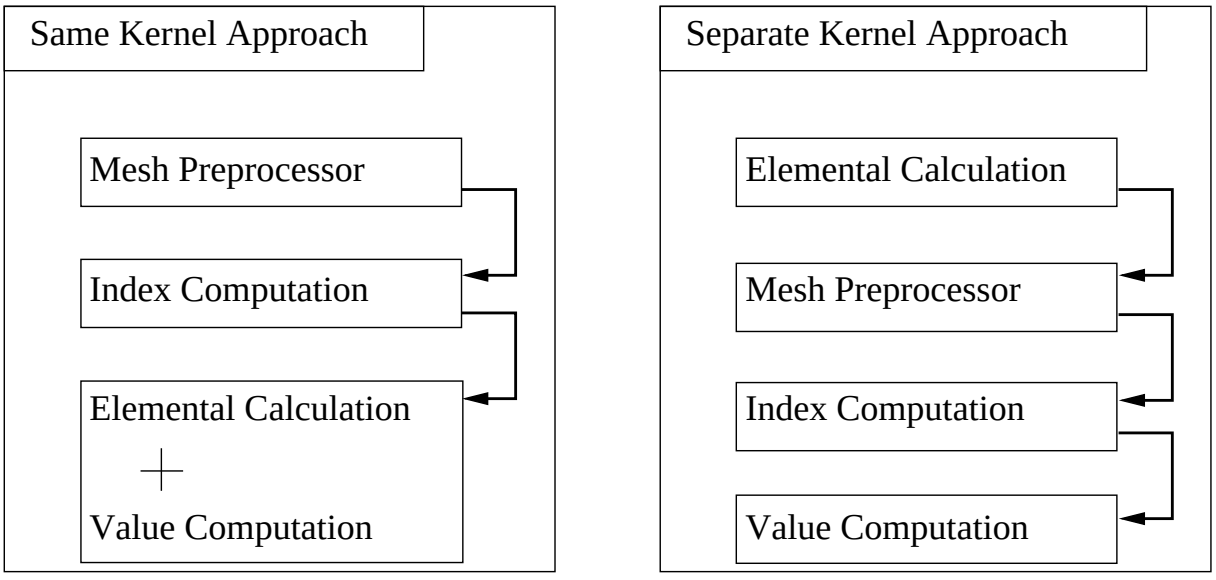
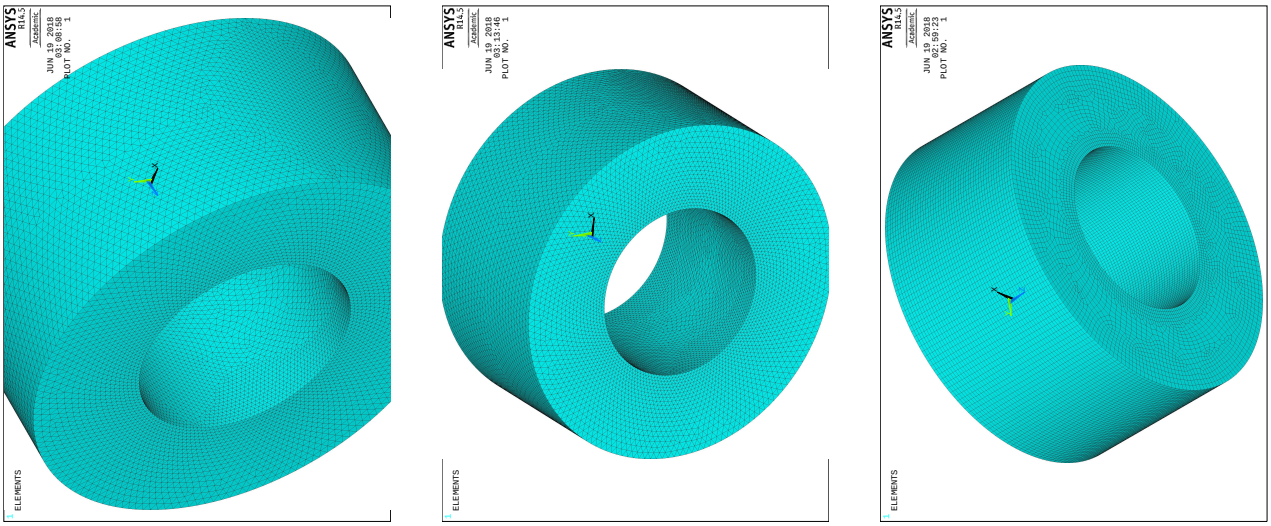


FIGURE 7 Flow charts for the same kernel and separate kernel approaches.



(a) TET4 (b) TET10 (c) HEX20

FIGURE 8 Hollow cylinder mesh using (a) TET4, (b) TET10 and (c) HEX20 element type

