A Two-Kernel based Strategy for Performing Assembly in FEA on the Graphic Processing Unit

Subhajit Sanfui and Deepak Sharma
Department of Mechanical Engineering
Indian Institute of Technology Guwahati
Assam, India, Pin-781039
Email: {s.sanfui, dsharma}@iitg.ernet.in

***Abstract*—This paper presents a strategy to perform assembly of system of equations arising in Finite Element Analysis (FEA) on Graphics Processing Units (GPU) based on the principle of dividing the workload into separate kernels. Three different sparse formats are analyzed for efficient storage along with two different implementations for the race condition arising in the traditional assembly (addto method). In the present study, ELL sparse storage format is found to be effective in terms of both storage space and performance on the GPU. To avoid race condition, the method of coloring is found to provide superior result in comparison to the method with atomics for the proposed assembly strategy.**

***Keywords*-FEA; Assembly; GPGPU; Sparse Storage;**

I. Introduction

Finite Element Method (FEM) is a numerical method for approximating solutions of boundary value problems for partial differential equations (PDEs). It has been used in many fields such as mechanical engineering, electrical engineering, medical applications to name a few, where a solution to PDEs is sought. Due to its natural benefits over prior approaches, it has become an integral part of a large variety of specializations related to mechanical engineering. In industries like aviation, automotive and construction, FEM is usually an inherent part of the computer aided design and engineering (CAD/CAE) process [1]. The process of applying concepts of FEM for the analysis of physical systems is often referred to as "Finite Element Analysis (FEA)". A typical FEA consists of three basic steps.

1) *Pre-processor Step* - In this step, the problem domain is defined by specifying the material properties, type of element, nodal coordinates and connectivity.
2) *Solver Step* - This step assembles algebraic equations for each element in matrix form and computes the values of the primary unknowns after application of proper boundary conditions.
3) *Post-processor Step* - In this step, the values of the primary unknowns are used to extract several useful information about the problem domain, for example stresses, strains etc. Plotting of relevant data is also performed at this stage.

It has been observed that among all these steps in FEA, generation of elemental stiffness data, assembly and solution of the resulting matrix in the solver stage are often computationally expensive in nature, [2], [3], [4] making it a suitable candidate for GPU implementation. However, the sparsity pattern in the global matrices after assembly is often highly irregular, making it quite difficult for an efficient parallel implementation. It has been demonstrated that in order to efficiently port FEA on the GPU, an implementation, radically different to that on the CPU is required [5]. Computation of elemental stiffness matrix for higher order elements [6] or for unstructured grid become especially time-consuming due to the repetitive calculation over all the elements. Assembly of the elemental matrices in many practical applications become particularly time-consuming due to a large mesh size. Typically in the literature, applications performing FEA on the GPU, tend to implement matrix-free methods, obviating the need to assemble or store the complete global matrices [7], [8]. The key principle in the matrix-free approach is that instead of performing sparse matrix-vector multiplication (SpMV) after assembling the global data, SpMV may be applied to the elemental matrices and the results may be assembled afterwards. In [6], numerical simulation of seismic wave propagation, resulting from earthquakes is implemented on an NVIDIA GPU. To avoid the race condition during assembly in FEA, the elements are colored such that no neighboring elements share the same color. Following this, assembly is performed for each color in a sequential manner. In [3], several strategies are proposed to perform assembly on an unstructured grid in two dimensions by decomposing the task into independent blocks of work. Assembly is performed by calling a subroutine that computes the elemental data for each element of the mesh. Each thread is assigned to perform a different task for different strategies. Race condition is avoided by assigning each thread to one NZ entry or implementing a coloring scheme similar to the one used in [6]. Authors found the coloring approach to be inefficient due to excessive global memory transactions. In [5], assembly is discussed through two different approaches, namely the local matrix approach (LMA), and the addto method. The addto method is the traditional assembly method, whereas, LMA is a matrix free method designed to alleviate difficulties in implementing addto on the GPU architecture. The authors concluded that LMA and addto can be efficient for GPU and CPU implementations respectively. In [9], an efficient technique for generation of

global system of equations is presented for application in the field of computational electromagnetics. The authors proposed a strategy to first assemble the stiffness matrix into coordinate format (COO) and later convert it into compressed sparse row (CSR) format for further processing.

The solution of the linear system of equations may be performed by direct and iterative methods. Typically in case of matrices of large dimensions, iterative methods such as Conjugate Gradient Methods are preferred [3]. In the present work, we concentrate our focus on structured grids with low order elements, where the assembly and solution of the system of equations are performed on the GPU.

The contributions of this paper are summarized as

- Proposing a strategy to divide the assembly task into different kernels for efficient implementation on the GPU.
- Customization of the proposed strategy in context of three different sparse storage formats.
- Proposing a new approach of implementing coloring with the proposed assembly method to avoid the race condition and its comparison with a low-wait algorithm with atomics.

In section II, a brief overview of the graphics processing units is presented. Section III outlines the formulation of FEM for a linear elasticity problem. In section IV and V, the assembly and solution of the system of equations are discussed respectively. Results are discussed in section VI and finally the paper is concluded in section VII.

