GPU – based Topology Optimization using Matrix-Free Conjugate Gradient Finite Element Solver with Customized Nodal Connectivity Storage

Shashi Kant Ratnakar[1], Subhajit Sanfui[2] and Deepak Sharma[3][0000-0002-8939-9833]

Department of Mechanical Engineering, Indian Institute of Technology, Guwahati, Assam – 781039, India
{r.shashi[1], s.sanfui[2], dsharma[3]}@iitg.ac.in

Abstract. Topology optimization has been used to generate light-weight structures. However, the main issue with its implementation is a large computation time because it involves finite element (FE) simulations coupled with optimization. From the last few years, the graphics processing unit (GPU) has been used for reducing computation time by performing the computation in parallel and, thus, becomes an active research area. In this paper, a fine-grained node-by-node GPU computing strategy is proposed for the matrix-free conjugate gradient FE solver. The strategy is implemented with a customized nodal connectivity strategy. The performance of the proposed implementation is analyzed using three different mesh sizes on an elasticity problem. Results demonstrate 3 × of GPU speedup over a standard CPU implementation.

Keywords: Topology Optimization, Matrix-Free FEM Solver, GPU, Connectivity.

1 Introduction

Structural topology optimization is a method of finding the optimal layout of a structure within a specified region, which is subjected to a set of applied loads and boundary conditions [1]. The topology optimization formulation for the elasticity problem is given in Eq. (1).

$$\begin{aligned} &\min C = \sum_{e=1}^{N} \rho_e^p \{u_e\}^T [k_e] \{u_e\} \\ &Sub.to: [K]u = \textstyle\sum_{e=1}^{N} \rho_e^p [k_e]\{u_e\}, \sum_{e=1}^{N} \rho_e \, v_e \leq V^*, 0 < \rho_{e_{min}} \leq \rho_e \leq 1 \end{aligned} \quad (1)$$

Here, N is the total no. of elements in the mesh, ρ_e is the elemental density, $[k_e]$ is the elemental stiffness matrix of e −th element, $\{\boldsymbol{u_e}\}$ is the set of nodal displacements for e −th element, V^* is the user-defined volume limit on a structure, $[K]$ is the global stiffness matrix, $\{\boldsymbol{f}\}$ is the global force vector, v_e is the elemental volume, and p is the penalization parameter.

It has been observed from the literature that the topology optimization is a compute-intensive problem primarily due to the FEA of structures in each iteration of optimization [2, 3, 4]. Performing FE computations in parallel is the first choice in the majority of GPU-based implementations in the literature. Since a $kernel$ for parallel computation can be designed in different ways, the strategies such as element-by-element (EbE) [5, 6, 11], node-by-node (NbN) [7, 8, 9] and degree-of-freedom (DbD) [10] have been proposed in the literature. In the EbE strategy, a compute $thread$ of GPU is assigned to an element of FE mesh to perform the FEA computation [5]. With this $kernel$, a race-condition is observed when more than one $threads$ try to write values at the same location and at the same time. This problem can be handled using the coloring method or by using the atomic operation of CUDA with GPU. In the NbN strategy, a compute $thread$ of GPU is assigned to a node of FE mesh [8]. Since a node can be shared among different elements, this $thread$ does all required computation accordingly. In the DbD strategy, a compute $thread$ is assigned to a degree-of-freedom (DOF) of a system for performing the required computations [10].

Wadbro et al. [7] used the EbE strategy to solve a heat conduction topology optimization problem and showed $20\times$ speedup over single-core CPU and speedup of $3\times$ over OpenMP-based parallel implementation. The NbN strategy with a matrix-free conjugate gradient-based solver on GPU was presented by Schmidt et al. [8] that achieved a significant speedup over a shared memory CPU system. The DbD strategy was used by Martínez-Frutos et al. [9] to parallelized matrix-free PCG solver and filtering, sensitivity computation, and density updates were performed using the EbE strategy. A maximum speedup of $22.5\times$ with respect to the CPU implementation was reported for a heat conduction problem. The elemental stiffness matrix generation and its assembly on GPU have also been explored [12, 13, 14].

Most of the studies discussed above used the density-based topology optimization method. However, in the literature, other topology optimization methods have also been used with the GPU. Ram et al. [2] used GPU to accelerate topology optimization in which FE simulations were performed on GPU and multi-objective evolutionary algorithm on CPU.

