GPU-based Element-by-Element Strategies for Accelerating Topology Optimization of 3D Continuum Structures Using Unstructured All-Hexahedral Mesh

Shashi Kant Ratnakar, Subhajit Sanfui and Deepak Sharma*
Department of Mechanical Engineering,
Indian Institute of Technology Guwahati,
Assam, India, 781039
Email: r.shashi@iitg.ac.in, s.sanfui@iitg.ac.in, dsharma@iitg.ac.in

Topology optimization has been successful in generating optimal topologies of various structures arising in real-world applications. Since these applications can have complex and large domains, topology optimization suffers from a high computational cost because of the use of unstructured meshes for discretization of these domains and their finite element analysis (FEA). This paper addresses this challenge by developing three GPU-based element-by-element strategies targeting unstructured all-hexahedral mesh for the matrix-free precondition conjugate gradient (PCG) finite element solver. These strategies mainly perform sparse matrix multiplication (SpMV) arising with the FEA solver by allocating more compute threads of GPU per element. Moreover, the strategies are developed to use shared memory of GPU for efficient memory transactions. The proposed strategies are tested with solid isotropic material with penalization (SIMP) method on four examples of 3D structural topology optimization. Results demonstrate that the proposed strategies achieve speedup up to 8.2× over the standard GPU-based SpMV strategies from the literature.

Keywords: Topology optimization, GPU, Matrix-free PCG solver, Unstructured all-hexahedral mesh.

1 Introduction

Structural topology optimization is a method of optimizing the material distribution within a design domain under a prescribed set of loading, and boundary conditions. Various methods are available for topology optimization in the literature, such as homogenization method [1], Solid Isotropic Material with Penalization (SIMP) method [2], level-set method [3], evolutionary computation [4], etc. Using these methods, topology optimization is used in many applications, such as compliant mechanism design [5, 6, 7], aerospace design [8, 9], biomedical design [10, 11], multi-physics and micro-design [12, 13], to name a few.

The structural topology optimization is an iterative method that involves various computational steps. These steps are:

- **Meshing:** the design domain is discretized into finite elements.
- **FEA:** finite element analysis is then performed to capture the structural response under the specified loading and boundary conditions.
- **Sensitivity Analysis:** sensitivity of the objective function with respect to design variable is computed.
- **Mesh-Independency Filter:** the sensitivities are then filtered over a neighbourhood of finite elements. This step eliminates the checker-board pattern in the final topology [14].
- **Update:** based upon the filtered sensitivity, the design variable is updated.

These steps are followed till the termination criterion is not met.

Among the various computational steps of topology optimization, FEA is found to be the most computationally expensive step [15, 16]. In the last few years, graphics processing unit (GPU) has been used for accelerating FEA that includes elemental stiffness matrix generation [17, 18], assembly [19, 20, 21], and the solver [22, 23]. The main reason of exploring GPU is that it offers a large number of computing cores at low cost and maintenance. The details of the architecture of GPU are given in the supplementary sheet [See Supplemental Figure 1]. The other steps of topology optimization, such as FEA solver [24, 25], sensitivity and FEA solver [26, 27], and FEA solver, sensitivity, filtering, and update [28, 29] are also performed on GPU. It is found from the literature that the FEA solver consumes the most of the computation time. Therefore, some attempts have been made for accelerating the direct and iterative solvers [30]. Iterative solvers are preferred because of simplicity and inherent parallelism. Since GPU has a limited amount of memory, the

*Corresponding author

matrix-free or assembly-free iterative solvers [31] are preferred because these solvers can perform the required computations at the elemental level without assembling and storing large global stiffness matrix.

The matrix-free FEA solver involves sparse matrix-vector multiplication (SpMV) and vector arithmetic operations. For a finite element mesh containing '$Elem$' number of elements, and 'n' being the number of elemental DoFs, the SpMV operation requires $Elem \times n(2n-1)$ number of floating-point operations, whereas a vector-vector product requires $Elem \times (2n-1)$ number of floating-point operations. Furthermore, SpMV also needs the elemental stiffness matrices and connectivity information of all elements. Hence, the amount of data-transfer in SpMV operation is much higher than that in a vector arithmetic operation, thereby making it a computational bottleneck for the matrix-free iterative solver [26]. In the literature, the following three types of GPU-based SpMV strategies can be found:

1. **element-by-element** (***ebe***) [32]: One compute thread of GPU is assigned to each element of FE mesh [24]. However, this strategy suffers with race-condition when more than one threads try to access and modify the same memory location, ultimately producing inconsistent results. The coloring method or atomic operation can be used for alleviating this issue.
2. **node-by-node** (***nbn***) [26]: A single thread computes the state for all degree-of-freedoms (DoF) of a node. This strategy requires access to data of neighboring elements. Since each node performs its computation independently, there is no race-condition observed [23].
3. **DoF-by-DoF** (***dbd***) [29]: This SpMV strategy aims to achieve even more finer level of parallelism by assigning a compute thread to a DoF of FE nodes. Similar to ***nbn*** strategy, this also require the access of neighboring element's data. Since ***dbd*** strategy works at the level of DoF there is no race condition among the compute threads.

The ***nbn***− and ***dbd***− strategies have shown good speedups for the domains discretized using structured meshes [26, 31]. For structural topology optimization structured meshes are more popular in the literature. However, there are many applications that use unstructured all-hexahedral meshes because of their ability to deal with arbitrary complex geometries [28]. Since with an unstructured mesh each node can be associated with different number of neighboring elements, it can result in unbalanced computational load among GPU threads for the ***nbn***− and ***dbd***− strategies. One of the remedies is to use the ***ebe***−strategy by carefully dealing with race-condition. To the best of authors knowledge, there are only two studies in the literature that used unstructured meshes while accelerating SIMP-based structural topology optimization on GPU. The first work is by Zegard and Paulino [28] that investigated feasibility of unstructured meshes for 2D structure. In this work, GPU was used to speedup the entire topology optimization procedure. The ***ebe***−strategy was used to assemble the global stiffness matrix. GPU-based Cholesky decomposition method along with CUBLAS library [33] was used to solve the system of linear equations. The other work is by Duarte et al. [24] that used polygonal meshes for both 2D and 3D continuum structures. In this work, the ***ebe***−based preconditioned conjugate gradient (PCG) solver was used on GPU. Both the studies used greedy graph coloring method for handling race-condition.

From these studies, it was observed that GPU acceleration of 3D topology optimization poses several challenges. One of the primary challenges is the efficient storage and access of elemental stiffness matrices and connectivity information on GPU especially for unstructured meshes, due to the scattered nature of the data [32]. The second key challenge is the development of efficient thread allocation strategies for matrix-free SpMV in order to properly utilize the massively parallel architecture of GPUs. This paper thus aims to address these challenges by developing efficient kernels for unstructured all-hexahedral mesh by allocating more compute threads to the ***ebe***−strategy for reducing the computational time of topology optimization. Following are the key contributions of this paper.

- Three ***ebe***−SpMV strategies are developed for accelerating 3D structural topology optimization using unstructured all-hexahedral meshes on GPU.
- A matrix-free PCG solver is developed using the proposed SpMV strategies and it is compared with the standard ***ebe***−strategy from the literature.

A comparative analysis of the proposed SpMV strategies is presented using four examples of 3D structural topology optimization. The rest of the paper is organized as follows. Section 2 presents the relevant literature on density-based topology optimization methods using GPU. Section 3 presents the basics of SIMP-based topology optimization and its implementation on GPU. The proposed ***ebe***−strategies are presented in Section 4. The numerical experiments are performed in Section 5 and the results are presented. Section 6 presents the conclusions with a note on future work.

2 Literature Survey

The density-based topology optimization method is one of the popular methods in the literature [34]. In this section, the relevant studies accelerating the same method on GPU are discussed.

The earliest implementation of density-based topology optimization on GPU was presented by Wadbro and Berggren [35]. The design domain was discretized using structured mesh, and a GPU-based matrix-free PCG solver was used. The SpMV computations in PCG solver were performed using the ***ebe***−strategy, and CUBLAS [33] was used for linear algebraic operations of vectors. A maximum speedup of 20× was shown over the CPU implementation for the 2D heat conduction problem.

Schmidt and Schulz [26] accelerated topology optimization of 3D linear elastic structures by employing the ***nbn***−SpMV strategy for the matrix-free conjugate gradient

(CG) solver. The data of three successive slices of nodes was stored into shared memory. A custom data-type was developed for storing nodal displacements and elemental density. An extruded 3D cantilever beam with 2 million nodes was considered. The double precision GPU implementation showed speedup of 1.7× over the shared memory system with 48 CPU cores for one CG iteration.

Zegard and Paulino [28] presented a GPU-based framework considering 2D unstructured meshes. The ***ebe***−strategy was used to assemble global stiffness matrix on GPU. A greedy-graph coloring algorithm was implemented to avoid race-condition. The Cholesky decomposition-based solver was used by assigning one thread per row of global stiffness matrix. CUBLAS library of CUDA toolkit was used for vector operations. The global stiffness matrix was stored using a packed lower triangular banded scheme. The GPU-based solver showed speedups between $15\times - 20\times$ over the CPU for three linear elasticity examples.

Duarte et al. [24] developed a polygonal mesh-based framework called 'PolyTop++' for 2D and 3D structures on GPU. The framework used two direct solvers and a matrix-free PCG iterative solver. SpMV was performed by following the ***ebe***−strategy with a greedy-graph coloring algorithm. Only half of the elemental stiffness matrix was stored, which finally transferred to 1D array. For a problem size of 10 million polygonal elements, the GPU-based PCG solver showed 13× speedup over the CPU counterpart and outperformed the direct solver by 1.5× for a mesh with 1 million elements.

Frutos and Perez [27] presented topology optimization of large-scale linear elastic structures on multi-GPU systems. Multiple GPUs were used to perform FEA, sensitivity analysis, and mesh filtering. This study used structured mesh for domain discretization and thus, used a single stiffness matrix for the entire mesh. A GPU-based matrix-free PCG solver was designed using the ***dbd***−strategy, and sensitivity analysis and mesh filtering were performed using the ***ebe***−strategy. The GPU solver showed maximum speedups of 8× and 14× for the double hook design problem [27] and the heat sink design problem [27] , respectively, for a mesh having 1.7 million DoFs.

Frutoz et al. [29] proposed a multilevel granularity-based implementation on GPU. The matrix-free PCG solver was implemented using the ***dbd***−strategy. Sensitivity analysis, mesh filtering, and density update were computed using the ***ebe***−strategy. The GPU-based PCG solver showed speedups of 19.9×, 19.4×, and 22.5× on the tied-arch bridge design [29], the 3D gripper design [29], and the heat sink design problem [29], respectively.