II. GPU Architecture & CUDA

Graphics Processing Units are specialized circuits which, although primarily designed for rendering graphics, have been largely implemented for accelerating computation arising in various research, scientific, analytical and other applications. A number of platforms such as OpenCL, OpenMP and CUDA are derived over the past several years for parallelizing general purpose applications. For the present implementation, CUDA, a parallel computing platform and API developed by NVIDIA, has been used. CUDA provides the architecture and programming model that accommodates both the host (CPU) and device (GPU) simultaneously. The device code is written using particular extensions to the C programming syntax, inside special functions termed as Kernels. These Kernels may generate grids of thousands or even millions of threads to parallelize a given task instead of running in a sequential manner on the CPU. A schematic representation of the data flow in a hybrid computing environment is presented in figure 1 For details of the architectural model and programming syntax, interested readers may refer to [10]. The contributions of this paper is in line with the trend that utilizes GPUs to accelerate general purpose applications. Examples include solution of PDE [11], [12], [13], Image Processing [14], [15], Molecular Dynamics [16], [17], Weather simulation [18], [19], Modeling and simulation of complex phenomenon [20], [21] and more. Following is the glossary of terms for GPU architecture and CUDA.

- **Host** is the CPU.

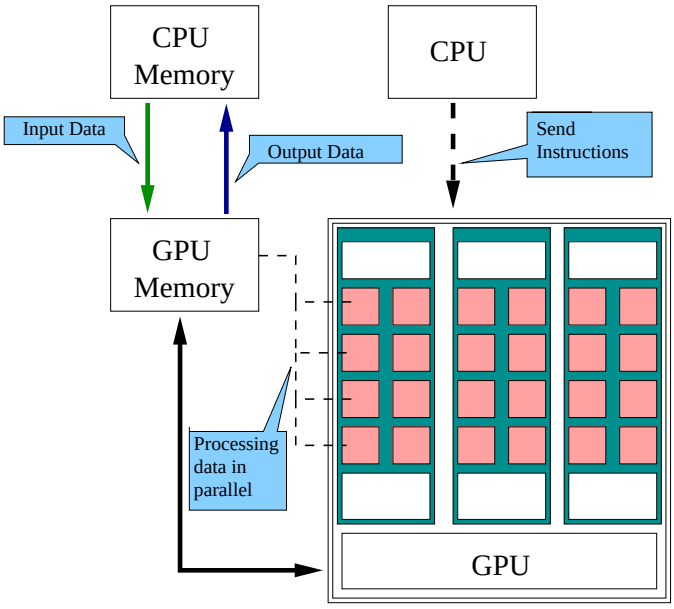

Fig. 1. Schematics of data processing on the GPU

- **Device** is the GPU.
- **Kernel** is a function executed on GPU.
- **Thread** is a unit of computation.
- **Block** is a collection of threads.
- **Grid** is a collection of blocks.
- The code is actually executed in groups of 32 threads, which is called a **warp**.
- **Global Memory** is memory that all threads can access at any time.
- **Shared Memory** is on-chip memory that all threads in a block can access.

III. FEM Formulation for Linear Elasticity Problem

The response of a mechanical system to certain loading conditions is governed by partial differential equations which may be derived from the basic laws of physics. However solving these equations analytically often pose a great challenge due to complexity and size of the domain, non-linear behavior etc. By the use of FEA, the complex and large domain is broken up into smaller and simpler elements called finite elements as shown in figure 2 and certain functions are used to approximate the behavior of these elements. Algebraic equations governing the response of these elements are assembled in matrix form to approximate the behavior of the entire domain. In this section the formulation for FEM of a linear Elasticity problem for a three dimensional system is presented.

A. Linear Elasticity

Linear elasticity is the study of deformation and stresses in solid bodies subjected to certain loading conditions. The governing equations of a linear elasticity boundary value problem constitute of three equilibrium equations from Newton's

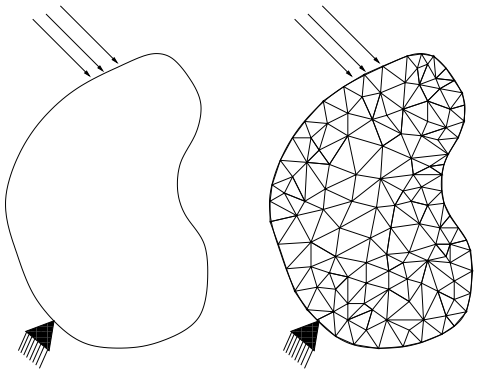

Fig. 2. Typical Finite Element Mesh

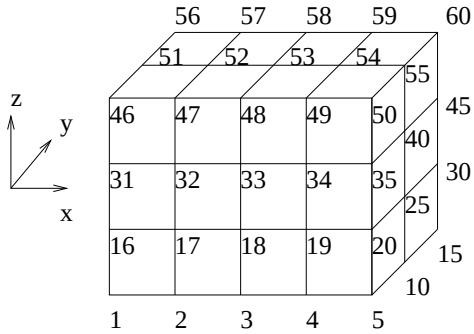

Fig. 3. Discretization and global numbering scheme

second law of motion, six strain-displacement relations and six constitutive equations, which are given as [1],

$$\sigma_{ij,j} \; + \; b_i \; = \rho \ddot{u}_i, \qquad i,j = 1,2,3, \tag{1}$$

$$\epsilon_{ij} = \frac{1}{2}(u_{j,i} + u_{i,j}), \tag{2}$$

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl}. \tag{3}$$

Here, b_i are the body forces per unit volume, ρ is the mass density, u_i are the displacements, σ_{ij} are the stresses, ϵ_{ij} are the strains and C_{ijkl}, the Cauchy stress tensor.