From the literature, it can be seen that various strategies for performing parallel computations for FE analysis for topology optimization have been proposed. However, these strategies need further modifications to exploit GPU maximally. The following are the contributions of the paper.

- Development of fine-grained parallelism using the NbN strategy for performing matrix-free FE solver using conjugate gradient method on GPU for topology optimization of 3D linear elastic continuum structure.
- Development of a customized connectivity storage system for efficient storage and access for matrix-vector multiplication with the NbN strategy.
- Comparative analysis of the proposed strategy on three different meshes with a standard CPU implementation.

The paper is organized as follows. The preliminaries of the structural topology optimization are explained in Section 2. The implementation details of the present work are discussed in Section 3. In Section 4, the results and discussions are presented. The conclusions are presented in Section 5 with a scope of future work.

2 Preliminaries for Structural Topology Optimization

In structural topology optimization, the final topology is obtained by optimizing the problem given in Eq. (1). Solid isotropic material with penalization (SIMP) [2] is the most widely used density-based method, which is used in this paper. The computational steps involved in the SIMP-based topology optimization method are given in Algo.1.

Algorithm 1. The SIMP-based structural topology optimization method

Input: $[K_e],\ [C],\ [CO],\ p,\ iter,\ \varepsilon,\ r_{min},\ V_f$.
Output: $\boldsymbol{\rho}$.

1. $float\ \boldsymbol{u} = 0.0,\ \boldsymbol{\rho} = V_f,\ \boldsymbol{grad} = 0.0, int\ i = 1;$
2. $\boldsymbol{u} = \boldsymbol{finite\_element\_analysis}\ (\boldsymbol{u},\ \boldsymbol{\rho},\ p,\ C,\ \varepsilon);$
3. $\boldsymbol{grad} = \boldsymbol{compute\_sensitivity}\ (\boldsymbol{grad},\ \boldsymbol{u},\ \boldsymbol{\rho},\ p,\ C);$
4. $while\ (i < iter)\ do$
5. $\quad \boldsymbol{grad} = \boldsymbol{mesh\_independency\_filter}(\boldsymbol{grad},\ \boldsymbol{u},\ \boldsymbol{\rho},\ r_{min},\ C,\ CO);$
6. $\quad \boldsymbol{\rho} = \boldsymbol{design\_variable\_update}(\boldsymbol{\rho},\ V_f,\ \boldsymbol{grad});$
7. $\quad \boldsymbol{u} = \boldsymbol{finite\_element\_analysis}(\boldsymbol{u},\ \boldsymbol{\rho},\ p,\ C,\ \varepsilon);$
8. $\quad \boldsymbol{grad} = \boldsymbol{compute\_sensitivity}(\boldsymbol{grad},\ \boldsymbol{u},\ \boldsymbol{\rho},\ p,\ C);$
9. $\quad i++;$
10. end

The inputs are $[K_e]$**,** connectivity matrix $[C]$**,** nodal coordinate matrix $[CO]$**,** penalty factor (p), maximum no. of iterations $(iter)$, termination criterion for FEA solver (ε), filter radius for mesh-independency filter (r_{min}), and the limit on final structural volume (V_f). The output is given in the form of the updated elemental densities $(\boldsymbol{\rho})$. At the start of the algorithm, the vector $\boldsymbol{\rho}$ is initialized with a value. The vector $\boldsymbol{u}$ declared in line no. 1 is used to store the nodal displacements values, and the gradient of compliance with respect to the densities are stored in the vector $\boldsymbol{grad}$. The rest of the steps in the algorithm are as follows.

$\boldsymbol{finite\_element\_analysis}\ (\boldsymbol{u},\ \boldsymbol{\rho},\ p,\ C,\ \varepsilon)$**.** The first major step in topology optimization is to compute the nodal displacements using FEA. The design domain is discretized into finite elements, and the local stiffness matrices are computed. The displacements are computed by solving the elasticity equation $[K(\boldsymbol{\rho})]\{\boldsymbol{u}\} = \{\boldsymbol{f}\}$, where $[K]$ is the global stiffness matrix, and $\{\boldsymbol{f}\}$ is the global load vector. There are many FE solvers available in the literature to solve the elasticity equation. The conjugate gradient (CG) method is used to solve the system of linear equations. The output of this step is the nodal displacement vector $\boldsymbol{u}$.