Ratnakar et al. [36] dealt with unstructured mesh for 3D structure on GPU. A matrix-free PCG solver was used with the ***ebe***−strategy. Thrust library [37] of CUDA toolkit was used for linear algebraic operations of vectors. A 3D L-beam problem was considered and a speedup of 4× was observed over the CPU implementation for a mesh with 103,680 elements. Furthermore, Ratnakar et al. [38] used the ***nbn***−strategy and a customized nodal connectivity strategy was proposed to reduce memory transactions between the thread and GPU global memory. In this work also, the matrix-free PCG solver and CUDA Thrust library were used. A 3D cantilever beam problem having 250,000 nodes was considered that showed a speedup of 3× over the CPU implementation.

From these studies, it is observed that the majority of the articles considered the structured mesh for topology optimization. However, many applications with complex domains, loading, and boundary conditions need unstructured meshing for discretization of a design domain. Some handful studies attempted to use unstructured mesh in topology optimization. However, the implementation poses various challenges such as efficient computing, storage and access of elemental matrices, thread allocation strategies for matrix-free SpMV, and optimal use of GPU memories to fully leverage GPU resources. This paper presents novel SpMV-strategies to accelerate topology optimization by developing kernels for unstructured all-hexahedral mesh.

3 SIMP-based Topology Optimization and its GPU Implementation

3.1 Problem Formulation

The structural topology optimization problem is formulated as compliance minimization problem with volume constraint [2] that is shown in equation (1).

$$\begin{aligned} \min_{\boldsymbol{\rho}} \quad & C(\boldsymbol{\rho},\mathbf{u}), \\ \text{subject to}: \quad & \mathrm{K}(\boldsymbol{\rho})\mathbf{u} = \boldsymbol{f}, \\ & V(\boldsymbol{\rho}) \le V^*, \\ & 0 \le \rho_e \le 1, \quad e \in \Omega, \end{aligned} \tag{1}$$

where C denotes the compliance of a structure, $\boldsymbol{\rho}$ is the vector of density variable, $\boldsymbol{u}$ is the nodal displacement vector, K is the global stiffness matrix, and $\boldsymbol{f}$ is the global load vector. The final volume of the structure, $V(\boldsymbol{\rho})$, should not exceed the user-defined volume fraction (V^*). To avoid singularity in K, a minimum value for ρ_e is taken as a small non-zero number (ρ_{min}) during implementation, that is, $0 < \rho_{min} \le \rho_e \le 1$.

The SIMP method [2] thus penalizes the intermediate material densities using a penalty parameter (p). The relationship between the elemental densities and material properties are given by the power-law as given in equation (2).

$$\begin{aligned} \{E_{ijkl}\}_e &= {\rho_e}^p \, E^0_{ijkl}, \qquad p > 1, \\ E_{ijkl} &= \begin{cases} 0 & \text{if} \quad \rho_e = 0, \\ E^0_{ijkl} & \text{if} \quad \rho_e = 1, \end{cases} \end{aligned} \tag{2}$$

where E^0_{ijkl} is the material property of a solid material.

The displacement vector ($\boldsymbol{u}$) is computed by solving $\mathrm{K}(\boldsymbol{\rho})\mathbf{u} = \boldsymbol{f}$ using FEA. Once the nodal displacements are

calculated, the sensitivity of objective function with respect to ρ_e is found as given in equation (3).

$$\begin{aligned} C(\boldsymbol{\rho}) &= \sum_{e=1}^{N_{elem}} {\rho_e}^p \boldsymbol{u}_e{}^T \mathrm{K}_e \boldsymbol{u}_e, \\ \frac{\partial C(\boldsymbol{\rho})}{\partial \rho_e} &= -p{\rho_e}^{p-1} \boldsymbol{u}_e{}^T \mathrm{K}_e \boldsymbol{u}_e, \end{aligned} \tag{3}$$

where $\boldsymbol{u}_e$ is the elemental displacement vector of element 'e', K_e is the elemental stiffness matrix of element 'e', N_{elem} is the total number of elements, and ρ_e is the elemental density. Topology optimization must ensure that the solution is mesh-independent and there is no checker-board pattern. A filter over the derivatives of the objective function is applied as given in equation (4). This step is called mesh-independency filter or density filter.

$$\frac{\widehat{\partial C(\boldsymbol{\rho})}}{\partial \rho_e} = \frac{\sum_{i \in \xi} \frac{\partial C(\boldsymbol{\rho})}{\partial \rho_i} \cdot \rho_i \cdot H_{ei}}{\rho_e \sum_{i \in \xi} H_{ei}}, \tag{4}$$

where ξ is the set of neighboring elements of 'e' that is found by a user-defined filter radius 'R'. H_{ei} is calculated using equation (5), where d_{ei} is the Euclidean distance between the centroid of element 'e' and element 'i'.

$$H_{ei} = \begin{cases} R - d_{ei}, & \text{if } ||d_{ei}|| \leq R, \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

The design variables are updated in every iteration by following the optimality criterion update scheme shown in equation (6). Here, m is the positive move limit, and η is the numerical damping coefficient that is considered as 0.5. B_e is computed using equation (7), where λ is the Lagrange multiplier that can be computed using the bi-section algorithm.

$$\widehat{\rho_e} = \begin{cases} max(\rho_{min}, \rho_e - m), & \text{if } \rho_e B_e^{\eta} \leq max(\rho_{min}, \rho_e - m), \\ min(1, \rho_e + m), & \text{if } min(1, \rho_e + m) \leq \rho_e B_e^{\eta}, \\ \rho_e B_e^{\eta}, & \text{otherwise,} \end{cases} \tag{6}$$

$$B_e = \frac{-\frac{\partial C(\boldsymbol{\rho})}{\partial \rho_e}}{\lambda \frac{\partial V(\boldsymbol{\rho})}{\partial \rho_e}}. \tag{7}$$

3.2 Computational Implementation

Figure 1 shows the flowchart of the various computational steps involved in topology optimization. Since FEA is the most time-consuming step [16], it is performed on the GPU. Other steps are performed on the CPU because they require serial computation and are less computationally expensive. SIMP-based topology optimization requires a variety of inputs, including the design domain, boundary and loading conditions, maximum number of iterations, volume fraction, etc. The design domain is then discretized using finite elements. In this paper, ANSYS *R*16.1 FE package is used to mesh the domain from which the nodal connectivity data is generated. Thereafter, pre-processing is done that includes preparing the connectivity matrix (C), computing the elemental stiffness matrix (K_e), and pre-processing data for mesh filtering. The design variables ($\boldsymbol{\rho}$) and nodal displacements ($\boldsymbol{u}$) are then initialized. The host (CPU) allocates space for K_e, C, and $\boldsymbol{\rho}$ and copies data in the device (GPU) global memory. Thereafter, kernels are launched by the host to perform FEA in parallel in which the size of thread *blocks* and *grid* are specified. The execution of FEA computations on GPU is discussed in the following subsections. After FEA computes the nodal displacements, $\boldsymbol{u}$ and $\boldsymbol{\rho}$ are copied back to the host. The next step is to compute and filter the sensitivities using the expressions given in equations (3). The elemental densities are updated based on the filtered sensitivities, and a convergence check is performed. This completes one iteration of topology optimization. These steps are repeated till the convergence criterion is not met.

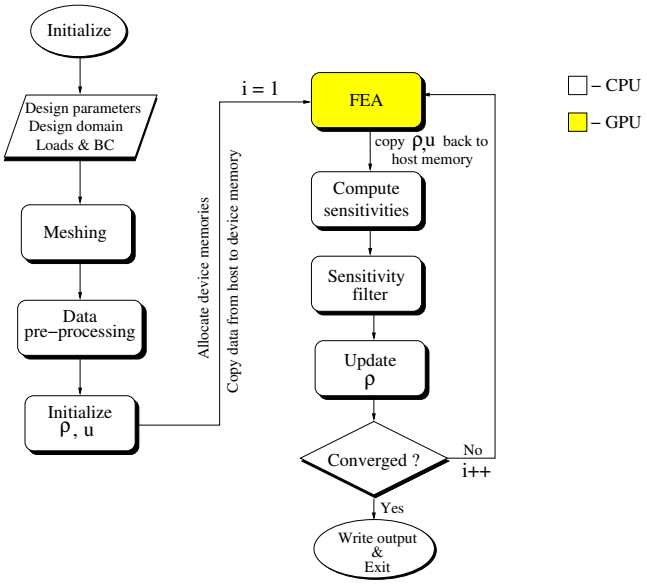


Fig. 1. Computational steps of SIMP-based topology optimization

3.3 Matrix-free PCG Solver

The steps of the matrix-free PCG FE solver are shown in Algorithm 1. The inputs required by the solver include initial guess ($\boldsymbol{u}_0$), preconditioner ($\boldsymbol{M}^{-1}$), connectivity matrix (C), elemental densities ($\boldsymbol{\rho}$), penalization parameter (p), global stiffness matrix (K), and load vector $\boldsymbol{f}$. Here, K and $\boldsymbol{f}$ are not explicitly generated but rather constructed on-the-fly using the elemental stiffness matrices. These elemental stiff-

ness matrices are stored in the array K. In Algorithm 1, the penalized form of K is represented by $\overline{\mathrm{K}}$. The Jacobi preconditioner ($\boldsymbol{M}^{-1}$) is computed on the CPU and is copied to the GPU.

The PCG solver starts by computing the residual ($\boldsymbol{r}_i$) in line 2. It can be seen that this step requires SpMV operation between the penalized stiffness matrix '$\overline{\mathrm{K}}$' and initial guess vector '$\boldsymbol{u}_0$'. SpMV operation is performed locally at the elemental level. The vector product of the residual with the preconditioner ($\boldsymbol{M}^{-1}$) is performed in line 3. In line 4, the output is copied to a new vector $\boldsymbol{d}_i$. Line 5 performs inner product of $\boldsymbol{r}_i$ and $\boldsymbol{z}_i$ to compute δ^{new}. In line 7, there is a loop over the maximum number of CG iterations and the minimum value of residual. One more SpMV operation is performed between $\overline{\mathrm{K}}$ and vector $\boldsymbol{d}_i$ in line 8. Algorithm computes α_i in line 9 using the output of SpMV. The parameter α_i is then used to update $\boldsymbol{u}_i$ and $\boldsymbol{r}_i$ in lines 10 and 11, respectively. The updated residual vector $\boldsymbol{r}_i$ is then multiplied with $\boldsymbol{M}^{-1}$, and the resulting vector is stored in $\boldsymbol{z}_i$. The inner product of $\boldsymbol{r}_i$ and $\boldsymbol{z}_i$ is computed in line 14, and the result is stored in variable δ^{new}. In line 15, the value of β_i is calculated using δ^{new} and δ^{old}. The parameter β_i is used to update the values in vector $\boldsymbol{d}_i$, as shown in line 16. This completes one iteration of the PCG solver. This iterative process continues until the termination conditions given in line 7 are met.