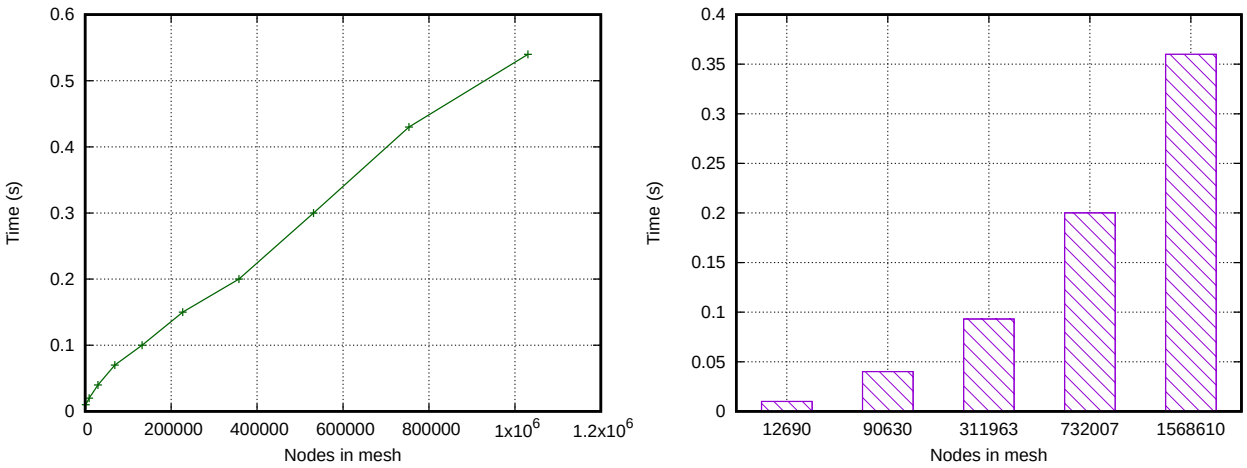
5 Results and Discussion

The assembly using the three-stage kernel division strategy for unstructured mesh is tested on two examples. The first example is a hallow cylinder in which the analysis and comparison of the proposed assembly strategy are performed. Another example of a connecting rod is considered in which the matrix is generated using the proposed strategy and the solution is generated for the given boundary conditions. Both the computation analyses are performed on a workstation with Intel Xeon ES1650 (6 core, 3.2 GHz) processor, 12 GB RAM, and K40c NVIDIA GPU. The GPU has 12 GB of global memory with 15 multiprocessors and 192 cores per multiprocessor.

5.1 Example 1: Hollow Cylinder

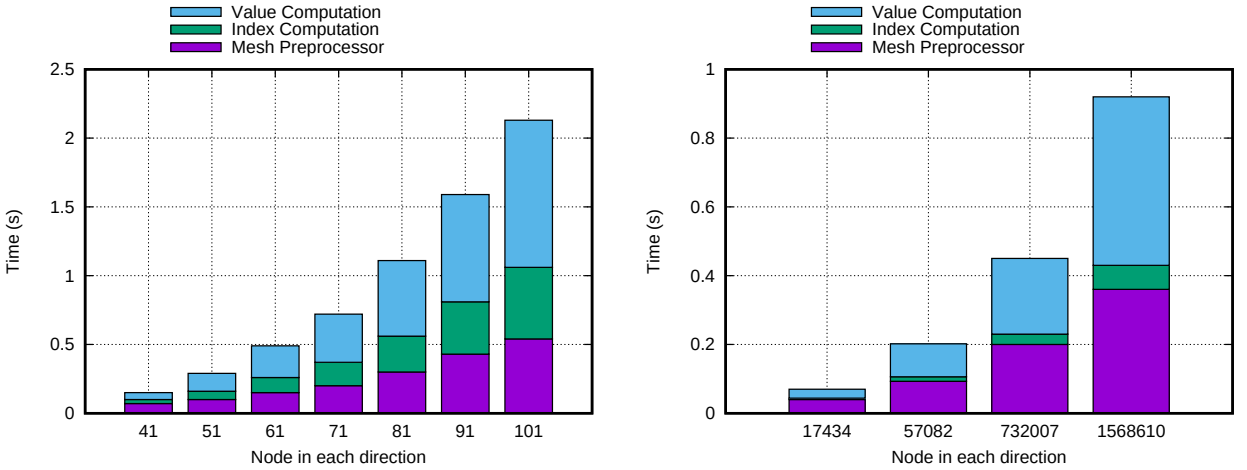
A hollow cylinder is considered, which is meshed with three types of meshes using ANSYS® software. Figure 8 shows the hollow cylinder with 4-noded tetrahedral (TET4), 10-noded tetrahedral (TET10) and 20-noded hexahedral (HEX20) meshes. The mesh preprocessing (neighbor matrix generation) time for different types of meshes is presented in table 1. This time is important because it is part of the total assembly time. Table 4 lists the times required by different steps of algorithm 1 for three different mesh sizes. It can be seen that among all the steps, *thrustDynamicSort* kernel takes the most amount of time for all the mesh sizes. Figure 9(a) shows the mesh preprocessing time for HEX20 for different numbers of nodes with the hollow cylinder example. It can be observed that the preprocessing mesh time is quite less for even a large size of mesh. With TET4 mesh type, the same observation can be seen from figure 9(b). The execution times of the mesh-preprocessor are found to have an almost linear relationship with the increasing number of nodes in the mesh.

The assembly time using HEX20 and TET4 mesh is shown in figure 10. This assembly time has been further divided into mesh preprocessing time, index computation time and values computation time. Figure 10(a) shows the variation among those computation times for different number of nodes using HEX20 element type using ELL sparse storage format. It is noted that the same number of nodes has been taken in all three-directions for meshing the hollow cylinder. It can be seen from the figure that the mesh preprocessing time consumes the least time as compared to other kernels for assembly. Most of the assembly time is consumed in the value computation kernel. Similar analysis can be seen in figure 10(b) by meshing the hollow cylinder with TET4 mesh type. The figure shows computation time of mesh preprocessing, index computation and values computation using ELL sparse storage format. Most of the time is consumed during the third stage in value computation followed by mesh preprocessing and index computation. The comparison of effective memory bandwidth using HEX20 and TET4 mesh is shown in figure 11. The achieved bandwidth for index computation, value computation and mesh preprocessor are shown. In figure