$\boldsymbol{compute\_sensitivity}\ (\boldsymbol{grad},\ \boldsymbol{u},\ \boldsymbol{\rho},\ p,\ C)$**.** The compliance of the structure and its gradient with respect to the densities (sensitivity) are computed in lines 3 and 9 of Algo. 1 by using Eq. (2) and (3), respectively.

$c(\rho) = \sum_{e=1}^{N} (\rho_e)^p \{u_e\}^T [k_e] \{u_e\},$	(2)	$\frac{\partial c}{\partial \rho_e} = -p\, \rho_e{}^{p-1} \{u_e\}^T [k_e] \{u_e\},$	(3)

The values of sensitivities for all elements are then stored in the vector $\boldsymbol{grad}$ as the output of this function.

$\boldsymbol{mesh\_independency\_filter(grad, u, \rho, r_{min}, C, CO)}$**.** The filtering is a step towards ensuring the existence of the solution to topology optimization. The mesh-independency filter in step 5 of Algo.1 work by modifying the elemental sensitivities over a patch of elements. A user-defined filtering radius decides the patch of elements (r_{min}) [1]. The filtered sensitivities are stored in the vector $\boldsymbol{grad}$.

$\boldsymbol{design\_variable\_update(\rho, V_f, grad)}$**.** At each iteration, the design variable (elemental densities) are updated using Eq. (4).

$$\rho_e = \begin{cases} \max(\rho_0, \rho_e - m) & if\ \rho_e B_e^{\eta} \le \max(\rho_0, \rho_e - m) \\ \rho_e B_e^{\eta} & if\ \max(\rho_0, \rho_e - m) < \rho_e B_e^{\eta} < \min(1, \rho_e + m) \\ \min(1, \rho_e + m) & if\ \min(1, \rho_e + m) \le \rho_e B_e^{\eta}, \end{cases} \quad (4)$$

where m is a positive move-limit, $\eta = 0.5$ is a numerical damping coefficient and B_e is computed using Eq. (5).

$$B_e = -\frac{\partial c}{\partial \rho_e} \Big/ \lambda \frac{\partial V}{\partial \rho_e}, \quad (5)$$

where λ is the Lagrangian multiplier, which is obtained by a bi-sectioning algorithm [8]. The updated densities are saved in the vector $\boldsymbol{\rho}$**.** These updated densities are the used in the next FEA iteration.

Some of the steps discussed above can be computationally expensive. In order to analyse them, the CPU implementation is profiled and the percentage of total computation time taken by each step is presented in Table 1. It can be seen that 96.5% of the total computation time is taken by FEA step. Therefore, FEA needs to be performed on GPU in order to reduce the total computation time.

Table 1. The percentage of total computation time taken by each computational step of Algo. 1.

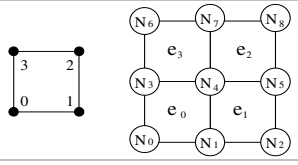
Computational step	Percentage	Computational step	Percentage
$\boldsymbol{finite\_element\_analysis}()$	96.5%	$\boldsymbol{compute\_sensitivity}()$	1.413%
$\boldsymbol{mesh\_independency\_filter}()$	1.003%	$\boldsymbol{design\_variable\_update}()$	1.002%

3 NbN Strategy for FEA on GPU

3.1 Customized Nodal Connectivity Storage

The customized nodal connectivity is explained with an example case of $4-$noded quadrilateral FE with two DOF per node for simplicity. In Fig. 1 (a), elements are numbered as e_0, e_1, etc. and the nodes are numbered as $\{N_0, \dots, N_8\}$. A local numbering for each element can also be seen in Fig. 1 (a). It can be observed that each node is part of multiple elements (up to 4 in the given mesh). Fig. 1(b) shows the connectivity matrix $[C]$ for the patch of elements shown in Fig. 1(a). The first row of [C] represents the connectivity of the nodes with the first element. Similarly, other rows represent connectivity with other elements. For the NbN computation, the list of elements that are shared by a node N_i is needed.

The connectivity detail is customized so that the data can be read efficiently on GPU. This customized connectivity is referred to as the reverse-connectivity matrix $[C_{rev}]$. The matrix $[C_{rev}]$ is created by performing a search operation through $[C]$ for each FE



(a) Nodal connectivity

$$[\mathbf{C}] = \begin{bmatrix} N_0 & N_1 & N_4 & N_3 \\ N_1 & N_2 & N_5 & N_4 \\ N_4 & N_5 & N_8 & N_7 \\ N_3 & N_4 & N_7 & N_6 \end{bmatrix}$$

(b) Connectivity matrix

Fig. 1. A 2D Patch of finite elements and nodes.