Equation (1) may be subjected to the essential and natural boundary condition as,

$$u_i = \bar{u}_i \; on \; \Gamma_u \qquad\qquad t_i = \bar{t}_i \; on \; \Gamma, \tag{4}$$

where, t_i are the boundary traction forces on boundary Γ. Γ_u is the portion of the boundary Γ, where the displacements are specified.

B. Formulation

The basic idea behind linear finite element formulation is to linearize the weak form of the governing equations of the problem and to solve these equations over the discretized domain [1].

The displacements of one 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. To construct a weak form of the governing equation, we can multiply equation (1) with virtual displacement δu_i and integrate over the domain Ω. Gauss Divergence theorem may be applied to the resulting equation calculated over all elements of a discretized domain and simplified to 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. Since, for a static problem, the first term in equation (6) is considered zero. Interested readers may refer to [1] for a more detailed formulation.

C. Assembly

For carrying out finite element analysis, we discretize 3D problem domain into a structured grid by using cubic hexahedron elements as shown in figure 3. The global node numbering scheme adopted in this work, is also shown in figure 3.

The eight shape functions are given by,

$$[N_i]^e = \frac{1}{8}(1 + \xi\xi_i)(1 + \eta\eta_i)(1 + \zeta\zeta_i), \tag{7}$$

where, ξ_i, η_i and ζ_i denote coordinates of the i^{th} node. The elemental stiffness matrix is given as [1]

$$\begin{aligned}
[K]^e &= \int_V [B]^T [D][B] \; dxdydz \\
&= \int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} B(\xi,\eta,\zeta)^T DB(\xi,\eta,\zeta) \mid J(\xi,\eta,\zeta) \mid \\
&\qquad\qquad d\xi d\eta d\zeta \\
&= \sum_{i=1}^{20}\sum_{j=1}^{20}\sum_{k=1}^{20} w_{\xi_i} w_{\eta_j} w_{\zeta_k} B(\xi_i,\eta_j,\zeta_k)^T DB(\xi_i,\eta_j,\zeta_k) \\
&\qquad\qquad \mid J(\xi_i,\eta_j,\zeta_k) \mid
\end{aligned} \tag{8}$$

Here, B is the matrix that contains partial derivatives of the shape functions with respect to x, y and z. J and D are the Jacobian and constitutive matrix respectively. w_{ξ_i}, w_{η_j} and w_{ζ_k} are the weights for carrying out numerical integration using Gauss Quadrature(GQ) method. In this case, a 20 point GQ method is used.

For a cubic hexahedron element, the size of $[K]^e$ matrix is 24×24. Upon assembling all $[K]^e$, the global stiffness $[K]$ matrix is constructed. In the following section, assembly is discussed in detail.

IV. Assembly on GPU

A. Assembly

The process of assembly is essentially an additive operation, where the elemental matrix is computed as explained in equation (8) and based on its position on the problem domain, the matrix is added to the global stiffness matrix [K]. Assembly is typically carried out in an element-by-element fashion, where, for each element of the mesh, the target indices

of the global stiffness matrix are computed and the elemental matrix is added to the values at the corresponding indices. For a structured grid, since the elemental stiffness matrix is same for all the elements, it may be computed only once. Because of the highly sparse nature of the global stiffness matrix, it is typically stored using specialized sparse formats such as COO and CSR [5], [9].

For implementation of FEA on GPU, two different strategies of assembly have been used in the literature, namely LMA and addto. In [5], LMA is concluded to perform better for GPU implementations and addto method for CPU implementations. Although, the addto method is massively data-parallel because all the iterations of the outer loop can be executed independently, it is found to be inefficient on the GPU. Due to the possibility of parallel threads writing to the same memory location, expensive coloring or atomics has to be implemented to avoid race condition. Another issue that hampers performance of the addto method is the bisection search for locating the key from the global stiffness matrix in sparse format. This leads to massive amount of uncoalesced accesses and thread divergence within warps leading to decreased performance. LMA is considered to be an effective data-parallel alternative because it reduces the storage requirements of the global stiffness matrix while taking care of race condition. The downside of using LMA instead of addto method is that it increases computation and memory bandwidth of the application proportional to the variance of the mesh [5]. In the present work, a modified addto method is implemented that takes care of the race condition and costly bisection search by dividing the operation into two different kernels to achieve performance.

Instead of the traditional assembly approach, where the indices and values of the sparse global matrix are computed and stored at the same time, a different approach is followed by dividing the workload into two kernels on the GPU. The first kernel is responsible for computing the indices only. Since this computation is based only on the geometry of the domain, this kernel consumes only a fraction of the total computation time as discussed later in the results section. After the indices are computed and stored onto the global memory, a second kernel is invoked that computes the non-zero (NZ) values based on the corresponding indices. The key idea behind this approach to assembly is that for a structured mesh, computation of NZ indices of the global stiffness may be performed based on the connectivity information alone, and hence can be performed separately from the actual assembly operation. This reduces the complexity of the bisection search drastically. Instead of the entire length of the arrays, bisection search is applied to a reduced target zone (of length varying from 24 to 81), thereby reducing the computation time significantly. Two efficient strategies are discussed later for avoiding the race condition inherent in the addto method.