(a) HEX20 (b) TET4

FIGURE 9 Wall clock time comparison for mesh preprocessing using (a) HEX20 and (b) TET4 with increasing number of nodes in the mesh



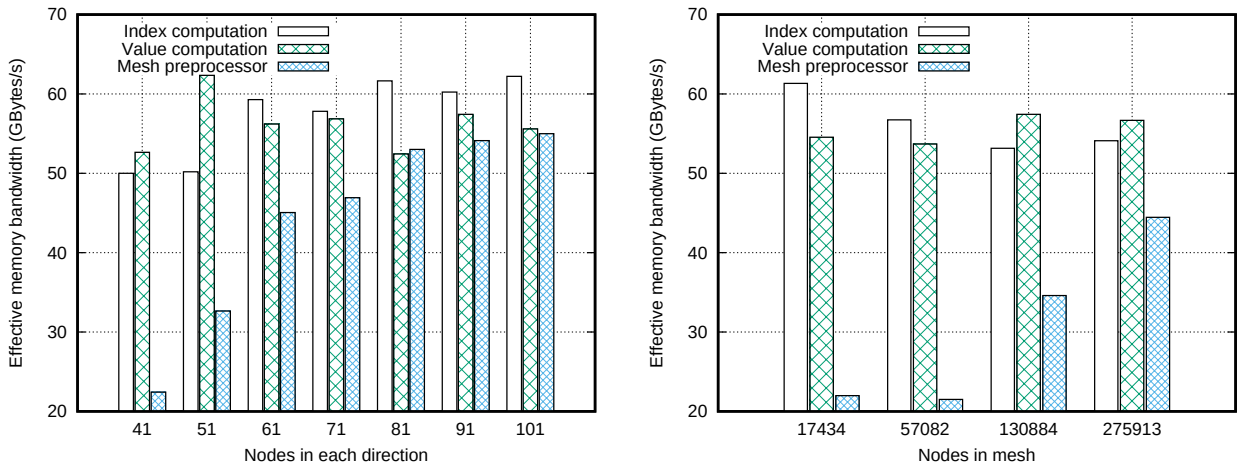
(a) HEX20 (b) TET4

FIGURE 10 Wall clock time comparison for (a) HEX20 and (b) TET4 element type for increasing number of nodes in the mesh

11(a), the variation of achieved memory bandwidth is shown for HEX20 element type using ELL sparse storage format for all those stages. Similar to figure 10(a), cylindrical mesh with same number of nodes in all direction is taken. It is observed that the achieved bandwidth for the mesh preprocessor is the least among the three stages. A similar observation can be done from the figure 11(b), where the achieved bandwidth of a hollow cylinder meshed with TET4 elements is compared for increasing node numbers. Little variation is seen in the bandwidth values for the index computation and values computation stages with increasing node numbers. The mesh preprocessor is observed to achieve the least memory bandwidth among the three stages.

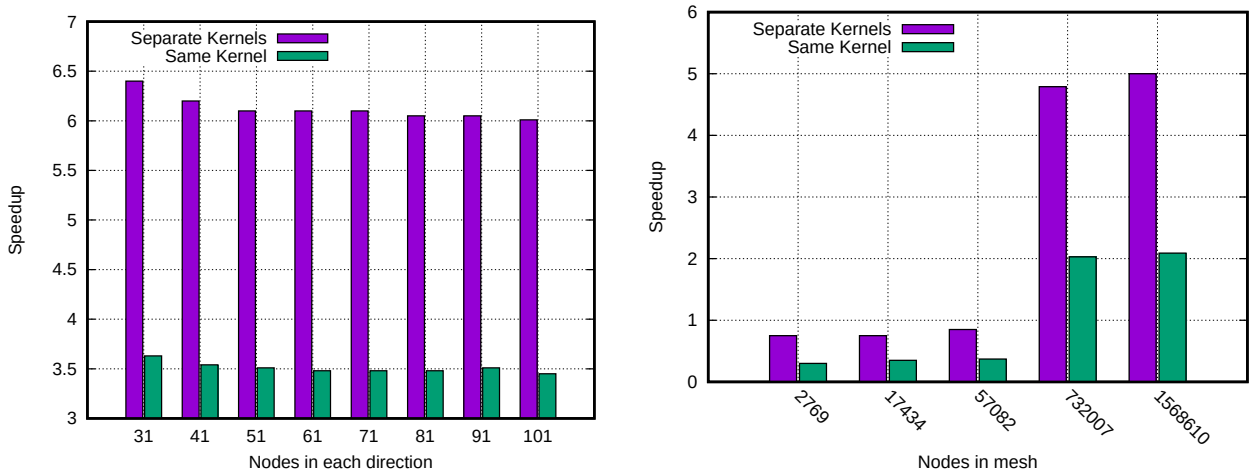
The computation time of assembly using the kernel division strategy is now compared with the SharedNZ implementation of Cecka et al.[3] on GPU. The SharedNZ algorithm is implemented using CUDA in which the kernel and memory utilization remain the same as given in Cecka et al. [2011][3]. In this implementation, each thread is assigned to compute one NZ entry of the global stiffness matrix. The elemental data is stored in the shared memory and a reduction operation is performed on the shared memory to obtain the final NZ entry.