The PCG algorithm requires two SpMV operations and several vector-vector product operations. SpMV operation is known to be the most computationally expensive step of a matrix-free FEA solver [26]. Two SpMV kernels using the ***nbn***− and ***ebe***−strategies are discussed in the following subsection. The vector-vector operations are performed on GPU by using the thrust library [37] of CUDA toolkit.

Algorithm 1: Matrix-free PCG FE solver for SIMP-based topology optimization

Data: K, $\boldsymbol{f}$, C, $\boldsymbol{M}^{-1}$, $\boldsymbol{u}_0$, $\boldsymbol{\rho}$, p, i_{max}, ε
Output: $\boldsymbol{u}$

1 $i \leftarrow 0$
2 $\boldsymbol{r}_i \leftarrow \boldsymbol{f} - \overline{\mathrm{K}}\boldsymbol{u}_i$ // SpMV kernel
3 $\boldsymbol{z}_i \leftarrow \boldsymbol{M}^{-1}\,\boldsymbol{r}_i$ // CUDA Thrust
4 $\boldsymbol{d}_i \leftarrow \boldsymbol{z}_i$ // CUDA Thrust
5 $\delta^{new} \leftarrow \boldsymbol{r}_i^T \boldsymbol{z}_i$ // CUDA Thrust
6 $\delta_i \leftarrow \delta^{new}$ // CUDA Thrust
7 **while** $i < i_{max}$ ***and*** $\delta^{new} > \varepsilon$ **do**
8 $\quad \boldsymbol{q}_i \leftarrow \overline{\mathrm{K}}\boldsymbol{d}_i$ // SpMV kernel
9 $\quad \alpha_i \leftarrow \delta^{new} / \boldsymbol{d}_i^T \boldsymbol{q}_i$ // CUDA Thrust
10 $\quad \boldsymbol{u}_i \leftarrow \boldsymbol{u}_i + \alpha_i \boldsymbol{d}_i$ // CUDA Thrust
11 $\quad \boldsymbol{r}_i \leftarrow \boldsymbol{r}_i - \alpha_i \boldsymbol{q}_i$ // CUDA Thrust
12 $\quad \boldsymbol{z}_i \leftarrow \boldsymbol{M}^{-1}\,\boldsymbol{r}_i$ // CUDA Thrust
13 $\quad \delta^{old} \leftarrow \delta^{new}$
14 $\quad \delta^{new} \leftarrow \boldsymbol{r}^T \boldsymbol{z}_i$ // CUDA Thrust
15 $\quad \beta_i \leftarrow \delta^{new} / \delta^{old}$
16 $\quad \boldsymbol{d}_i \leftarrow \boldsymbol{z}_i + \beta_i \boldsymbol{d}_i$ // CUDA Thrust
17 $\quad i \leftarrow i + 1$
18 **end**

3.4 Matrix-free SpMV Kernels on GPU

In this section, two SpMV strategies are discussed through algorithmic representation that are used in the literature.

3.4.1 Node-by-Node (*nbn*) Kernel

The ***nbn***−kernel is developed for unstructured all-hexahedral mesh created using 8-noded hexahedral elements. The data required by the kernel are the elemental stiffness matrices (K_e), connectivity matrix (C), nodal displacements ($\boldsymbol{u}$), and elemental density vector ($\boldsymbol{\rho}$). A custom data storage is adopted [38] that rearranges the connectivity matrix so that the data needed by a thread can be accessed with fewer number of memory transactions. The new rearranged connectivity matrix is called as reverse-connectivity matrix (C_{rev}), and is created by performing one-time exhaustive search at the beginning. The kernel output is stored in an array '$\boldsymbol{r}$'.

The ***nbn***−kernel is shown in Algorithm 2 that starts by assigning a global index to a compute thread that represents a node in the mesh. Line 2 shows a loop over the total number of nodes in the mesh (*Node*). A temporary variable *val* is declared in line 3 for storing the SpMV result for thread 't'. Line 4 reads the global indices of the neighbourhood elements of thread 't' from C_{rev} and stores them in $\boldsymbol{\xi}_t$. The thread then loops over these neighborhood elements in line 5. For each neighborhood element 'e', thread 't' reads index 'id_1' from C_{rev} in line 6. The index id_1 represents local position of node 't' inside an element 'e'. The elemental density of element 'e' is read from $\boldsymbol{\rho}$ and is penalised using SIMP parameter 'p' in line 7. In line 8, the thread loops over the nodes of element 'e'. Since the kernel is developed for 8−noded hexahedral element, the loop runs over 8 nodes. In line 10, an index id_2 is read from C for each node of an element 'e'. The index id_2 represents the global index of node 'j'. Finally, the matrix-vector multiplication (mat-vec) between K_e and $\boldsymbol{u}_e$ is performed in line 11 and the result is stored in the variable *val*. The expression in line 11 is an implicit representation for the computation of all three DoFs of a node. The cumulative result is written to output vector $\boldsymbol{r}$ in line 12.

It can be observed that each thread needs multiple accesses to C_{rev} and C for reading the connectivity indices. In addition, each thread must read K_e of each element in $\boldsymbol{\xi}_t$. Since the number of elements in $\boldsymbol{\xi}_t$ can be different for each node, it creates a load imbalance among the compute threads. This is one of the major limitations of the ***nbn***−strategy in which the overall performance of the kernel may deteriorate.

3.4.2 Element-by-Element (*ebe*) Kernel

Figure 2 shows the elemental stiffness matrix (K_e) and nodal displacement vector ($\boldsymbol{u}_e$) of an element. Since 8-noded hexahedral element with 3 DoF per node is considered, the size of K_e and $\boldsymbol{u}_e$ is 24×24 and 24×1, respectively. All entries of K_e and $\boldsymbol{u}_e$ are multiplied by a single compute thread

Algorithm 2: ***nbn***−SpMV Kernel

Data: K, C, C_{rev}, $\boldsymbol{u}$, $\boldsymbol{\rho}$, p, $Node$
Output: $\boldsymbol{r}$

1 $t = blockIdx.x \times blockDim.x + threadIdx.x$;
2 **if** $t < Node$ **then**
3 $val = 0.0$;
4 $\boldsymbol{\xi}_t \leftarrow$ read from C_{rev}; // neighborhood elements
5 **foreach** $e \in \boldsymbol{\xi}_t$ **do**
6 $id_1 \leftarrow$ read from C_{rev};
7 $\rho_e = \rho[e]^p$;
8 **for** $j \leftarrow 0$ *to* 7 // nodes of 'e'
9 **do**
10 $id_2 \leftarrow$ read from C;
11 $val \mathrel{+}= \rho_e \times \{K_e[id_1][j] \times u_e[id_2]\}$;
  // matrix-vector product
12 $r[t] = val$;
13 **end**

in the standard ***ebe***−strategy [36] that is represented by the outermost red box. This thread is responsible for reading the required data, performing computations, and writing the results.

Algorithm 3 presents the kernel for the standard ***ebe***−strategy. The data requirement of the kernel is similar to the ***nbn***−kernel, except for C_{rev}. The thread index 't' is allocated to an element 'e' of the mesh in line 1. In line 3, a vector $\boldsymbol{C}_\ell$ is allocated in local memory of GPU for reducing transactions between a thread and global memory. The global indices of eight nodes of an element 'e' are stored in the vector $\boldsymbol{C}_\ell$ in line 4. The elemental density is read and penalized in line 5. A loop over the nodes of an element 'e' can be seen in line 6. The total entries multiplied in one iteration of this loop are represented by the green boxes in Figure 2. In line 8, the global index 'id_1' of node 'i' is read from local memory. Index id_1 represents the write position in the output vector $\boldsymbol{r}$. The variable val, which stores the mat-vec value for node 'i', is initialized in line 9. The second loop over the nodes of an element 'e' can be seen in line 10. In line 11, the global index 'id_2' of node 'j' is read from $\boldsymbol{C}_\ell$. The same index also represents the location in $\boldsymbol{u}_e$ which is multiplied with entry of K_e. Finally, in line 12, the thread performs the mat-vec operation between K_e and $\boldsymbol{u}_e$. The total entries multiplied in one iteration of loop in line 10 are shown in the innermost blue boxes in Figure 2. In line 13, the product of mat-vec is stored in an array $\boldsymbol{r}$. During writing of the result to the output array, a race condition can be observed. This issue is resolved in line 13 by using CUDA's atomic operation.

It can be observed from Algorithm 3 that there are two nested loops in this kernel with the most common data transaction being the retrieval of the global indices of the nodes 'i' and 'j'. Each thread requires $(64+8) = 72$ memory transactions to read these indices, and these data are not shared with other threads in the thread block. Hence, the local memory is used for temporary storage of connectivity data since it reduces transactions between the thread and the device's global memory. Although each thread has the same computational load, Algorithm 3 shows that a single thread has to perform several computations. However, the multiplication of entries from the red boxes in Figure 2 is independent of each other. This observation motivates us to improve the kernel by distributing the computational load across more number of threads.

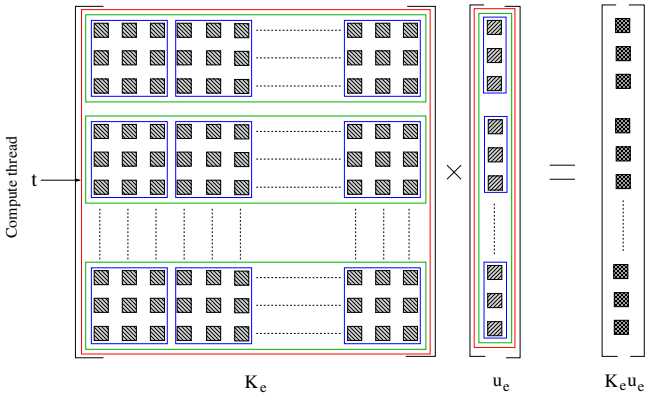


Fig. 2. Matrix-vector multiplication by the ***ebe***−strategy

Algorithm 3: ***ebe***−SpMV Kernel

Data: K, C, $\boldsymbol{u}$, $\boldsymbol{\rho}$, p, $Elem$
Output: $\boldsymbol{r}$

1 $t = blockIdx.x \times blockDim.x + threadIdx.x$;
2 **if** $t < Elem$ **then**
3 $\boldsymbol{C}_\ell$ [8]; // local memory space
4 $\boldsymbol{C}_\ell \leftarrow$ copy from C;
5 $\rho_e = \rho[t]^p$;
6 **for** $i \leftarrow 0$ *to* 7 // nodes of 't'
7 **do**
8 $id_1 \leftarrow \boldsymbol{C}_\ell[i]$;
9 $val = 0.0$;
10 **for** $j \leftarrow 0$ *to* 7 **do**
11 $id_2 \leftarrow \boldsymbol{C}_\ell[j]$;
12 $val \mathrel{+}= \rho_e \times \{K_e[i][j] \times u_e[id_2]\}$; // matrix-vector product
13 $r[id_1] \mathrel{+}= val$; // Atomic add
14 **end**

4 Proposed EbE Kernels

In this section, we discuss three fine-grained kernels using the ***ebe***−strategy for performing SpMV on GPU. The idea is to share a single thread's computational load over

more number of threads. The detailed discussion are presented in the following subsections.