To perform assembly on the GPU, the following data is required.

1) The elemental stiffness matrix ($[K]^e$).
2) Connectivity/Domain matrix containing the global node numbers (connectivity/domain).
3) Matrix containing the NZ indices corresponding to all nodes (NZIndex).

Each thread block has access only to the data corresponding to the elements and nodes assigned to it.

To make use of one of the sparse formats for storing the global stiffness matrix, the exact number of non-zero elements must be known beforehand. Although this requirement is not stringent in case of traditional assembly on CPU, it becomes necessary for allocating memory to the device pointers and to determine the configuration parameter of the kernels. Hence, an expression is derived analytically from the geometry of the problem domain to provide the exact number of NZ in the global stiffness matrix for a three dimensional structured mesh containing cubic hexahedron elements of equal size. This expression is derived by subtracting the number of repeated write operations from the total write operations during assembly calculated from the connectivity information of the domain. Similar expressions may be derived for other geometries with the same principle for calculating the number of NZ beforehand. If x, y and z be the number of nodes in the x, y and z directions respectively, the number of NZ is found to be given by

$$NZ = 108(x+y+z) - 162(xy+yz+zx) + 243xyz - 72 \quad (9)$$

At the beginning of the application, memory allocation and initialization of the required arrays based on the number of NZ calculated from (9) is performed. Following this, the assembly task is divided into two separate kernels.

1) Kernel 1: This kernel is responsible for computing the row and column indices of the NZ entries but not the values. Each node in the problem domain corresponds to three rows and three columns of the global stiffness matrix. One thread is assigned to each of the nodes of the mesh. The idea behind this kernel is that the number of NZ in a particular row can be calculated by multiplying the degrees of freedom per node with the number of neighboring nodes of the node corresponding to the particular row. Figure 4 shows the number of neighboring nodes of corner, edge, face and interior nodes to be 7, 11, 17 and 26 respectively. A node that has n neighboring nodes, will have $3(n+1)$ non-zero entries in the corresponding rows of the global stiffness matrix. Since nodal computations are not shared among any threads, no race condition is observed. The basic steps followed in this kernel for all the approaches are as follows,

- Based on the position of the node (interior, edge, corner or face), row indices are filled in the respective array.
- A set is formed for each node containing the immediate neighboring node numbers.
- Based on the neighboring set, the column indices are computed and filled in the respective array.

2) Kernel 2: This kernel computes the values of the NZ entries in the global stiffness matrix based on the indices calculated in the previous kernel. An element by element assembly strategy is adopted for the implementation. Since,

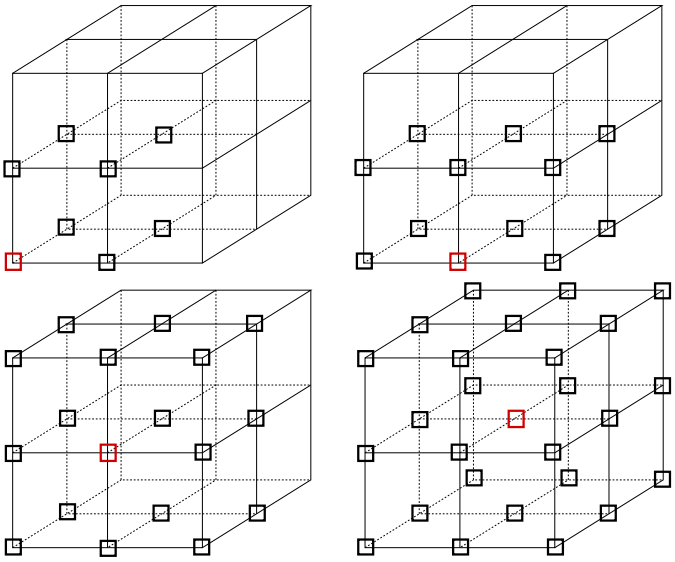

Fig. 4. Neighboring nodes of corner, edge, face and interior nodes

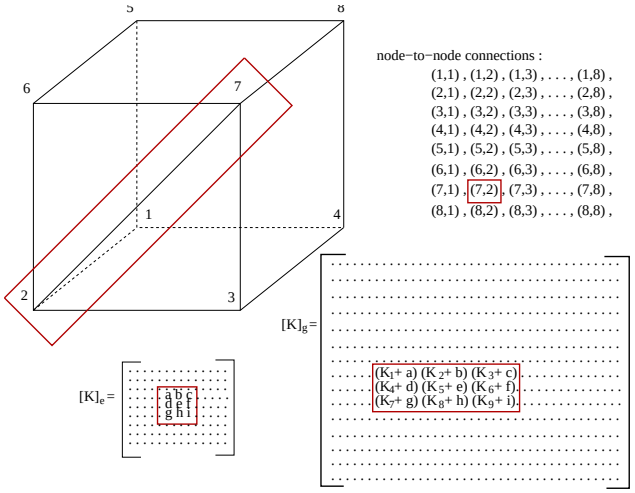

Fig. 5. Assembly Procedure in Kernel 2

several threads are responsible for computing the same shared node, race condition must be taken care of. Two different approaches are presented for the same. For the approach with atomic operations, one thread is assigned to one element of the mesh, whereas, for the coloring approach, this kernel is broken into eight different kernels and eight threads are assigned to each element of the finite element mesh. Each element has eight nodes, and hence $8 \times 8 = 64$ *node-to-node connections*. Each of these connections writes 9 entries into the value array. The basic steps of this kernel are as follows,

- Computing row index, column index and value for each entry of a connection to be written.
- Finding out the target range of row indices for each *connection*.
- Finding the column index within the corresponding range in the column array by bisection search.
- Adding the value of the elemental stiffness entry to the particular global stiffness entry.