The speedup for the kernel division strategy using ELL storage format over SharedNZ algorithm is shown in figure 12 for HEX20 and TET4 mesh types respectively. The same and separate kernel approaches for numerical integration and assembly



(a) HEX20 (b) TET4

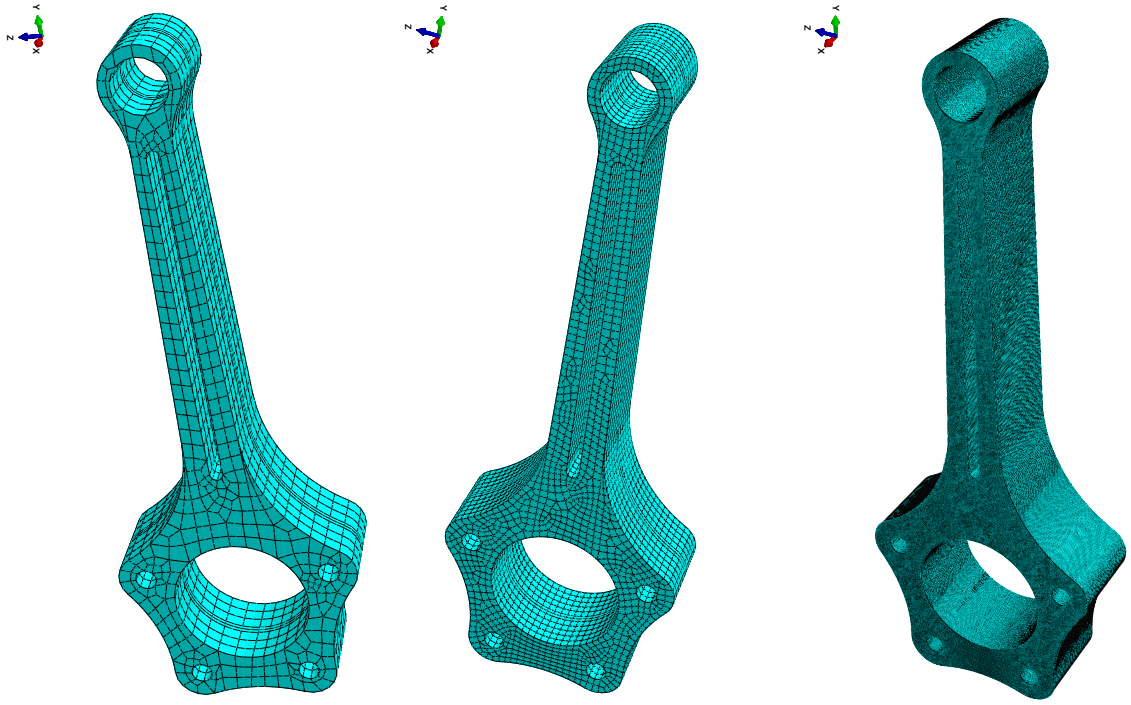
FIGURE 11 Effective bandwidth comparison for (a) HEX20 and (b) TET4 element type for increasing number of nodes in the mesh



(a) HEX20 (b) TET4

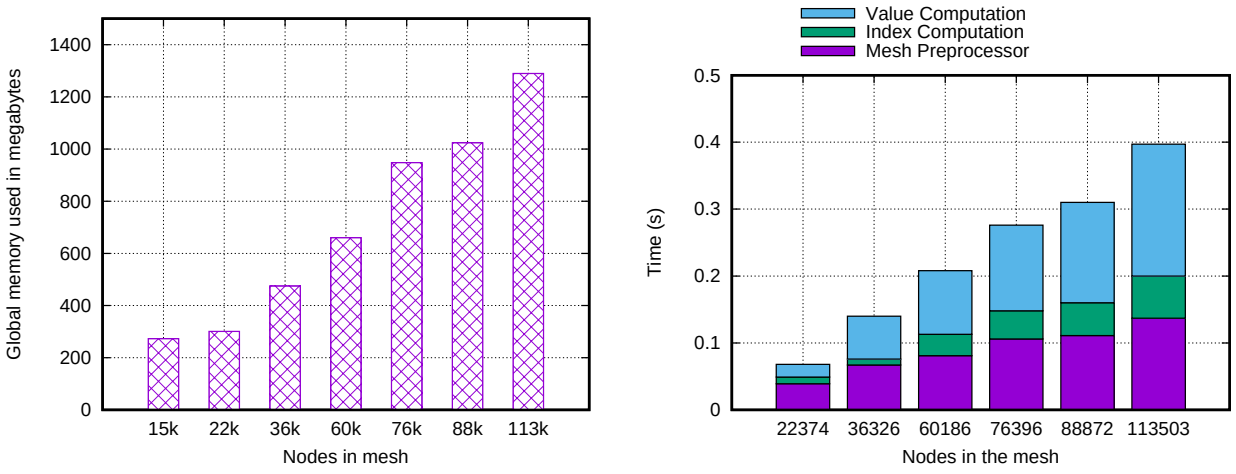
FIGURE 12 Speedup for (a) HEX20 and (b) TET4 type elements using same and separate kernel approach with respect to SharedNZ[3] implementation.

are shown in these figures. It can be seen that both approaches show significant speedup over the SharedNZ implementation in which the separate kernel approach shows speedup close to six for different node numbers for HEX20 mesh type. However, a decent speedup of four can be seen for a large number of nodes with TET4 mesh type. In terms of speedup, HEX20 mesh seems to outperform TET4 mesh for both same kernel and separate kernel approaches for the present example. In table 3, details of the element type, number of nodes and elements, runtime of different stages and global memory usage are listed for the current example.