4.1 8−Thread per Element (*ebe8*) Kernel

In the proposed kernel, eight threads are assigned to an element for the ***ebe***−strategy. Since 8−noded hexahedral element is used for meshing, this kernel thus divides the workload of the standard ***ebe***−strategy among eight compute threads of GPU. The mat-vec operation between K_e and $\boldsymbol{u}_e$ of one finite element 'e' is shown in Figure 3. It can be seen that the red boxes divide 24 rows of K_e into eight groups and each group contains three rows. In this kernel, one thread is assigned to each of the eight groups and thus, reduces computational load on a single GPU thread. It is referred to as the ***ebe8***−kernel.

The computational steps of the ***ebe8***−kernel are given in Algorithm 4. The data requirement is similar to the standard ***ebe***−kernel. The thread is assigned to a global index in line 1. In line 2, a vector '$\boldsymbol{C}_s$' is allocated in shared memory to store the elemental connectivity data. Here, the size of $\boldsymbol{C}_s$ is determined by the size of thread block. A thread is then assigned to an element 'e' in line 4. Every thread allocated to an element 'e' copies one entry from C to a vector $\boldsymbol{C}_s$ in line 5. A synchronization barrier is applied in line 6. Line 7 reads the elemental density and penalizes it. In line 8, the row index of thread 't' is computed, and in line 9 the index 'id_1' is read from $\boldsymbol{C}_s$. Index id_1 represents the global position in the output array $\boldsymbol{r}$. In line 10, the thread loops over the columns of K_e. In line 12, thread reads another index 'id_2' within the loop, which is the global location of $\boldsymbol{u}_e$ for multiplying with the corresponding K_e entries. The mat-vec operation is performed in line 14 and the result is stored in the variable 'val'. The cumulative result of the loop is then stored in an array $\boldsymbol{r}$. Atomic operation of CUDA is used in line 15 to avoid the race-condition. The innermost blue boxes in Figure 3 show the elements that are multiplied in one iteration of *for-loop* in line 10.

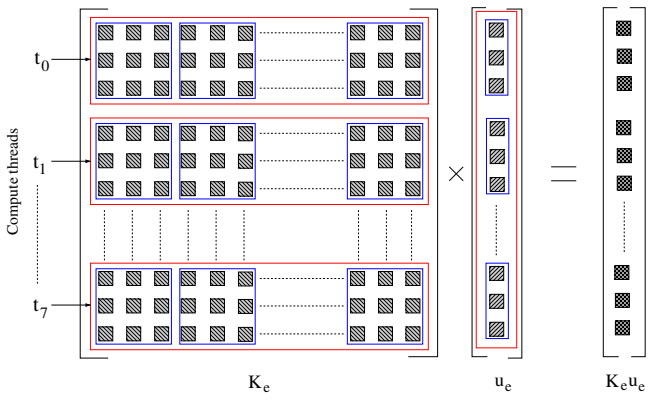


Fig. 3. Matrix-vector multiplication by the ***ebe8***−strategy

Algorithm 4: ***ebe*** 8−thread per element SpMV (***ebe8***) Kernel

Data: K, C, $\boldsymbol{u}$, $\boldsymbol{\rho}$, p, $Elem$
Output: $\boldsymbol{r}$

1 $t = blockIdx.x \times blockDim.x + threadIdx.x$;
2 __shared__ $\boldsymbol{C}_s[size]$; // shared memory space
3 **if** $t < (Elem * 8)$ **then**
4 $e = (int)\ (t/8)$; // global element index
5 $\boldsymbol{C}_s \leftarrow$ copy from C;
6 $__syncthreads()$;
7 $\rho_e = \rho[e]^p$;
8 $row = t - (e \times 8)$; // row number
9 $id_1 \leftarrow \boldsymbol{C}_s$;
10 **for** $col \leftarrow 0$ *to* 7 // columns of K_e
11 **do**
12 $id_2 \leftarrow \boldsymbol{C}_s$;
13 $val = 0.0$;
14 $val += \rho_e \times \{K_e[row][col] \times u_e[id_2]\}$; // matrix-vector product
15 $r[id_1] += val$; // Atomic add
16 $__syncthreads()$;
17 **end**

4.2 24−Thread per Element (*ebe24*) Kernel

The computational load is further divided in this kernel by assigning one row of K_e to a thread. The kernel thus allocates 24 threads to each 8−noded hexahedral finite element in the mesh. Figure 4 shows grouping of K_e entries among GPU threads. The K_e entries inside the red boxes belong to one thread that multiply with the corresponding entries of $\boldsymbol{u}_e$. This strategy is referred to as the ***ebe24***−kernel.

The computational steps of the ***ebe24***−strategy is shown in Algorithm 5. The steps up to line 5 are identical with Algorithm 4. The thread 't' is assigned to an element 'e' in line 7. The elemental density is penalized in line 8 and is stored in ρ_e. The row index of K_e, which is allocated to this thread, is shown in line 9. In line 10, the index 'id_1' is read from $\boldsymbol{C}_s$ in shared memory, and the temporary variable val is initialized in line 11. There is a loop over the columns of K_e in line 12. Index id_2 is read from $\boldsymbol{C}_s$ for each value of col. Finally, in line 15, the thread computes $K_e \times \boldsymbol{u}_e$, and the penalized density (ρ_e) is multiplied by the product. By using the CUDA's atomic operation, the output of mat-vec operation is written to an array $\boldsymbol{r}$ in line 16 without any race-condition. Mat-vec operation in line 15 represents multiplication for the entries inside the innermost blue boxes in Figure 4. Mat-vec operation for the entire row is done by looping over these boxes.

4.3 64−Thread per Element (*ebe64*) Kernel

This kernel is proposed for further reducing the computational load of the standard ***ebe8***−strategy by assigning 64 threads to an element. As shown in Figure 5, the entries of K_e are now grouped into 3×3 tiles shown in different colors. Each tile is then assigned to a compute thread that multiplies

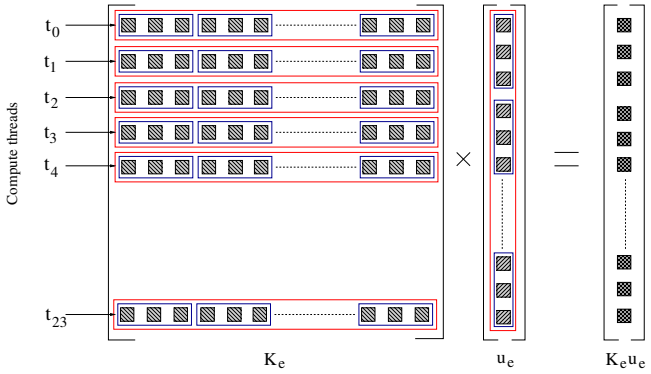


Fig. 4. Matrix-vector multiplication by the ***ebe*24**−strategy

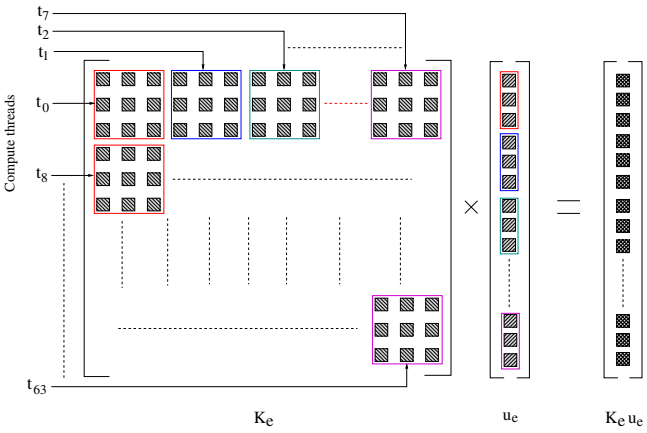


Fig. 5. Matrix-vector multiplication by the ***ebe*64**−strategy

Algorithm 5: ***ebe*** 24−thread per element (***ebe*24**) SpMV Kernel

Data: K, C, $\boldsymbol{u}$, $\boldsymbol{\rho}$, p, $Elem$
Output: $\boldsymbol{r}$

1 $t = blockIdx.x * blockDim.x + threadIdx.x$;
2 __shared__ $\boldsymbol{C}_s[size]$; // shared memory space
3 **if** $threadIdx.x < size$ **then**
4 $\quad C_s[threadIdx.x] \leftarrow$ copy form C;
5 $\_\_syncthreads()$;
6 **if** $t < (Elem * 24)$ **then**
7 $\quad e = (int)\ (t/24)$; // global element index
8 $\quad \rho_e = \rho[e]^p$;
9 $\quad row = t - (e * 24)$; // row index
10 $\quad id_1 \leftarrow \boldsymbol{C}_s$;
11 $\quad val = 0.0$;
12 $\quad$ **for** $col \leftarrow 0$ *to* 7 // columns of K_e
13 $\quad$ **do**
14 $\quad\quad id_2 \leftarrow \boldsymbol{C}_s$;
15 $\quad\quad val += \rho_e * \{K_e[row][col] * u_e[id_2]\}$; // matrix-vector product
16 $\quad r[id_1] += val$; // Atomic add
17 $\_\_syncthreads()$;
18 **end**

Algorithm 6: ***ebe*** 64−thread per element (***ebe*64**) SpMV Kernel

Data: K, C, $\boldsymbol{u}$, $\boldsymbol{\rho}$, p, $Elem$
Output: $\boldsymbol{r}$

1 $t = blockIdx.x * blockDim.x + threadIdx.x$;
2 __shared__ $\boldsymbol{C}_s[size]$; // shared memory space
3 **if** $threadIdx.x < size$ **then**
4 $\quad \boldsymbol{C}_s[threadIdx.x] \leftarrow$ copy from C
5 $\_\_syncthreads()$;
6 **if** $t < (Elem * 64)$ **then**
7 $\quad val = 0.0$;
8 $\quad e = (int)\ (t/64)$;
9 $\quad \rho_e = \rho[e]^p$;
10 $\quad row = (t - e * 64)/8$; // row index
11 $\quad col = t - (8 * (t/8))$; // column index
12 $\quad id_1 \leftarrow \boldsymbol{C}_s$;
13 $\quad id_2 \leftarrow \boldsymbol{C}_s$;
14 $\quad val += \rho_e * \{K_e[row][col] * u_e[id_2]\}$; // matrix-vector product
15 $\quad r[id_1] += val$; // Atomic add
16 $\_\_syncthreads()$;
17 **end**

each entries of this tile with the corresponding three entries of $\boldsymbol{u}_e$. Since 24×24 entries are grouped into 64 tiles, the same number of threads are used in this kernel. This strategy is referred to as the ***ebe*64**−kernel.