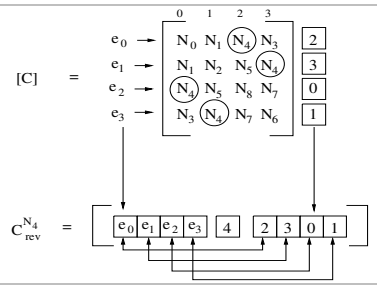


Fig. 2. Construction of [C_{rev}] from [C] for one node N_4.

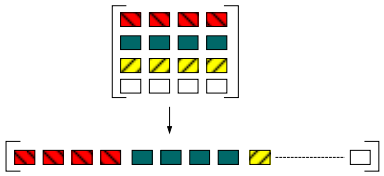


Fig. 3. Linearization of a 2D matrix into a 1D array following the row-major order.

node. Let us consider a single element N_4 from Fig. 1(a). The process of constructing $[C_{rev}]$ from $[C]$ is shown in Fig. 2. The search operation through $[C]$ will show that the node N_4 is part of four elements $\{e_0, e_1, e_2, e_3\}$. The local positioning of N_4 in these elements is $\{2, 3, 0, 1\}$ respectively. Both sets of data are copied in $C_{rev}^{N_4}$ as shown in Fig. 2. The columns $1-4$ of $C_{rev}^{N_4}$ stores the elements and its local positioning in these elements is stored in columns $6-9$, respectively. Column 5 contains the total no. of elements shared with N_4. The same process is carried out for each node in the mesh to create $[C_{rev}]$. Since FE mesh does not change during topology optimization, the matrix $[C_{rev}]$ needs to be created only once at the beginning of the computation. The elemental stiffness matrix is computed on CPU and stored in a single matrix $[K_e]$ at the beginning of Algo 1. Both $[C_{rev}]$ and $[K_e]$ are copied into the GPU global memory in 1D arrays following the row-major order as shown in Fig. 3. This is done to simplify the memory read operations on GPU.

3.2 Matrix-Free FE Solver using CG on GPU

The present implementation uses a matrix-free conjugate gradient method as the finite element solver. The algorithm of the CG solver is briefly explained in the Algo. 2. Referring to line no. 5 of Algo. 2, it can be noticed that inside the loop there is a matrix-vector multiplication (Mat-vec) between K and $\boldsymbol{p_k}$. Apart from the Mat-vec, there are a number of linear algebraic operations involved.

In present work, the entire FEA is carried out on GPU. The Mat-vec operation is performed on GPU by using our custom-developed CUDA kernel, which is based on the NbN strategy. The rest of the linear algebraic operations are parallelized using CUDA toolkit's Thrust library.

Algorithm 2. Conjugate gradient (CG) algorithm

$\boldsymbol{Input}: K, \boldsymbol{f}, \boldsymbol{u_0}, \varepsilon, k_{max}$
$\boldsymbol{Output}: \boldsymbol{u_k}$
1. $r_0 \leftarrow \boldsymbol{f} - K\boldsymbol{u_0}$
2. $\boldsymbol{p_0} \leftarrow \boldsymbol{r_0}$
3. $k \leftarrow 0$
4. **while** $k < k_{max}$ **do**
5. $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T K p_k}$
6. $\boldsymbol{u_{k+1}} \leftarrow \boldsymbol{u_k} + \alpha_k \boldsymbol{p_k}$
7. $\boldsymbol{r_{k+1}} \leftarrow \boldsymbol{r_k} - \alpha_k K \boldsymbol{p_k}$
8. if $|\boldsymbol{r_{k+1}}| \leq \varepsilon$ then
9. exit
10. end if
11. $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
12. $\boldsymbol{p_{k+1}} \leftarrow \boldsymbol{r_{k+1}} + \beta_k \boldsymbol{p_k}$
13. $k \leftarrow k + 1$