(a) 15318 nodes (b) 60186 nodes (c) 2020284 nodes

FIGURE 13 Connecting rod mesh using (a) 15318 nodes, (b) 60186 nodes and (c) 2020284 nodes



(a) Global memory(GPU) (b) Execution times

FIGURE 14 (a) Global memory usage and (b) Wall clock time comparison for assembly using HEX20 mesh of connecting rod with increasing number of nodes in the mesh

5.2 Example 2: Connecting Rod

A connecting rod example is considered as the second example for which unstructured mesh is required. Figure 13 shows three different meshes of a connecting rod with different number of nodes. The execution times for different functions in the mesh-preprocessor stage are listed in table 4. Similar to the previous example, the *thrustDynamicSort* kernel is found to have the largest execution time, followed by *fillNodeNum* kernel in all three meshes.

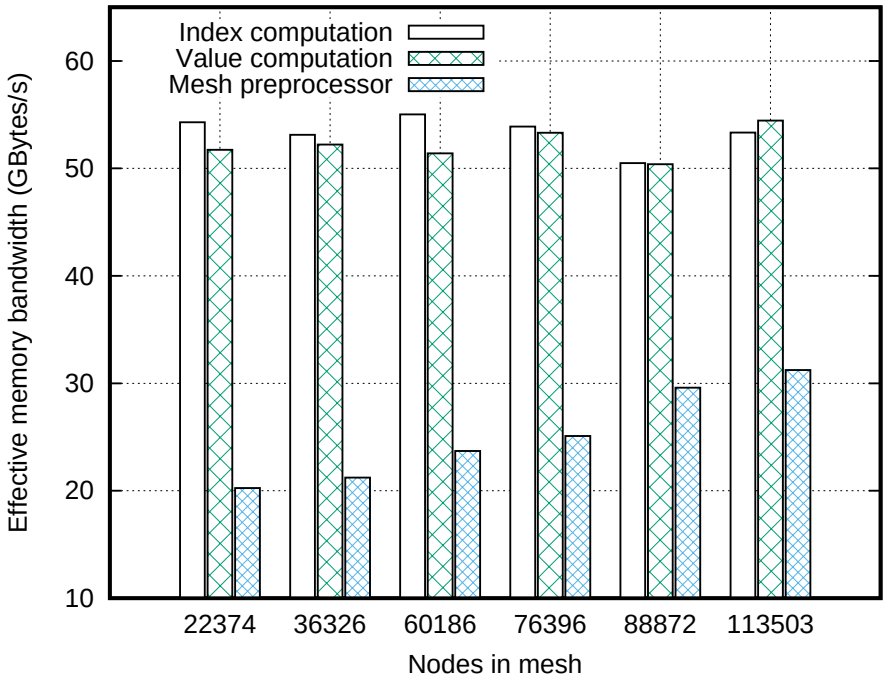
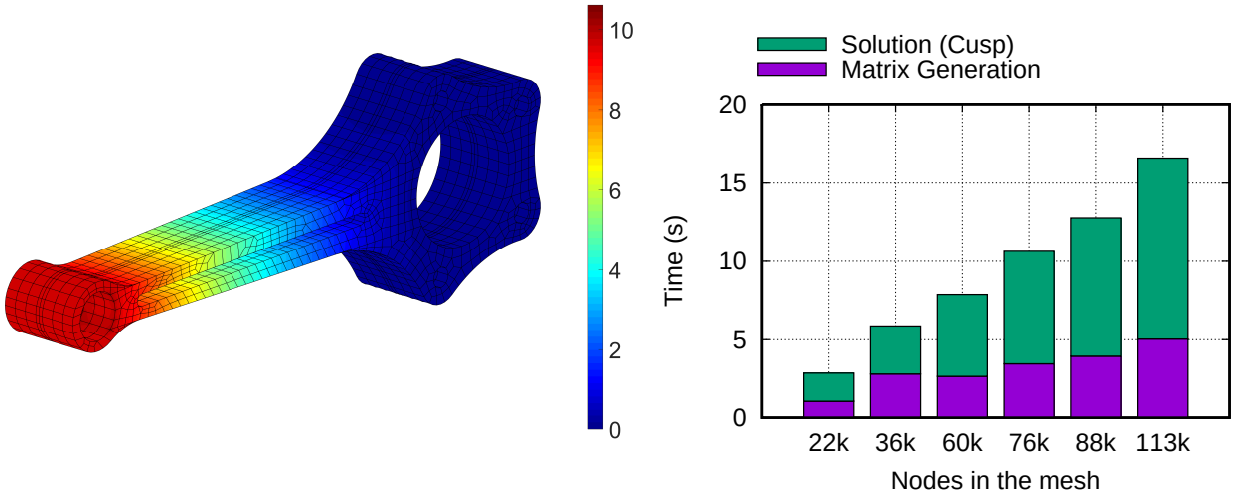


FIGURE 15 Effective bandwidth comparison for assembly using HEX20 mesh of connecting rod with increasing number of nodes in the mesh.

Since the unstructured mesh is used for this example, the amount of global memory used for assembly with increasing number of nodes is shown in figure 14(a). The assembly time is shown figure 14(b) with increasing mesh sizes. Similar to the previous example, the assembly time is further divided into times required by mesh-preprocessor, index computation and values computation. It can be observed again that the index computation stage consumes the least amount of time, followed by the mesh preprocessor for all the mesh sizes. The achieved memory bandwidth for the different mesh sizes of the connecting rod is shown in figure 15. The values for all three individual stages are shown in the plot. The achieved bandwidth values for the index computation and value computation stages are observed to have little variation, whereas it is seen to increase with the mesh sizes for the mesh preprocessor stage.