The computational steps of the ***ebe*64**−kernel are shown in Algorithm 6. The initial steps up to line 5 are similar to Algorithm 4. The temporary variable '*val*' is declared in line 7 to store the result. Line 8 shows the global index of an element '*e*' assigned to a thread. Line 9 computes the penalized elemental density of the element '*e*'. The row and column indices for thread '*t*' are computed in lines 10, and 11, respectively. Indices id_1 and id_2 are read form shared memory in lines 12 and 13, respectively. The mat-vec operation is performed in line 14. Finally, thread '*t*' writes result in the output array $\boldsymbol{r}$ using the atomic operation in line 15.

It can be seen from Algorithms 4, 5, and 6 that both the ***ebe*8**− and ***ebe*24**− kernels have one *for-loop*, while the ***ebe*64**−kernel has none. In addition, shared memory is used in all three fine-grained SpMV kernels to store the connectivity data required by a finite element for reducing the global memory transactions.

5 Numerical Experiments

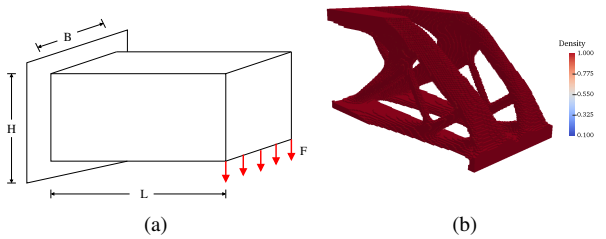
Four structural topology optimization examples are used to test the performance of the proposed SpMV strategies. For each example, five different mesh sizes are used to evaluate the scalability of the proposed strategies. The first two examples are the 3D cantilever beam and the 3D L-beam. These examples have regular domain geometries. The third and fourth examples are the Michell cantilever and the con-

necting rod of an automobile engine. These examples have complex domain geometries and boundary conditions. All four examples are meshed using 8−noded hexahedral elements. The computational performance for each example is compared with the standard ***ebe***−kernel [36] presented in Section 3.4.2.

The simulations are performed on a CPU with Intel Xeon E5 − 2650 v4 processor equipped with 12 cores and 24 threads. It has clock speed of 2.2 GHz and offers bandwidth of 76.8 GB/s. The GPU instances are run on NVIDIA Tesla K40*c* card that has 2880 cores and 12 GB of memory with bandwidth of 288 GB/s. For topology optimization, an artificial material with the following properties are considered: Young's modulus $(E) = 1$ and Poisson's ratio $(\nu) = 0.3$. The implementations presented in this paper are unit-less and the topology is unaffected by changes in material properties. The design domains are discretized using ANSYS R16.1 APDL module, and the discretized domains for all four examples are shown in the supplementary sheet [See Supplemental Material]. The SIMP penalty parameter (p) is set to 3.0 for all examples, and the residual error value for PCG (ε) is set to 10^{-5}. All numerical experiments are run for 50 iterations of optimization.

5.1 3D Cantilever beam

The domain and boundary conditions of a 3D cantilever beam are shown in Figure 6(a). The $L : B : H$ ratio of the cantilever beam is $2 : 1 : 1$. Each node of the left face is subjected to the Dirichlet boundary condition, while each node at the lower edge of the right face is subjected to the Neumann boundary condition.



(a) (b)

Fig. 6. (a) Domain, loading and boundary conditions, and (b) obtained topology of 3D cantilever beam.

A limit on the final volume of the optimal structure (V_f) is imposed that is 30% of the original volume. The minimum value of elemental density (ρ_{min}) is kept as 0.1. The total number of elements for all five mesh sizes are listed in Table 1.

The obtained topology after 50 iterations is shown in Figure 6(b). The topology is obtained for the largest mesh size (CB_5) of 3D cantilever beam using the ***ebe*64**−kernel. For CB_5 mesh, the other four GPU SpMV kernels produce the same topology. In Figure 6(b), the lower element densities are filtered out and only $\rho_e > 0.9$ are shown. For this example, a similar topology was reported by Schmidt and Schulz [26]. The structural compliance value of the final topology for all mesh sizes of 3D cantilever beam using the ***ebe*64**−kernel is given in Table 2. For a given mesh size, all SpMV kernels generate the same value of compliance.

Table 1. Total number of elements in the respective mesh sizes of all four examples.

Mesh	1	2	3	4	5
CB	31,250	85,750	182,250	432,000	1,024,000
LB	32,768	85,184	188,384	438,976	1,000,000
MC	38,148	85,674	183,432	453,175	1,170,894
CR	33,178	85,376	182,637	465,708	1,033,820

Table 2. Structural compliance values of the final topology for all mesh sizes using the ***ebe*64**−kernel.

Mesh	1	2	3	4	5
CB	4945	6380	7864	10051	12718
LB	2107	2690	2975	4219	5278
MC	149	152	160	168	176
CR	436,350	439,732	441,549	448,162	452,735

5.2 3D L-beam

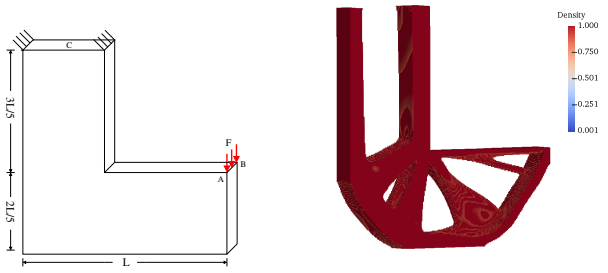
The second example is a 3D L-beam that is shown in Figure 7(a). The figure shows the design domain, loading, and boundary conditions of the example. The face 'C' is subjected to the Dirichlet boundary condition, while the edge 'AB' is subjected to the Neumann boundary condition. The beam thickness is taken as $0.25 \times (2L/5)$. During optimization, a unit load is applied at each node of the edge 'AB'. For optimization, $V_f = 45\%$ and $\rho_{min} = 0.001$ are used. The details of the mesh sizes used for 3D L-beam example are given in Table 1.

The obtained topology obtained after 50 iterations is shown in Figure 7(b). The topology corresponds to LB_5 mesh using the ***ebe*64**−kernel. The lower element densities are filtered out, and the figure shows for $\rho_e > 0.9$.

Table 2 shows the structural compliance values obtained after 50 iterations for all mesh sizes of 3D L-beam. The given values in the table correspond to the ***ebe*64**−kernel.

5.3 Michell Cantilever

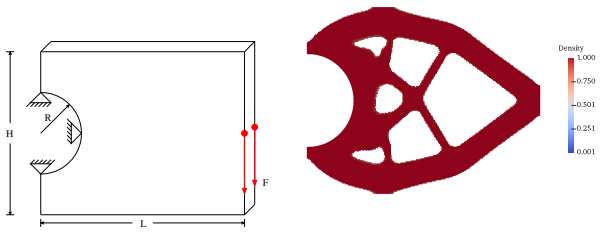
In topology optimization, the Michell cantilever is a well-known benchmark example [39]. Since Michell can-



(a) (b)

Fig. 7. (a) Domain, loading, and boundary conditions, and (b) obtained topology of 3D L-beam.

tilever is a 2D example, it is converted into a 3D structure by taking a few elements in the third direction. This example has a semi-circular boundary that is supported as shown in Figure 8(a). The supported semi-circular boundary is subjected to the Dirichlet boundary condition, whereas the loaded points in the figure are subjected to the Neumann boundary condition. The $L:H$ ratio is taken as $5:4$. For topology optimization, $V_f = 45\%$ and $\rho_{min} = 0.001$ are used. Table 1 presents the different mesh sizes considered for this example. The number of elements in the third direction for each mesh size are 3, 3, 4, 5, and 6, respectively.



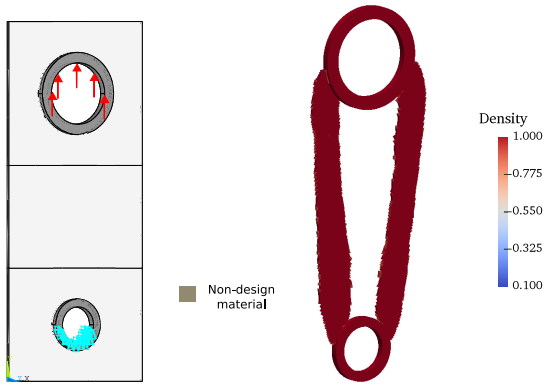
(a) (b)

Fig. 8. (a) Domain, loading, and boundary conditions, and (b) obtained topology of Michell cantilever.

The obtained topology for MC_5 mesh is shown in Figure 8(b). The densities are filtered and only $\rho_e > 0.9$ are shown in the figure. Table 2 presents the structural compliance values for all mesh sizes. These values are obtained using the ***ebe*64**−kernel after 50 iterations.

5.4 Connecting rod of an Automobile Engine

The last example of a connecting rod of an automobile engine is considered for topology optimization. This example is adopted from the work of Nana et al. [40]. The initial domain geometry, loading, and boundary conditions of the connecting rod are shown in Figure 9(a). The following dimensions are considered; height = 350 units, length = 150 units, inner and outer diameters of the upper bore = 60 units and 80 units, and inner and outer diameters of the lower bore = 35 units and 50 units. Both bores are considered non-design material, and during optimization, no material is removed from these bores. As shown in Figure 9(a), the lower half of the bottom bore is subjected to the Dirichlet boundary condition, while the upper half of the top bore is subjected to the Neumann boundary condition. The values $V_f = 30\%$ and $\rho_{min} = 0.001$ are considered for optimization. Table 1 presents the details of the different mesh sizes for the connecting rod.