Algorithm 3. NbN Mat-vec Kernel

$\boldsymbol{Input}: K_e, C, C_{rev}, \boldsymbol{u}, \boldsymbol{\rho}, p, Node.$
$\boldsymbol{Output}: res$
1. $int\ n = threadIdx.x + blockIdx.x * blockDim.x;$
2. if $(n < Node)$
3. $float\ val = 0.0;$
4. $int\ ele\_shared \leftarrow read\ from\ C_{rev};$
5. $for \leftarrow i = 1\ to\ ele\_shared\ do$
6. $int\ id_1 \leftarrow read\ from\ C_{rev};$
7. $float\ \rho_e = \boldsymbol{\rho}(e)^p;$
8. $for \leftarrow j = 1\ to\ 8\ do$
9. $int\ id_2 \leftarrow read\ from\ C;$
10. $val \mathrel{+}= \rho_e * (K_e[i][j] * \boldsymbol{u}[id_2]);$
11. end
12. end
13. $\boldsymbol{res}[id_1] = val;$
14. end

Mat-vec Kernel using NbN Strategy: In this kernel, each compute $thread$ is assigned to one node. This $thread$ reads the required data from GPU memory, performs the computations, and finally writes back result into the output array. The steps of the NbN Mat-vec kernel are shown in Algo.3. All the input data are copied into the global memory of GPU. In line 1 of Algo. 3, the global $thread$ index refers to the global number of a node in FE mesh. In line 3, the $thread$ initializes the variable val, which stores Mat-vec result for one element. Next, the $thread$ loads a value ele_shared, which is extracted from $[C_{rev}]$**.** Essentially, ele_shared stores all those elements which share the current node n. For each element in ele_shared, the $thread$ loads an index id_1 in line 6. The index id_1 indicates the position in the output array $\boldsymbol{res}$, where the $thread$ will be writing the result of Mat-vec. In line 7, the $thread$ reads elemental density and penalizes it with p. This penalized density will be later multiplied with the result of Mat-vec. The $thread$ then reads another index id_2 from the matrix $[C]$ **.** The index id_2 signifies the components of elemental displacement vector $\boldsymbol{u}$ with which the elements of $[K_e]$ will be multiplied. Finally, the Mat-vec is computed in line 10, and after multiplying it with the penalized density, the result is temporarily stored in variable val. When both for loop ends, the final value of val is copied into the output vector $\boldsymbol{res}$, as shown in line 13.

4 Results and Discussion

The proposed NbN strategy with customized nodal connectivity implementation is tested on a 3D cantilever beam example as shown in Fig. 3. The ratio L:B:H of the beam is taken as 2:1:1. The meshing is done using $8-$noded hexahedral elements using $ANSYS\ R16.1$ FE package. Three different mesh sizes are used for performance analysis. Table 2 shows the details of these meshes.

A fraction of 30% of the total volume is taken that acts a constraint over the final volume of the structure. The other parameters are taken as; Young's modulus $E = 1$,

Poisson's ratio $\mu = 0.3$, penalty exponent $p = 3.0$, filter radius $= 4.0$, and minimum density $= 0.001$. The maximum no. of iterations is 50, and the termination criterion of CG is taken as 10^{-3}. An NVIDIA Tesla K40 card is used for performing computations in parallel, which has 2880 cores, and 12 GB memory. The serial code is run on Intel Xeon(R) CPU E5 1620 (3:70 GHz × 8, and 16 GB RAM).

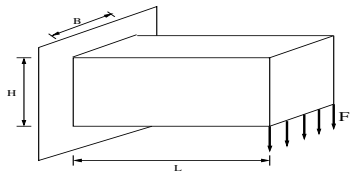


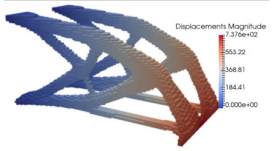
Fig. 3. 3D cantilever beam

Table 2. The number of nodes and elements for the mesh sizes.

Mesh	No. of nodes	No. of elements
Mesh 1	128,000	120,159
Mesh 2	182,250	172,304
Mesh 3	250,000	237,699

The final topology is shown in Fig. 4(a) in which only the elemental densities higher than 0.8 are plotted. The final topology is found to be similar obtained by Schmidt et al. [8]. The displacement field in Fig. 4(b) shows that the deflection is higher at the end where the load is applied. The comparison of computation time of one iteration between CPU and GPU implementations is shown in Fig. 5. It can be noticed that for all mesh sizes, GPU implementation outperforms the CPU implementation. The speedup with respect to the no. of nodes in the mesh sizes is shown in Fig. 6. It can be observed that GPU speedup is varying between 2.5 × - 3 × over CPU.