Figure 5 shows a typical diagonal node-to-node connection. After locating the 9 target indices using bisection search, entries in $[K]^e$ corresponding to this connection are added to the target indices of $[K]$.

B. Sparse Formats

After the process of assembly, the resulting global stiffness matrix is in general sparse in nature. To take advantage of the sparsity while solving the system of equations as well as to cut down on the storage requirements, the assembled matrix is stored using one of the specialized sparse formats. In the existing literature COO and CSR formats are primarily preferred [3], [9]. Three different formats are analyzed in the present work and corresponding assembly strategies are developed.

1) Assembly using COO format: COO or coordinate format is the most basic sparse matrix storage format. It is typically very suitable for assembly on the GPU [9]. It consists of three one dimensional arrays for storing the row indices, column indices and the values of the NZ entries. The algorithm for this format is as follows,

- Kernel 1 stores the portion of the domain matrix required by each particular thread-block to its corresponding shared memory.
- With the help of the data stored in shared memory, each thread block computes the row and column indices and writes into global memory.
- Kernel 2 stores the portion of NZIndex and Connectivity matrix required by each thread block into its corresponding shared memory. The elemental stiffness is also stored into the shared memory if required.
- Each thread performs a bisection search within the target indices computed from sh_NZIndex array.
- Required column index is located and the value is added to the corresponding entry in the value array.

2) Assembly using CSR/CSC format: CSR and CSC formats are similar to the COO format, except instead of either the row or column indices, the row or column offsets are recorded. This reduces the storage requirements for either the row (CSR) or column (CSC) array. Since the global stiffness matrix is symmetric in nature, both CSR and CSC formats essentially yield the same result for the present implementation. The process of directly assembling into CSR format is found to be inefficient [9] as compared to COO. However, in the present implementation of the addto method, by dividing the assembly task effectively, similar performance from from CSR method has been extracted compared to COO. The algorithm of assembly into CSR is also similar to that of COO. It only differs in the step where Kerenl 1 stores the row indices or column indices depending on CSR or CSC. The number of write operations performed is equal to $(number\ of\ rows + 1)$ instead of the number of NZ entries for this array.

3) Assembly using ELL format: To the best of the authors' knowledge, this format has not been implemented for assembly

of finite element matrices on the GPU prior to the present work. ELL format stores the NZ entries using two dense matrices, each containing the same number of rows as the original matrix and columns equal to the maximum number of NZ in any row of the original matrix. One matrix stores the column indices and the other stores the values of the NZ entries. Due to the column major ordering of this format, coalesced accesses to the global memory is ensured. Also, since this format is already structured, NZIndex array is not required by the threads to compute the indices. This reduces the shared memory and register requirements by almost 30 percent resulting in a higher performance. The algorithm is as follows

- Kernel 1 stores the portion of the domain matrix required by each particular thread-block to its corresponding shared memory.
- With the help of the shared data, the column indices are calculated and written to the Index array.
- Kernel 2 stores the required portion of the connectivity matrix into the shared memory.
- Based on the shared data, the values are computed and written to the corresponding index of the value matrix.

In the proposed method of assembly, the effective time complexity of the binary search algorithm is reduced drastically. The time complexity of a binary search algorithm is $O(log\ n)$. In case of traditional addto method, the value of n is equal to the number of NZ in $[K]$. By dividing the assembly task into separate kernels, the value of n is reduced to a value between 24 & 81 for cubic hexahedron elements. In the traditional addto method, after performing binary search, if the index is not present in the index array, it is appended to the end of the array and the value is written to the corresponding index in the values array. If the index is already present, the value simply is added to the value at the corresponding index. This creates high control divergence within warps. The modified addto method reduces divergence by dividing the kernels.

C. Strategies to avoid race condition

A race condition is a situation where the output becomes dependent on the order or timing of a number of uncontrollable events. This is a common occurrence in many studies concerning FEM on GPU. A number of strategies are implemented in the literature to avoid race condition. Examples include, assigning each NZ to one thread[3], mesh coloring[3][6], atomic operations[9], assembly-free methods [5] etc. In [9] race condition is avoided by making multiple copies of the indices where more than one write operations are to be performed. For each write a different *copy* of the index is selected. This ensures that no two parallel threads write to the same memory location. Later, a reduction operation is performed for the repeated indices using atomics, while converting into CSR format. This method, although relatively simple, poses problems for the present implementation because of the high number of repeated writes. In Kernel 1, each node is assigned to each thread, and there is no *s*haring of nodes among any thread. So, no race condition is observed. However,