(a) (b)

Fig. 9. (a) Domain, loading, and boundary conditions, and (b) obtained topology of connecting rod of an automobile engine.

The obtained topology after 50 iterations is shown in Figure 9(b) corresponding to $\rho_e > 0.9$. Since the bores are considered as the non-design material, no material is removed from them. The obtained topology shows that the non-design parts of the original design domain are connected by two symmetric links. The obtained topology matches with the one reported by Nana et al. [40].

The values of structural compliance for all mesh sizes of the connecting rod example are listed in Table 2. These values are obtained using the ***ebe*64**−kernel.

These results show that the proposed GPU kernels can solve a wide variety of structural topology optimization examples. Simple and complex domain geometries, loading, and boundary conditions in the four examples are used in this paper. For both structured and unstructured all-hexahedral meshes, the proposed kernels generated the known topologies from the literature. The performance of the proposed GPU implementations is analyzed in the following section.

5.5 Convergence

The convergence plots of all examples are shown in Figure 10. The plots show the change in the structural compliance value corresponding to the largest mesh size of each example with respect to the number of iterations of optimization. All four examples show smooth convergence of the objective function values with respect to number of iterations. In the first few iterations ($5-10$), the compliance

value reduces significantly, and thereafter, a minor change is observed.

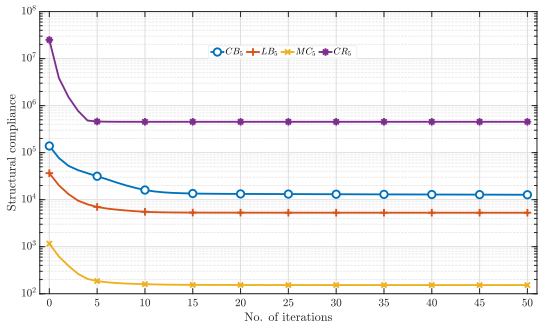


Fig. 10. Convergence plots for the largest mesh size of all examples using the ***ebe*64**−kernel.

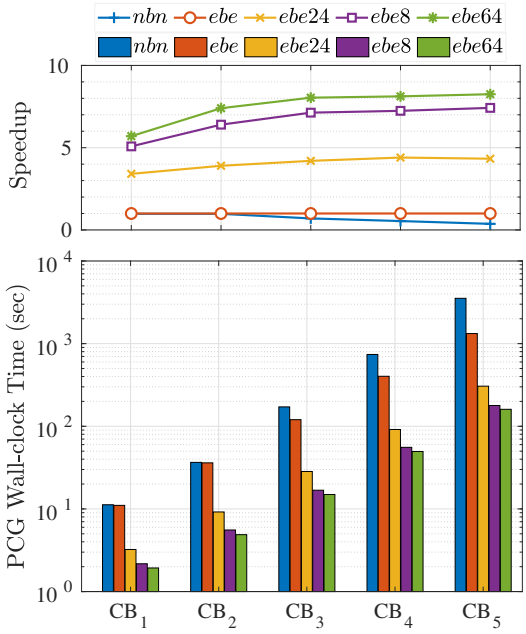


Fig. 11. PCG wall clock time, and speedup with respect to the ***ebe***−kernel for the 3D cantilever beam example.

5.6 Computational Performance

Computational performance of various strategies discussed in Section 3.4 and Section 4 is discussed for different mesh sizes. The wall-clock time of the PCG solver for each of the five GPU SpMV kernels is measured. This PCG wall-clock time represents the total wall-clock time taken by the solver in one iteration of optimization. The speedup is computed with respect to the standard ***ebe***−kernel discussed in Section 3.4.2. For the GPU instances, one dimensional grid of thread blocks were launched with 512 threads per block. For the ***ebe*24**−kernel, 504 threads per block were launched since 512 is not a multiple of 24. This block dimension for the ***ebe*24**−kernel can help transferring data from global memory to shared memory on GPU.

The PCG wall-clock time of all five GPU kernels for five mesh sizes is shown in Figure 11 for the 3D cantilever beam example. It can be seen that the ***nbn***−kernel consumes the most time for all mesh sizes, followed by the ***ebe***−kernel. The ***ebe*64**−kernel, on the other hand, requires the least amount of time. As a result, the ***ebe*64**−kernel shows the highest speedup, followed by the ***ebe*8**−kernel. For the ***ebe*8**− and ***ebe*64**− kernels, there is an increase in speedup with increasing mesh size. In addition, the ***ebe*24**−kernel outperforms the ***ebe***−kernel by nearly $3\times-4.5\times$ times for various mesh sizes. Despite the fact that the ***ebe*24**−kernel assigns more threads per element, it is still outperformed by the ***ebe*8**−kernel for all mesh sizes. The mesh CB_5 with the ***ebe*64**−kernel shows the highest speedup of $8.2\times$.

Figure 12 shows the PCG wall-clock time and speedup for the 3D L-beam example. Similar to the previous example, the ***nbn***−kernel consumes the most time, followed by the ***ebe***−kernel. It is clear from the figure that the ***ebe*8**−kernel outperforms all other kernels. A linear trend in speedup can be seen with respect to different mesh sizes. The ***ebe*64**−kernel is the second best performing kernel, but its speedup is nearly constant as the mesh size increases. The ***ebe*8**− and ***ebe*64**− kernels outperform the ***ebe*24**−kernel here as well. The ***ebe*8**−kernel for LB_5 mesh shows the maximum speedup of $8.2\times$ for the 3D L-beam example.

The PCG wall-clock time and speedup for the Michell cantilever example are shown in Figure 13. The ***nbn***−kernel has the highest wall-clock time. Over different mesh sizes, the speedup of ***ebe*24**−kernel ranges from $3\times-3.6\times$. It can be seen that the ***ebe*64**−kernel performs slightly better than the ***ebe*8**−kernel for smaller mesh sizes (MC_1 and MC_2). However, the performance of the ***ebe*8**−kernel is better for larger meshes. For the largest mesh MC_5, the ***ebe*8**−kernel achieves the highest speedup of $7.2\times$.

The PCG wall-clock time and speedup for the connecting rod example are shown in Figure 14. The PCG wall-clock time follows the similar trend with the previous examples. For smaller meshes, the ***ebe*64**−kernel outperforms the ***ebe*8**−kernel. However, the ***ebe*8**−kernel outperforms the ***ebe*64**−kernel for CR_4 and CR_5 meshes. The speedup of the ***ebe*24**−kernel ranges between $3.3\times-3.7\times$. In the connecting rod example, the highest speedup of $7.4\times$ is observed with the ***ebe*8**−kernel. This speedup, however, corresponds to the second largest CR_4 mesh. The same kernel shows a speedup of $7.1\times$ for the largest mesh size CR_5.

With the exception of the 3D cantilever beam example, it can be observed that the ***ebe*8**−is the best performing kernel in terms of speedup and scaling. The ***ebe*24**−kernel is outperformed by both the ***ebe*8**− and ***ebe*64**− kernels in all examples. Since the ***ebe*24**−kernel allocates 24 threads to each element that is not a multiple of the CUDA warp size (32), it causes thread divergence within a thread block leading to inferior performance. It is also worth noting that the ***ebe*8**−kernel outperforms the ***ebe*64**−kernel for larger

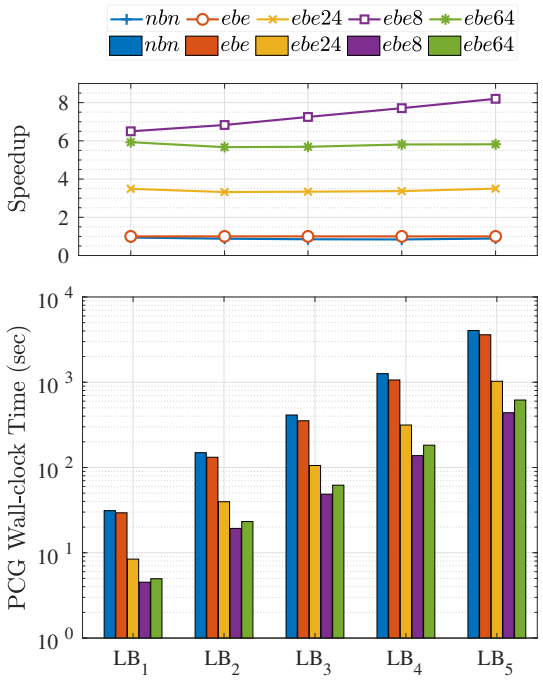


Fig. 12. PCG wall clock time, and speedup with respect to the ***ebe***−kernel of 3D L-beam example.

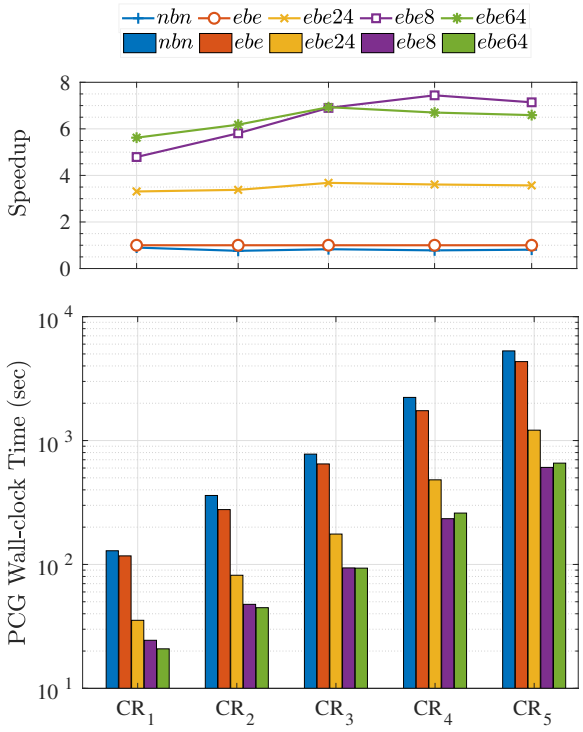


Fig. 14. PCG wall clock time, and speedup with respect to the ***ebe***−kernel of the connecting rod example.