Cusp library is used for solution of the resulting global stiffness matrix after applying proper boundary conditions. Cusp is a GPU-based library for sparse linear algebra. Conjugate gradient method is used for solution of the resulting matrix on GPU. The resulting displacement field of the connecting rod domain is plotted in figure 16(a). Figure 16(b) shows the total time taken by the FE analysis with increasing mesh sizes. This time is further divided into time required by the matrix generation and solution stage. It is important to mention here that the times consumed for data transfer from the host to the device is very small compared to the total execution time (less than 0.1%) and is thereby omitted from figure 16(b). The reason behind this is that the entire computation, from the elemental stiffness calculation to the solution of the linear system of equations, is executed on the GPU with the help of custom kernels and *thrust* operations. It is observed from figure 16(b) that the matrix generation step consumes approximately 30% - 45% of the total FE analysis time on GPU.

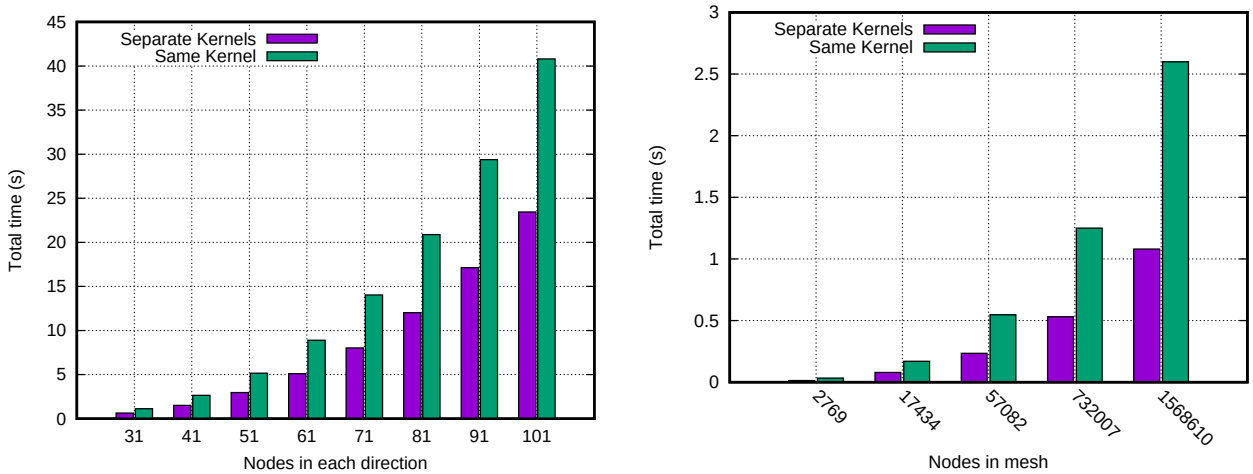
From algorithm 7 of value computation, it can be seen that the elemental stiffness matrices are assembled into the global stiffness matrix. To compute the element stiffness matrices, the numerical integration is required as shown in equation (7). Two approaches have been considered to generate the elemental stiffness matrices as discussed in Section 4 using the same and separate kernel approaches. Figure 17 shows the computation time for both approaches using HEX20 and TET4 mesh types respectively considering ELL format. For both mesh types, the separate kernel approach requires less time for assembly than the same kernel approach. This observation is counter-intuitive, since the same kernel approach obviates the need to read and write the elemental stiffness matrices to the global memory of GPU. However, in the same kernel approach more computational load has been assigned to each of the threads of the GPU and thread synchronization is performed that can make this approach



(a) Displacement field

(b) Total execution time

FIGURE 16 (a)Plot of the displacement field and (b) Total execution time of the application, divided into matrix generation and solution using Cusp library for the connecting rod domain



(a) HEX20

(b) TET4

FIGURE 17 Matrix generation time for (a) HEX20 and (b) TET4 type elements based on position of the numerical integration kernel

inferior than the separate kernel approach. In table 5, details of the element type, number of nodes and elements, runtime of different stages and global memory usage are listed for the connecting rod example.

6 Conclusion

A three-stage FEA matrix generation strategy has been presented for unstructured meshes on GPU. Numerical integration and assembly operations have been implemented for same kernel and separate kernel approaches. This step of FEA was chosen for acceleration because of several studies reporting it to be the most time-consuming step after the linear solver, for which several optimized parallel libraries exist. A novel approach of dividing the workload into a mesh-preprocessor, index computation and

values computation was presented and simulations were run for meshes having up to 3 million DOFs on a single GPU with two different element types.

In order to successfully decouple the index and value computation stage of matrix generation, a new data structure called as the neighbor matrix was proposed to store the neighboring nodes for each node of the mesh during the first stage of the matrix generation. A robust parallel implementation of the neighbor matrix computation was presented that can be adopted easily for different element and mesh types. Using the neighbor matrix, the indices and values of NZ entries were computed in the second and third stage respectively. For these two stages, a generalized algorithm with different sparse formats was presented.

The proposed strategy was tested on two examples and it was found to outperform the sharedNZ implementation of Cecka et al.[3], showing a speedup of 4× to 6× for all element types over a wide range of mesh sizes. Higher order hexahedral elements are found to achieve more speedup than the low order tetrahedral elements on a particular benchmark. The mesh preprocessor stage was found to consume a very less amount of time with respect to the total assembly time. The index computation kernel was found to consume significantly less percentage of total assembly especially for the lower order element. It was also concluded that performing assembly and numerical integration in the same kernel performed worse than having separate kernels for the two processes. This difference is more pronounced for lower order elements. Since Cusp library was used for generating the solution, the matrix-free approaches can be implemented on the GPU with the help of the mesh preprocessor in future.