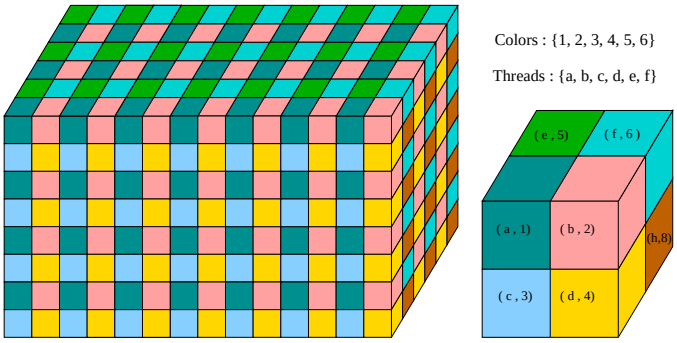

Fig. 6. Coloring Scheme

in Kernel 2, each thread is assigned to an element of the finite element mesh. Since all the elements share a number of nodes with one or more neighboring elements, race condition becomes unavoidable. Two different approaches are analyzed for avoiding the race condition.

- Atomic Operations
- A specialized coloring algorithm

The primary use of atomic operation is to lock a particular memory location until the operation is complete [10]. This, in general, is undesirable because this serializes the execution of parallel threads resulting in reduced performance. The present implementation is designed to keep the number of threads waiting for a lock to be released to a minimum to justify the use of atomics. Especially for a large enough domain, the amount of serialization becomes particularly less and the results become comparable with the other method.

In the second approach, the grid is colored using eight different colors such that no elements of the same color share a node. An example of the coloring scheme is shown in figure 6. Eight different kernels each for one color are launched in serial. The grid size is kept fixed as the algorithm is further parallelized such that instead of one, eight threads collaborate to assemble each element. Although a coloring scheme reduces the parallelism in the algorithm, there is usually a large amount of parallelism left specially in case of large mesh sizes. As shown in figure 6, threads a to h collaborate to compute the elements of color 1 to 8 in eight different kernels. Also, since each thread-block computes approximately one eighth of the number of elements as in the previous approach, the shared memory and register requirements also become far less, resulting in a higher theoretical occupancy and lower execution time. This is why, although the eight kernels are launched in serial to perform the same task, this scheme of coloring helps ensure the same level of GPU utilization.

V. Solution of simultaneous equations

The global stiffness matrix obtained by assembly, as described in the previous section, is singular in nature. To complete the set of simultaneous equations, the essential and natural boundary conditions are applied by modifying suitable entries in the global stiffness matrix and load vector as mentioned in (4) respectively. Due to the large number of variables

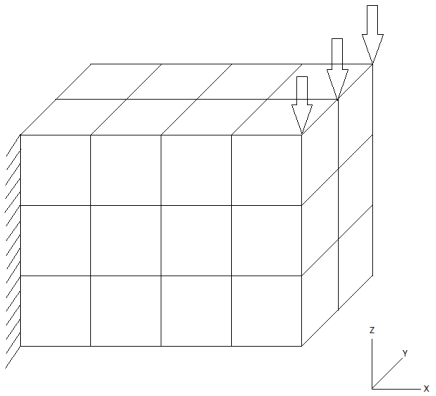

Fig. 7. Standard cantillever problem with distributed end load

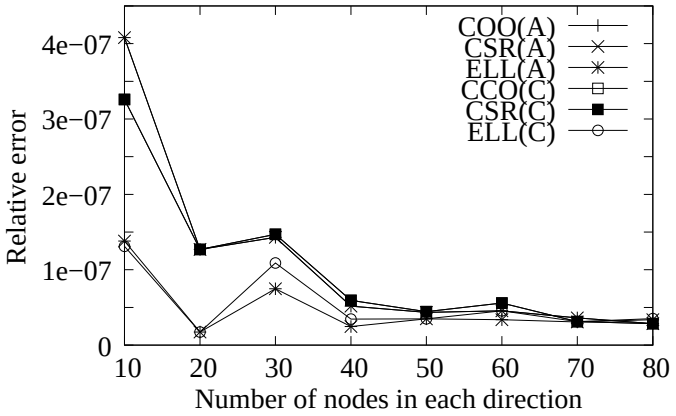

Fig. 8. Average Relative Error vs number of nodes in each direction of a cubic grid for different implementations. (C) and (A) refer to the method with coloring and atomics respectively.

specially in case of large grids, iterative methods, such as Krylov (sub)space solvers are typically more suited than direct solvers such as Cholskey decomposition. To solve the system of equations, any standard library providing optimized sparse matrix vector operations may be used. Cusp, a library for sparse linear algebra and graph computations is used for the present implementation [22].

VI. Results and Discussion

To validate the proposed assembly strategies for the GPU, the systems of equations for the deformation analysis of a simple cantillever beam subjected to certain loading conditions (as shown in figure 7) are compared with the results obtained on the CPU. The average relative error in single and double precision are compared and presented in figure 8. The amount of error is found to decrease with increasing grid size.