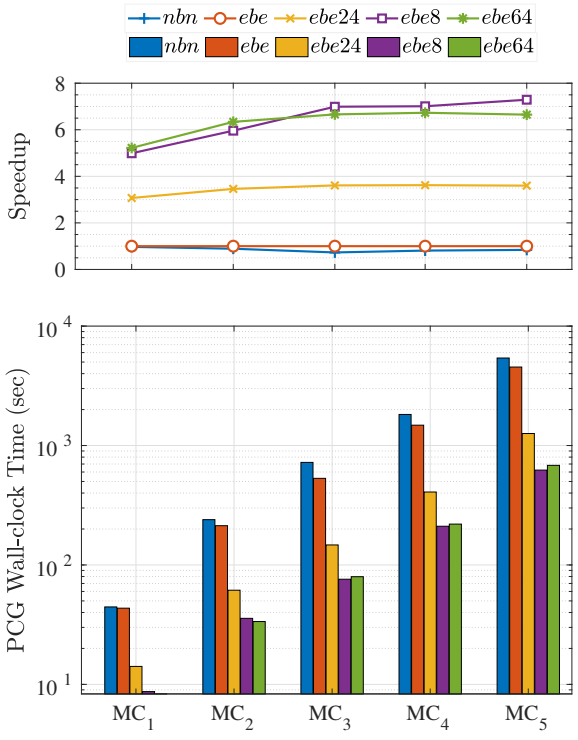


Fig. 13. PCG wall clock time, and speedup with respect to the ***ebe***−kernel of the Michell cantilever example.

unstructured all-hexahedral meshes. The ***ebe8***−kernel allocates 8 threads for reading 24×24 entries of K_e from global memory. However, the ***ebe64***−kernel allocates 64 threads that increases load of accessing the same amount of data by more number of threads from global memory resulting its inferior performance than the ***ebe8***−kernel.

The computational steps within the PCG solver include SpMV and many vector arithmetic (VeA) operations. Table 3 presents the percentage share of these two operations out of the total PCG wall-clock time for the largest mesh sizes of all four examples. The percentage of time listed in Table 3 is measured for the base implementation (***ebe***), and the best performing kernel (***ebe8***). It can be observed that for ***ebe***− kernel the SpMV takes $97.5\% - 99.4\%$ of total PCG time, which makes it the bottleneck for the solver. However, for ***ebe8***− kernel the percentage share of SpMV decreases to $82.36\% - 85.32\%$ while the percentage share of VeA increases to $14.68\% - 17.64\%$.

The various computational steps of topology optimization were shown earlier in Figure 1. The percentage of the total wall-clock time ($\%WC$) of these computational steps is shown in Figure 15. The percentages are calculated with respect to the total wall-clock time of the entire topology optimization using the largest mesh size of the corresponding example. The step 'pre-comp.' includes all preliminary computations, such as reading the mesh and connectivity data from files, allocating and initializing memories on GPU, and computing K_e. The step 'PCG' includes the total $\%WC$ taken by the PCG solver on GPU. 'Comp.' includes the computa-

Table 3. Percentage of time taken by SpMV and vector arithmetic operations in the PCG solver.

Mesh	***ebe***		***ebe*8**	
	SpMV	VeA	SpMV	VeA
CB_5	99.4%	0.6%	84.76%	15.24%
LB_5	97.8%	2.2%	82.36%	17.64%
MC_5	97.46%	2.54%	85.32%	14.68%
CR_5	97.89%	2.11%	83.17%	16.83%

tion of structural compliance and its sensitivities. 'Filter' includes the time taken by mesh-independency filter. 'Update' represents the update design variables and 'Output' represents the writing of results into files.

It can be observed from Figure 15 that even when the PCG solver is run on the GPU, it still consumes the most time of topology optimization. The PCG solver consumes $92\% - 99\%$ of the total wall-clock time with the ***nbn***− and ***ebe***−kernels. The $\%WC$ of other steps is insignificant in comparison to 'PCG'. When the proposed SpMV kernels are used, the $\%WC$ of 'PCG' is significantly reduced. For the ***ebe*8**−kernel, the $\%WC$ of 'PCG' ranges between $55\% - 77\%$, while 'pre-comp.' has the second highest $\%WC$ ranging between $13\% - 33\%$. For the ***ebe*24**−kernel, the $\%WC$ of 'PCG' varies between $68\% - 87\%$. The $\%WC$ of 'PCG' corresponding to the ***ebe*64**−kernel ranges between $53\% - 79\%$ for various examples.

6 Conclusion

This paper presented three ***ebe***−strategies for topology optimization of 3D structures using unstructured all-hexahedral mesh. These strategies were developed for performing SpMV computations in parallel on GPU with the matrix-free PCG solver of FEA. Since unstructured all-hexahedral mesh was used for descritizing the design domain, various computational challenges were addressed using the proposed strategies. SIMP method was used for testing the proposed strategies with the standard ***ebe***− and ***nbn***− strategies on four examples of three-dimensional design domains. Among the proposed strategies, the ***ebe*8**−strategy was found the best and outperformed the standard ***ebe***− and ***nbn***− strategies with $7.2\times - 8.2\times$ of speedup. It was found that the thread divergence and multiple transactions with global memory issues were encountered by the ***ebe*24**− and the ***ebe*64**− strategies, respectively. These issues led to an inferior performance with respect to the ***ebe*8**−strategy, even though more compute threads was assigned to these strategies. However, the ***ebe*24**− and ***ebe*64**− strategies were found better than the standard ***ebe***− and ***nbn***− strategies. As a note on future work, these strategies can be further improved by dealing with the issues of thread divergence and large memory transactions. Moreover, the proposed strategies can be used for solving compliant mechanism examples, compliance minimization with volume and stress constraints, to name a few.

References

[1] Bendsøe, M. P., and Kikuchi, N., 1988. "Generating optimal topologies in structural design using a homogenization method". *Computer methods in applied mechanics and engineering,* **71**(2), pp. 197–224.

[2] Bendsøe, M. P., 1989. "Optimal shape design as a material distribution problem". *Structural optimization,* **1**(4), pp. 193–202.

[3] Allaire, G., Jouve, F., and Toader, A.-M., 2002. "A level-set method for shape optimization". *Comptes Rendus Mathematique,* **334**(12), pp. 1125–1130.

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

[5] Sharma, D., Deb, K., and Kishore, N., 2011. "Domain-specific initial population strategy for compliant mechanisms using customized genetic algorithm". *Structural and Multidisciplinary Optimization,* **43**(4), pp. 541–554.

[6] Sharma, D., and Deb, K., 2014. "Generation of compliant mechanisms using hybrid genetic algorithm". *Journal of The Institution of Engineers (India): Series C,* **95**(4), pp. 295–307.

[7] Sharma, D., Deb, K., and Kishore, N., 2014. "Customized evolutionary optimization procedure for generating minimum weight compliant mechanisms". *Engineering Optimization,* **46**(1), pp. 39–60.

[8] Tomlin, M., and Meyer, J., 2011. "Topology optimization of an additive layer manufactured (alm) aerospace part". In Proceeding of the 7th Altair CAE technology conference, pp. 1–9.

[9] Zhu, J.-H., Zhang, W.-H., and Xia, L., 2016. "Topology optimization in aircraft and aerospace structures design". *Archives of Computational Methods in Engineering,* **23**(4), pp. 595–622.

[10] Sutradhar, A., Paulino, G. H., Miller, M. J., and Nguyen, T. H., 2010. "Topological optimization for designing patient-specific large craniofacial segmental bone replacements". *Proceedings of the National Academy of Sciences,* **107**(30), pp. 13222–13227.

[11] Coelho, P. G., Cardoso, J. B., Fernandes, P. R., and Rodrigues, H. C., 2011. "Parallel computing techniques applied to the simultaneous design of structure and material". *Advances in Engineering Software,* **42**(5), pp. 219–227.

[12] Elesin, Y., Lazarov, B. S., Jensen, J. S., and Sigmund, O., 2014. "Time domain topology optimization of 3d nanophotonic devices". *Photonics and Nanostructures-Fundamentals and Applications,* **12**(1), pp. 23–33.

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

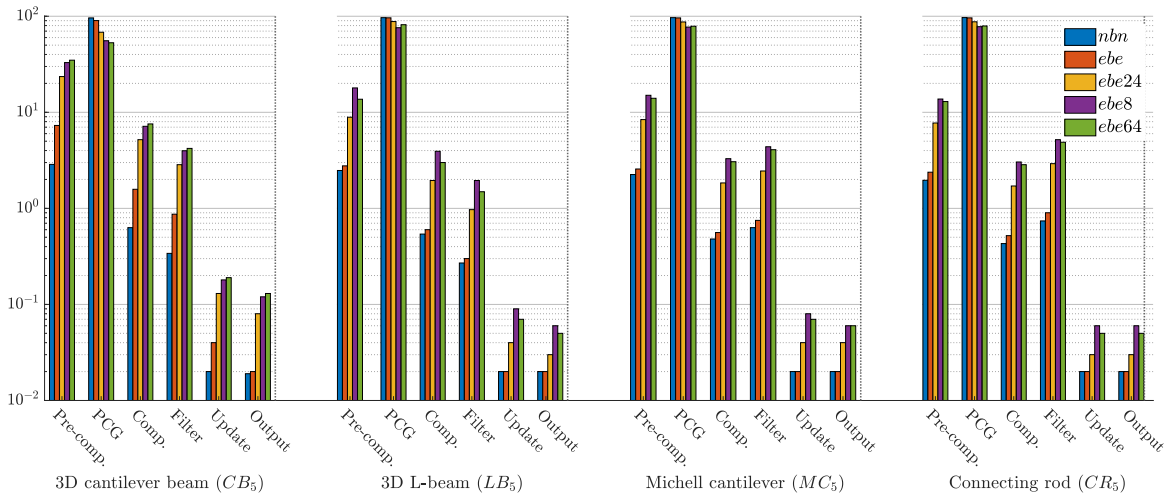


Fig. 15. Percentage of total wall-clock time of different computational steps corresponding to GPU kernels.

[14] Bendsøe, M. P., and Sigmund, O., 2013. *Topology optimization: theory, methods, and applications*. Springer Science & Business Media.

[15] Mahdavi, A., Balaji, R., Frecker, M., and Mockensturm, E. M., 2006. "Topology optimization of 2d continua for minimum compliance using parallel computing". *Structural and Multidisciplinary Optimization,* **32**(2), pp. 121–132.

[16] Koric, S., Lu, Q., and Guleryuz, E., 2014. "Evaluation of massively parallel linear sparse solvers on unstructured finite element meshes". *Computers & Structures,* **141**, pp. 19–25.

[17] Sanfui, S., and Sharma, D., 2018. "Gpu acceleration of local matrix generation in fea by utilizing sparsity pattern". In 1st International Conference on Mechanical Engineering (INCON 2018), Jadavpur University, India., pp. 1–4.

[18] Sanfui, S., and Sharma, D., 2020. "A three-stage graphics processing unit-based finite element analyses matrix generation strategy for unstructured meshes". *International Journal for Numerical Methods in Engineering,* **121**(17), pp. 3824–3848.

[19] Sanfui, S., and Sharma, D., 2017. "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, MNNIT Allahabad, India., pp. 1–9.