Although the present work focuses only on single GPU implementations, the same approaches can be utilized for a multi-GPU implementation as well in future work. For such an implementation, the finite element mesh needs to be divided into independent patches for distributing them among the GPUs as shown in figure 18. This can be done easily by using a domain decomposition technique. For the mesh preprocessor stage and the index computation stage, no dependency among the GPUs are observed. Hence, no inter-GPU communication is required for these stages. For the value computation stage, however, the boundary nodes and boundary elements need to be treated differently. This is due to sharing of nodes along the boundary of different patches as shown in figure 18, which requires values at the same non-zero locations to be appended from different GPUs.

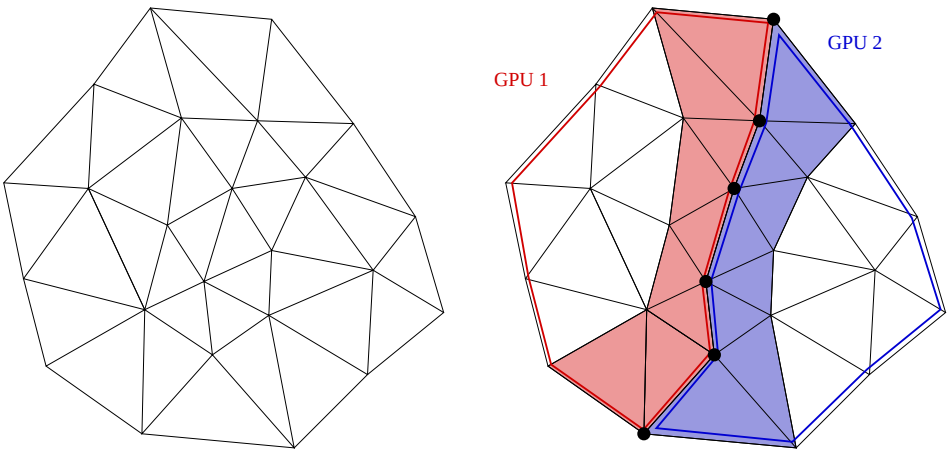


FIGURE 18 Shared nodes and elements in a multi-GPU implementation

Conflict of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

References

1. Zienkiewicz OC, Taylor RL, Lee R. *The finite element method*. 3. McGraw hill London . 1977.

2. Georgescu S, Chow P, Okuda H. GPU Acceleration for FEM-based Structural Analysis. *Archives of Computational Methods in Engineering* 2013; 20(2): 111-121. doi: 10.1007/s11831-013-9082-8

3. Cecka C, Lew AJ, Darve E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering* 2011; 85(5): 640–669. doi: 10.1002/nme.2989

4. Maciol P, Plaszewski P, Banaś K. 3D finite element numerical integration on GPUs. *Procedia Computer Science* 2010; 1(1): 1093 - 1100. ICCS 2010doi: http://dx.doi.org/10.1016/j.procs.2010.04.121

5. Ram L, Sharma D. Evolutionary and GPU computing for topology optimization of structures. *Swarm and Evolutionary Computation* 2017; 35: 1–13.

6. Komatitsch D, Michéa D, Erlebacher G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing* 2009; 69(5): 451 - 460. doi: http://dx.doi.org/10.1016/j.jpdc.2009.01.006

7. Banaś K, Krużel F, Bielański J. Finite element numerical integration for first order approximations on multi- and many-core architectures. *Computer Methods in Applied Mechanics and Engineering* 2016; 305: 827 - 848. doi: http://dx.doi.org/10.1016/j.cma.2016.03.038

8. Knepley MG, Terrel AR. Finite Element Integration on GPUs. *CoRR* 2011; abs/1103.0066.

9. Fu Z, Lewis TJ, Kirby RM, Whitaker RT. Architecting the finite element method pipeline for the GPU. *Journal of Computational and Applied Mathematics* 2014; 257: 195 - 211. doi: https://doi.org/10.1016/j.cam.2013.09.001

10. Reguly IZ, Giles MB. Finite Element Algorithms and Data Structures on Graphical Processing Units. *International Journal of Parallel Programming* 2015; 43(2): 203–239. doi: 10.1007/s10766-013-0301-6

11. Altinkaynak A. An efficient sparse matrix-vector multiplication on CUDA-enabled graphic processing units for finite element method simulations. *International Journal for Numerical Methods in Engineering* 2017; 110(1): 57–78.

12. Challis VJ, Roberts AP, Grotowski JF. High resolution topology optimization using graphics processing units (GPUs). *Structural and Multidisciplinary Optimization* 2014; 49(2): 315-325. doi: 10.1007/s00158-013-0980-z

13. Schmidt S, Schulz V. A 2589 line topology optimization code written for the graphics card. *Computing and Visualization in Science* 2011; 14(6): 249-256. doi: 10.1007/s00791-012-0180-1

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

15. Kiss I, Gyimothy S, Badics Z, Pavo J. Parallel Realization of the Element-by-Element FEM Technique by CUDA. *IEEE Transactions on Magnetics* 2012; 48(2): 507-510. doi: 10.1109/TMAG.2011.2175905

16. Bolz J, Farmer I, Grinspun E, Schröoder P. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.* 2003; 22(3): 917–924. doi: 10.1145/882262.882364

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

18. Cantwell C, Sherwin S, Kirby R, Kelly P. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids* 2011; 43(1): 23 - 28. Symposium on High Accuracy Flow Simulations. Special Issue Dedicated to Prof. Michel Devilledoi: https://doi.org/10.1016/j.compfluid.2010.08.012

19. Dziekonski A, Sypek P, Lamecki A, Mrozowski M. Finite element matrix generation on a GPU. *Progress In Electromagnetics Research* 2012; 128: 249–265.