A. Performance Analysis

For the experimental setup, the CPU version of the assembly code is tested on a Intel Xeon ES1650 (6 core, 3.2 GHz) with 16 gigabytes of RAM. For the GPU version, a Tesla K40c

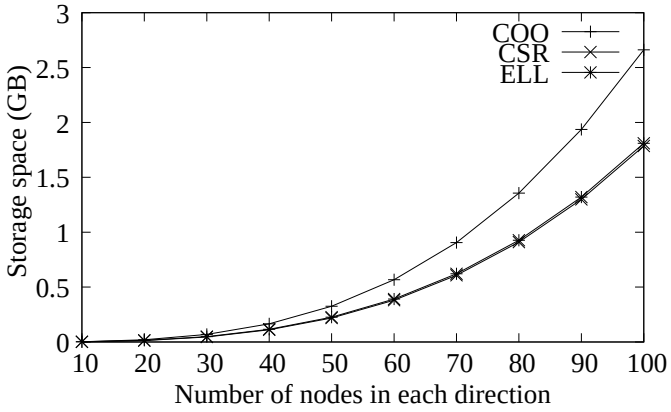

Fig. 9. Storage requirements vs number of nodes in each direction of a cubic grid for different formats

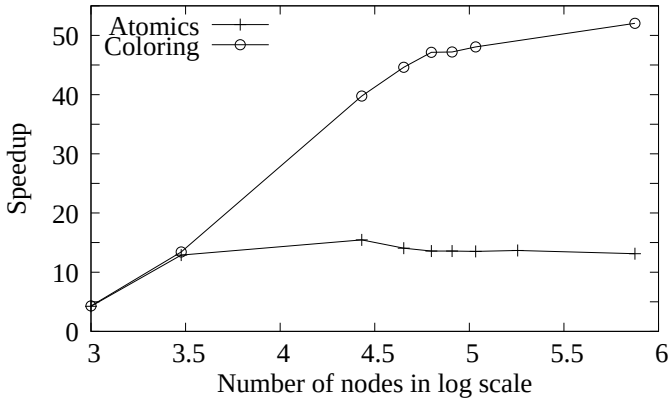

Fig. 10. Achieved speedup for COO format using atomics and graph coloring

with 12 gigabytes of global memory is used. The K40c has 15 streaming multiprocessors and 192 cores per multiprocessor, with a memory bandwidth of 288 GB per second.

The code used for reference follows an algorithm similar to the proposed addto method of assembly in COO format written for CPU. Because of the serial nature of the algorithm, no coloring or atomics is required. The timings of the GPU code are recorded with cudaEventRecord and include time taken by the kernels as well as CPU-GPU data transfers. To measure the execution time for the CPU, clock() function is used.

In figure 9 the storage requirements of the global stiffness matrix for different storage formats are presented. While the storage requirements for ELL and CSR are almost similar, COO format requires almost 50% more storage space compared to the other formats for the same grid size. For the present hardware, the largest grid possible to assemble has close to 4 million nodes. For a larger mesh size, domain decomposition techniques [23] may be employed to perform assembly. A graph partitioning library such as METIS [24] may be used to partition the finite element mesh for the same.

In figures 10, 11 and 12, the achieved speedup is compared with the number of elements using a semi logarithmic scale. With all the three storage formats, coloring approach is found to produce better results than the approach with atomics.

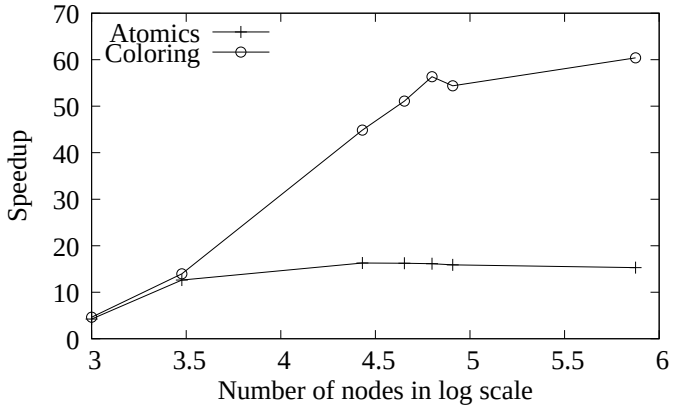

Fig. 11. Achieved speedup for CSR format using atomics and graph coloring

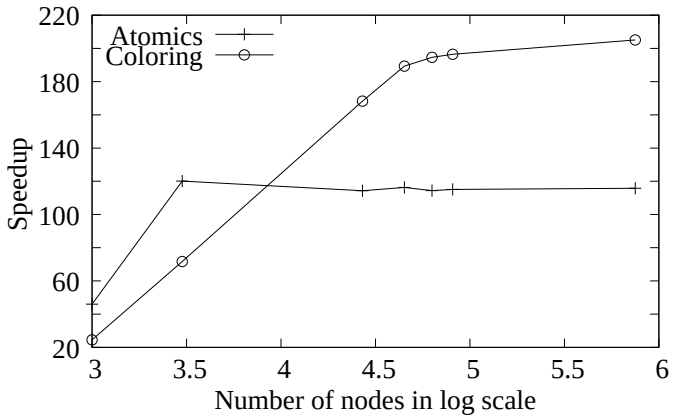

Fig. 12. Achieved speedup for ELL format using atomics and graph coloring

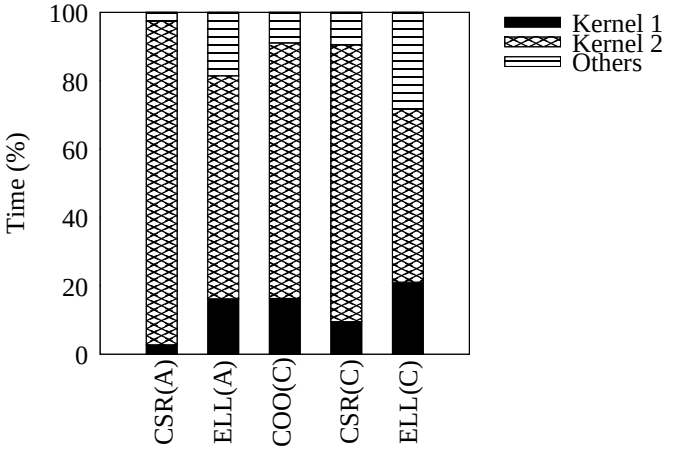

Fig. 13. Execution time of kernels for different implementations. (C) and (A) refer to the method with coloring and atomics respectively.