[20] Kiran, U., Sharma, D., and Gautam, S. S., 2019. "Gpu-warp based finite element matrices generation and assembly using coloring method". *Journal of Computational Design and Engineering,* **6**(4), pp. 705–718.

[21] Sanfui, S., and Sharma, D., 2019. "Exploiting symmetry in elemental computation and assembly stage of gpu-accelerated fea". In Proceedings at the 10th International Conference on Computational Methods (ICCM2019), ScienTech, pp. 641–651.

[22] Kiran, U., Agrawal, V., and Sharma, D., 2019. "A gpu based acceleration of finite element and isogeometric analysis"". In Proceedings at the 10th International Conference on Computational Methods (ICCM2019), ScienTech, pp. 641–651.

[23] Kiran, U., Gautam, S. S., and Sharma, D., 2020. "Gpu-based matrix-free finite element solver exploiting symmetry of elemental matrices". *Computing,* **102**(9), pp. 1941–1965.

[24] Duarte, L. S., Celes, W., Pereira, A., Menezes, I. F., and Paulino, G. H., 2015. "Polytop++: an efficient alternative for serial and parallel topology optimization on cpus & gpus". *Structural and Multidisciplinary Optimization,* **52**(5), pp. 845–859.

[25] Suresh, K., 2013. "Efficient generation of large-scale pareto-optimal topologies". *Structural and Multidisciplinary Optimization,* **47**(1), pp. 49–61.

[26] Schmidt, S., and Schulz, V., 2011. "A 2589 line topology optimization code written for the graphics card". *Computing and Visualization in Science,* **14**(6), pp. 249–256.

[27] Martínez-Frutos, J., and Herrero-Pérez, D., 2016. "Large-scale robust topology optimization using multi-gpu systems". *Computer Methods in Applied Mechanics and Engineering,* **311**, pp. 393–414.

[28] Zegard, T., and Paulino, G. H., 2013. "Toward gpu accelerated topology optimization on unstructured meshes". *Structural and multidisciplinary optimization,* **48**(3), pp. 473–485.

[29] Martínez-Frutos, J., Martínez-Castejón, P. J., and Herrero-Pérez, D., 2017. "Efficient topology optimization using gpu computing with multilevel granularity". *Advances in Engineering Software,* **106**, pp. 47–62.

[30] Kiran, U., Sanfui, S., Ratnakar, S. K., Gautam, S. S., and Sharma, D., 2019. "Comparative analysis of gpu-based solver libraries for a sparse linear system of equations". In *Advances in Computational Methods in Man-*

ufacturing. Springer, pp. 889–897.

[31] Martínez-Frutos, J., Martínez-Castejón, P. J., and Herrero-Pérez, D., 2015. "Fine-grained gpu implementation of assembly-free iterative solver for finite element problems". *Computers & Structures,* **157**, pp. 9–18.

[32] Cecka, C., Lew, A. J., and Darve, E., 2011. "Assembly of finite element methods on graphics processors". *International journal for numerical methods in engineering,* **85**(5), pp. 640–669.

[33] NVIDIA, C., 2013. Basic linear algebra subroutines (cublas) library.

[34] Sigmund, O., and Maute, K., 2013. "Topology optimization approaches". *Structural and Multidisciplinary Optimization,* **48**(6), pp. 1031–1055.

[35] Wadbro, E., and Berggren, M., 2009. "Megapixel topology optimization on a graphics processing unit". *SIAM review,* **51**(4), pp. 707–721.

[36] Ratnakar, S. K., Sanfui, S., and Sharma, D., 2020. "Simp-based structural topology optimization using unstructured mesh on gpu". In 2nd International Conference on Future Learning Aspects of Mechanical Engineering (FLAME), Amity University. U.P., India., pp. 1–8.

[37] Bell, N., and Hoberock, J., 2012. "Thrust: A productivity-oriented library for cuda". In *GPU computing gems Jade edition*. Elsevier, pp. 359–371.

[38] Ratnakar, S. K., Sanfui, S., and Sharma, D., 2020. "Gpu – based topology optimization using matrix-free conjugate gradient finite element solver with customized nodal connectivity storage". In 2nd International Conference on Future Learning Aspects of Mechanical Engineering (FLAME), Amity University. U.P., India., pp. 1–8.

[39] Sanders, E. D., Pereira, A., Aguiló, M. A., and Paulino, G. H., 2018. "Polymat: an efficient matlab code for multi-material topology optimization". *Structural and Multidisciplinary Optimization,* **58**(6), pp. 2727–2759.

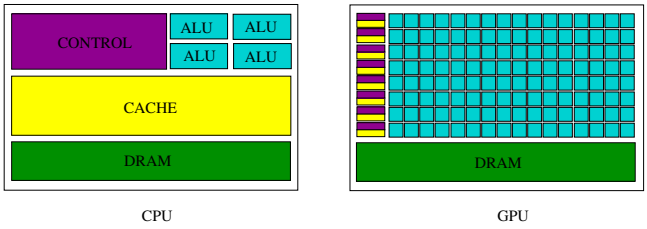
[40] Nana, A., Cuillière, J.-C., and Francois, V., 2016. "Towards adaptive topology optimization". *Advances in Engineering Software,* **100**, pp. 290–307.

7 Supplemental Material

GPU Architecture

GPU device was originally designed to address the need of faster 3D visualization in various applications. Now, it can be used for scientific computing since it offers high compute density, high throughput, and high latency tolerance for large computations in parallel. GPU devices are portable, low-cost, and need low maintenance compared to the traditional HPC setups. The modern day programming paradigms such as CUDA, OpenCL, etc. allow the user to use GPU as a general-purpose computing device and to take advantage of the data-level parallelism of the program by using multiple cores in parallel.

The architecture of GPU is vastly different from CPU as shown in Supp. Figure 1. The CPU is generally optimized for sequential code performance. It has high-capacity sophisticated control unit, which is connected to a small number of arithmetic logic units (ALUs). A large cache memory is provided in the CPU to reduce the latency in accessing data and instructions. In GPU, most of the chip area is dedicated to a large number of parallel compute threads. The individual processors on GPU are called streaming processors (SP) and they have their own register memory. A group of SPs together forms a streaming multiprocessor (SM). A control unit and cache are shared among SPs within SM.



Supp. Fig. 1. Architectures of CPU and GPU devices

GPU device consists of several types of memories. The performance of any application greatly depends upon the efficient usage of these memories. The types of memory, their location on the device, and their scope of access are listed in Supp. Table 1.

Supp. Table 1. Different types of memories available on a GPU device

Memory	Location	Scope of access
Register	On-chip	Thread
Local	On-chip	Thread
Shared	On-chip	Block
Global	Off-chip	Grid
Constant	Off-chip	Grid

The register and local memories are private to each thread. The shared memory is used by threads to communicate and share data. Individual threads can also access global memory, however, transactions between threads and global memory is slower than on-chip memories. Global memory is generally used to share and copy data from the host (CPU). The constant memory is read-only type, which remains unchanged throughout the program.

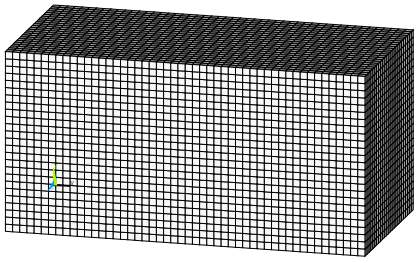
The parallel program that runs on GPU is defined by a function called kernel. The kernel is invoked by the host after specifying the number of threads. These threads are organized into batches known as thread blocks. The threads within a thread block can cooperate and share data with each other. However, the threads from one block cannot directly communicate with threads from another block. The total number of threads specified during the kernel launch is divided into these thread blocks, and this group of blocks is called a grid.

Various programming paradigms such as CUDA, OpenCL, and OpenACC are available to develop kernels for GPU. OpenCL and OpenGL are API-based programming models that work over GPU devices. CUDA is a programming model developed by NVIDIA to run on GPUs manufactured by them. CUDA works as an extension of C/C++/Fortran languages and is the most popular tool for writing GPU kernels. It provides flexibility and control to the programmer for building parallel applications using GPUs.

Discretized Domains of the Numerical Examples

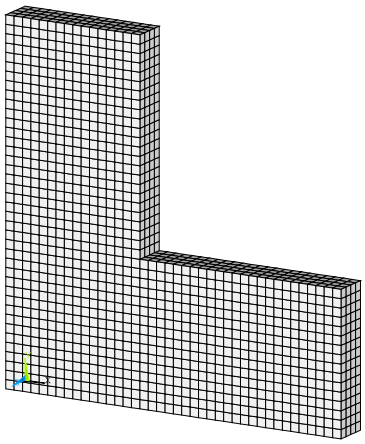
In this section, the discretized domains of the numerical examples used in this paper are discussed. The design domains of all four examples are meshed on ANSYS R16.1 APDL module, using 8−noded hexahedral elements.

Supp. Figure 2 shows the discretized domain of 3D cantilever beam example. It can be observed that the mesh is structured and composed of identical 8−noded hexahedral elements.



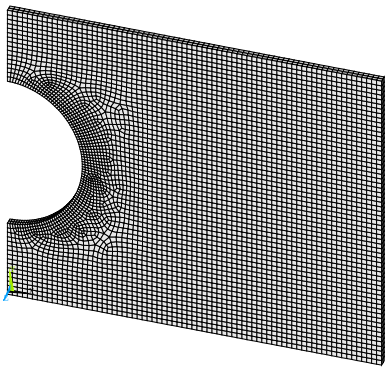
Supp. Fig. 2. Discretized domain of 3D cantilever beam with 8−noded hexahedral elements.

The discretized design domain of 3D L-beam example is shown in Supp. Figure 3. As can be seen, this example is also meshed using a structured mesh of 8−noded hexahedral elements.



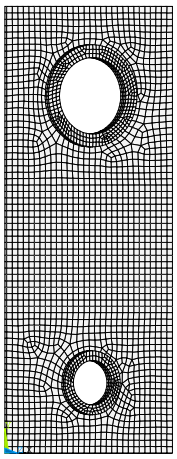
Supp. Fig. 3. Discretized domain of 3D L-beam with 8— noded hexahedral elements.

Supp. Figure 4 shows the discretized domain of the Michell cantilever example. It can be seen from the figure that there are unstructured hexahedral elements near the curved boundary due to its geometry.



Supp. Fig. 4. Discretized domain of Michell cantilever with 8— noded hexahedral elements.

In Supp. Figure 5, the discretized domain of the connecting rod is shown, and it can be seen that the mesh is unstructured in and around the area of both bores.



Supp. Fig. 5. Discretized domain of connecting rod with 8— noded hexahedral elements.