(a) Topology



(b) Displacement field

Fig. 4. The optimal topology obtained and its displacement field

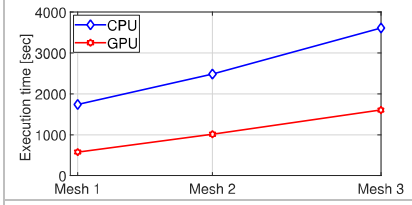


Fig. 5. Computation time per optimization iteration for three mesh sizes used.

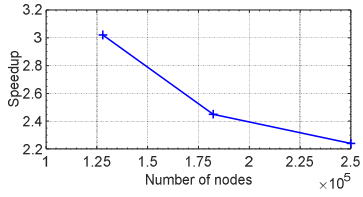


Fig. 6. CPU-GPU speedup for three mesh sizes.

5. Conclusion

The NbN strategy with the customized nodal connectivity was proposed for GPU implementation and tested on a 3D cantilever problem. Three meshes were used and a speedup up to 3 × was observed against the CPU. The optimum topology was found to be similar reported in the literature. It can be concluded that the customized nodal connectivity storage minimizes the data transfer between the *thread* and GPU memory,

resulting in a significant speedup over the CPU. In the future, the customized nodal connectivity storage can be modified and tested with unstructured meshes. A GPU-kernel using the DOF-by-DOF strategy can be designed to exploit fine-grained parallelism further. Lastly, the proposed NbN strategy can be explored for the topology optimization of problems other than the linear elasticity.

References

1. Bendsoe, M. P., Sigmund, O.: Topology optimization: theory, methods, and applications. Springer Science & Business Media (2013).
2. Ram, L., Sharma, D.: Evolutionary and GPU computing for topology optimization of structures. Swarm and Evolutionary Computation, 35, 1-13 (2017).
3. Sharma, D., Deb, K., Kishore, N., N.: Customized Evolutionary Optimization Procedure for Generating Minimum Weight Compliant Mechanisms. Engineering Optimization, 46 (1), 39-60 (2014).
4. Sharma, D., Deb, K., Kishore, N., N.: Domain-Specific Initial Population Strategy for Compliant Mechanisms Using Customized Genetic Algorithm, Structural and Multidisciplinary Optimization, 43 (4), 541-554 (2011).
5. Zegard, T., Paulino, G. H.: Toward GPU accelerated topology optimization on unstructured meshes. Structural and Multidisciplinary Optimization, 48(3), 473-485 (2013).
6. Duarte, L. S., Celes, W., Pereira, A., Menezes, I. F., Paulino, G. H.: PolyTop++: an efficient alternative for serial and parallel topology optimization on CPUs & GPUs. Structural and Multidisciplinary Optimization, 52(5), 845-859 (2015).
7. Wadbro, E., Berggren, M.: Megapixel topology optimization on a graphics processing unit. SIAM review, 51(4), 707-721 (2009).
8. Schmidt, S., Schulz, V.: A 2589-line topology optimization code written for the graphics card. Computing and Visualization in Science, 14(6), 249-256 (2011).
9. Martínez-Frutos, J., Martínez-Castejón, P. J., Herrero-Pérez, D.: Efficient topology optimization using GPU computing with multilevel granularity. Advances in Engineering Software, 106, 47-62 (2017).
10. Martínez-Frutos, J., Herrero-Pérez, D.: Large-scale robust topology optimization using multi-GPU systems. Computer Methods in Applied Mechanics and Engineering, 311, 393-414 (2016).
11. Kiran, U., Sharma, D., Gautam, S. S.: GPU-warp based finite element matrices generation and assembly using coloring method. Journal of Computational Design and Engineering, 6(4), 705-718 (2019).
12. Sanfui, S., Sharma, D.: Exploiting Symmetry in Elemental Computation and Assembly Stage of GPU-Accelerated FEA. In: Liu, G. R., Cui, F., Xiangguo, G. X. (eds.) 10th International Conference on Computational Methods (ICCM2019), 9–13 July 2019, pp. 641 – 651, ScienTech Publisher, Singapore.
13. Sanfui, S., Sharma, D.: GPU Acceleration of Local Matrix Generation in FEA by Utilizing Sparsity Pattern, In 1st International Conference on Mechanical Engineering (INCON 2018), 4–6 January 2018, Jadavpur University, India.
14. Sanfui, S., Sharma, D: A Two-Kernel based Strategy for Performing Assembly in FEA on the Graphic Processing Unit, In IEEE International Conference on Advances in Mechanical, Industrial, Automation and Management Systems, 3-5 February 2017, MNNIT Allahabad, India.