In figure 13, the execution time of each implementation are compared. Figure 13 also shows time required by different parts of each implementation. ELL format with coloring is found to be the most efficient of all approaches. It also has the lowest percentage of time taken by Kernel 2 among all methods. In all the approaches, Kernel 2 is found to be the most time consuming part of the proposed assembly procedure, whereas, Kernel 1 takes almost same time for all cases.

VII. Conclusion

This paper presents a number of strategies to perform assembly of global stiffness matrix for FEA in three dimensions performed on CUDA enabled graphics cards. We present a strategy to efficiently implement addto method on GPU with different sparse formats, while alleviating its inherent difficulties, as mentioned in [5]. The method is found to produce considerable speedup compared to a well-optimized code for assembly on CPU as demonstrated in the previous section. It should be mentioned here that such high values of speedup, specially for the ELL format is only possible because the assembly process does not include the numerical integration step, which is only permissible for structured meshes. Including the numerical integration will inevitably reduce the speedup by a significant amount. It can be concluded that, the proposed coloring approach performs better than the approach with atomics for the test problem. For a Finite element mesh with a high number of elements sharing one node, the method with atomics may perform better due to the reduced parallelism in coloring approach. Among all the sparse formats, assembly into ELL is found to take least time due to the reduced shared memory and register requirements, whereas assembly into CSR format requires least amount of storage space. It is also seen that unlike the traditional addto method as concluded in [9], assembly into CSR provides similar results to that of COO in case of the modified approach. Although implementation on unstructured grid is not discussed in the present work, similar strategies can be adopted for the same.

References

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

[2] S. Georgescu, P. Chow, and H. Okuda, "GPU acceleration for FEM-based structural analysis," *Archives of Computational Methods in Engineering*, vol. 20, no. 2, pp. 111–121, 2013.

[3] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *International Journal for Numerical Methods in Engineering*, vol. 85, no. 5, pp. 640–669, 2011.

[4] P. Macio, P. Paszewski, and K. Bana, "3d finite element numerical integration on GPUs," *Procedia Computer Science*, vol. 1, no. 1, pp. 1093 – 1100, 2010. {ICCS} 2010.

[5] G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin, "Finite element assembly strategies on multi-core and many-core architectures," *International Journal for Numerical Methods in Fluids*, vol. 71, no. 1, pp. 80–97, 2013.

[6] D. Komatitsch, D. Micha, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, no. 5, pp. 451 – 460, 2009.

[7] V. Challis, A. Roberts, and J. Grotowski, "High resolution topology optimization using graphics processing units (GPUs)," *Structural and Multidisciplinary Optimization*, vol. 49, no. 2, pp. 315–325, 2014.

[8] S. Schmidt and V. Schulz, "A 2589 line topology optimization code written for the graphics card," *Computing and Visualization in Science*, vol. 14, no. 6, pp. 249–256, 2011.

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

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

[11] Y. Zhao, "Lattice boltzmann based PDE solver on the GPU," *Vis. Comput.*, vol. 24, pp. 323–333, Mar. 2008.

[12] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: A domain specific language for building portable mesh-based PDE solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 9:1–9:12, ACM, 2011.

[13] V. Simek, R. Dvorak, F. Zboril, and J. Kunovsky, "Towards accelerated computation of atmospheric equations using CUDA," in *Computer Modelling and Simulation, 2009. UKSIM '09. 11th International Conference on*, pp. 449–454, March 2009.

[14] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on CUDA," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 3, pp. 198–201, Dec 2008.

[15] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 126–131, Aug 2009.

[16] W. Liu, B. Schmidt, G. Voss, and W. Mller-Wittig, "Accelerating molecular dynamics simulations using graphics processing units with "CUDA"," *Computer Physics Communications*, vol. 179, no. 9, pp. 634 – 641, 2008.

[17] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342 – 5359, 2008.

[18] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–7, April 2008.

[19] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka, "An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pp. 1–11, Nov 2010.

[20] F. M. Jr., T. Szakly, R. Mszros, and I. Lagzi, "Air pollution modelling using a graphics processing unit with CUDA," *Computer Physics Communications*, vol. 181, no. 1, pp. 105 – 112, 2010.

[21] D. Komatitsch, D. Gddeke, G. Erlebacher, and D. Micha, "Modeling the propagation of elastic waves using spectral elements onacluster of192 GPUs," *Computer Science - Research and Development*, vol. 25, no. 1-2, pp. 75–82, 2010.

[22] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2012. Version 0.3.0.

[23] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: Domain decomposition methods in hybrid cpugpu architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 1316, pp. 1490 – 1508, 2011.

[24] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," tech. rep., 1995.