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

21. Dinh Q, Marechal Y. Toward Real-Time Finite-Element Simulation on GPU. *IEEE Transactions on Magnetics* 2016; 52(3): 1-4. doi: 10.1109/TMAG.2015.2477602

22. Sanfui S, Sharma D. A two-kernel based strategy for performing assembly in FEA on the graphics processing unit. In: AMIAMS ., ed. *2017 International Conference on Advances in Mechanical, Industrial, Automation and Management Systems (AMIAMS)*; 2017: 1-9

23. Zayer R, Steinberger M, Seidel H. Sparse matrix assembly on the GPU through multiplication patterns. In: HPEC ., ed. *2017 IEEE High Performance Extreme Computing Conference (HPEC)*; 2017: 1-8

24. Kiran U, Sharma D, Gautam SS. GPU-Warp based Finite Element Matrices Generation and Assembly using Coloring Method. *Journal of Computational Design and Engineering* 2018. doi: https://doi.org/10.1016/j.jcde.2018.11.001

25. Gribanov I, Taylor R, Sarracino R. Parallel implementation of implicit finite element model with cohesive zones and collision response using CUDA. *International Journal for Numerical Methods in Engineering* 2018; 115(7): 771-790. doi: 10.1002/nme.5825

26. Sanfui S, Sharma D. Exploiting Symmetry in Elemental Computation and Assembly Stage of GPU-Accelerated FEA. In: G. R. Liu GX., ed. *Proceedings at the 10th International Conference on Computational Methods (ICCM2019)*. 6. ScienTech Publisher; 2019: 641–651.

27. Woźniak M. Fast GPU integration algorithm for isogeometric finite element method solvers using task dependency graphs. *Journal of Computational Science* 2015; 11: 145–152.

28. WoÅžniak M, KuÅžnik K, PaszyÅĎski M, Calo V, Pardo D. Computational cost estimates for parallel shared memory isogeometric multi-frontal solvers. *Computers & Mathematics with Applications* 2014; 67(10): 1864 - 1883. doi: https://doi.org/10.1016/j.camwa.2014.03.017

TABLE 1 The mesh preprocessing time for different element types

Element type	Elements	Nodes	Time taken (s)
TET4	732007	130884	0.201
TET10	732129	1012919	0.813
HEX20	175550	742677	0.110

TABLE 2 Time taken by different functions in the pre-computation stage in seconds (hollow cylinder)

Function	30000 Nodes	1000000 Nodes	2000000 Nodes
countElemNum	0.000051	0.000769	0.001533
Inclusive Scan	0.000085	0.000189	0.000301
fillElemNum	0.000049	0.001597	0.004856
Inclusive Scan	0.00008	0.00018	0.000294
initializeNodeNum	0.000033	0.000559	0.001105
fillNodeNum	0.000621	0.036936	0.078467
thrustDynamicSort	0.041134	0.786364	1.560426
Inclusive Scan	0.000081	0.000185	0.000209

TABLE 3 Element type, number of nodes, number of elements, runtime and memory used for storing the global stiffness matrix on the GPU global memory are presented for example 1. S1, S2 and S3 under the runtime column denote mesh preprocessor, index computation and value computation stages respectively.

Element type	Nodes	Elements	Runtime (s)			Memory (MBs)
			S1	S2	S3	
HEX20	68921	64000	0.07	0.03	0.05	44.67
	132651	125000	0.10	0.06	0.13	85.95
	226981	216000	0.15	0.11	0.23	147.08
	357911	343000	0.20	0.17	0.35	231.92
	531441	512000	0.30	0.26	0.55	344.37
	753571	729000	0.43	0.38	0.78	488.31
	1030301	1000000	0.54	0.52	1.07	667.63
TET4	17434	90630	0.04	0.00	0.03	3.35
	57082	311963	0.09	0.01	0.10	10.96
	130884	732007	0.20	0.03	0.22	27.22
	275913	1568610	0.36	0.07	0.49	57.39

□

TABLE 4 Time taken by different functions in the pre-computation stage in seconds (connecting rod)

Function	15318 Nodes	60186 Nodes	2020284 Nodes
countElemNum	0.000060	0.000081	0.018002
Inclusive Scan	0.000052	0.000089	0.000149
fillElemNum	0.000055	0.000149	0.024251
Inclusive Scan	0.000068	0.000058	0.000154
initializeNodeNum	0.000045	0.000108	0.007312
fillNodeNum	0.001129	0.006801	0.678876
thrustDynamicSort	0.032785	0.076183	9.114569
Inclusive Scan	0.00007	0.000063	0.000148

TABLE 5 Element type, number of nodes, number of elements, runtime and memory used for storing the global stiffness matrix on the GPU global memory are presented for example 2. S1, S2 and S3 under the runtime column denote mesh preprocessor, index computation and value computation stages respectively.

Element type	Nodes	Elements	Runtime (s)			Memory (MBs)
			S1	S2	S3	
HEX20	22374	4134	0.04	0.01	0.02	14.49
	36326	13958	0.07	0.01	0.06	23.54
	60186	11988	0.08	0.03	0.10	39.00
	76396	15483	0.12	0.04	0.13	49.50
	88872	18163	0.11	0.05	0.15	57.59
	113503	23547	0.14	0.06	0.20	74.54