GPU-based Strategies for Accelerating Topology Optimization of 3D Continuum Structures using Unstructured Mesh

A Thesis Submitted
In Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Shashi Kant Ratnakar

(Roll No. 146103004)

Under Supervision of

Dr. Deepak Sharma

to the

DEPARTMENT OF MECHANICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

August, 2022

TH-3072_146103004

TH-3072_146103004

Abstract

Industrial product design is a complex and time-consuming process. Modern industry requirements call for products that yield high-performance but are also resource-efficient. Striking such a balance is immensely challenging for design engineers. For years, topology optimization has been utilized to develop the early conceptual design, which may then be developed into the final product. The computing cost of topology optimization, especially for large-scale problems, makes its widespread application more challenging. With the advancement of cost-effective many-core high performance computing architectures such as graphics processing units (GPU), the challenge of high computational cost has been addressed by the scientific computing community in recent years. Still, the development of efficient GPU computing strategies for the topology optimization of large-scale 3D continuum structures remains a challenge and an active area of research. The challenges are more pronounced in cases where unstructured meshes are used to discretize the models with complex geometry. Considering the benefits of topology optimization as a design tool, an efficient GPU computing strategy that accelerates the computational process will ultimately contribute to the development of a better design while simultaneously reducing the design cycle time. This thesis aims to address some of the major challenges in GPU-based acceleration of structural topology optimization discretized by 3D unstructured meshes. In the first part of this thesis, a GPU-based finite element analysis (FEA) solver is developed. The solver is equipped to handle large-scale 3D unstructured meshes efficiently. The proposed preconditioned conjugate gradient (PCG) FEA solver employs a matrix-free or assembly-free strategy in which GPU threads solve the system of linear equations at the elemental level. In comparison to the corresponding CPU-based version, the proposed FEA solver exhibits considerable speedups across several benchmark examples of varying sizes. The second part of the thesis focuses on enhancing the computational performance of the proposed GPU-based

TH-3072_146103004

FEA solver by developing efficient GPU thread allocation strategies. Utilizing the massive parallelism of the GPU, the proposed thread allocation strategies distribute the computational load of a single GPU thread across a large number of threads. This thesis proposes three such fine-grained thread allocation strategies. The analysis of the results demonstrates a significant improvement in the computational performance as compared to the conventional GPU-based strategy from the literature. Using the symmetry property of elemental stiffness matrices, the proposed GPU-based FEA solver is further improved by reducing the amount of CPU-GPU data transfer. This is accompanied by the development of novel data storage and data access patterns on the GPU, which reduces the amount of GPU memory required by the FEA solver. The efficient GPU computing strategies proposed in this thesis accelerate the major computational bottlenecks of the GPU-based matrix-free FEA solver, resulting in a topology optimization framework with much less execution time and reduced memory consumption.

Indian Institute of Technology Guwahati

TH-3072_146103004

CERTIFICATE

It is certified that the work contained in the thesis entitled "**GPU-based Strategies for Accelerating Topology Optimization of 3D Continuum Structures using Unstructured Mesh**", by "Mr. Shashi Kant Ratnakar", has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Deepak Sharma

27th August, 2022.

Department of Mechanical Engineering,

I.I.T. Guwahati.

TH-3072_146103004

TH-3072_146103004

Dedicated to

Sudarshan Lal Ratnakar (my father)

Savita Ratnakar (my mother)

Ravi and Shraddha (my siblings)

Swati (my wife)

TH-3072_146103004

TH-3072_146103004

Acknowledgement

First, I would like to express my sincerest gratitude to my thesis supervisor, Dr. Deepak Sharma for his guidance and support during my Ph.D. research. This research work would not have come to its fruition without his encouragement, motivation, and expertise. His mentorship has enabled me to transition from an engineering graduate to a researcher. It was a privilege to work under his guidance.

I would also like to thank the members of my doctoral committee, Prof. Debabrata Chakraborty, Dr. Sachin Singh Gautam, and Prof. Arbind Kumar Singh, for their valuable feedback and suggestions that were crucial to the completion of this thesis.

I am eternally grateful to my parents for their unwavering faith in me, which served as a continuous source of encouragement for me during this journey. I would also like to thank my wife for her invaluable support and encouragement.

I would like to thank my friends Lt. Col. Navjot Singh Sanghu, Dr. Agyapal Singh, Dr. Akash Anil, Dr. Vishal Agrawal, Dr. G. Saipraneeth, Dr. Subhajit Sanfui, Raktim, and Utpal, for making my time at the Indian Institute of Technology, Guwahati, exceedingly memorable and enjoyable.

Finally, I express my thanks to all those who have helped me directly or indirectly for successful completion of this work.

Shashi Kant Ratnakar
IIT Guwahati
August, 2022

TH-3072_146103004

TH-3072_146103004

Journal Publications

- Shashi Kant Ratnakar, Subhajit Sanfui and Deepak Sharma, 2022, "Graphics Processing Unit-Based Element-by-Element Strategies for Accelerating Topology Optimization of Three-Dimensional Continuum Structures Using Unstructured All-Hexahedral Mesh", ASME Journal of Computing and Information Science in Engineering, 22(2), 1–11. https://doi.org/10.1115/1.4052892

- Shashi Kant Ratnakar, Utpal Kiran and Deepak Sharma, 2022, "Acceleration of Structural Topology Optimization using Symmetric Element-by-Element Strategy for Unstructured Meshes on GPU", Engineering Computations, 39(10), 3354 – 3375. https://doi.org/10.1108/EC-01-2022-0022

Conference Publications

- Shashi Kant Ratnakar, Subhajit Sanfui and Deepak Sharma, 2021, "SIMP-Based Structural Topology Optimization Using Unstructured Mesh on GPU", *in* N. Kumar, S. Tibor, R. Sindhwani, J. Lee and P. Srivastava, eds, 'Advances in Interdisciplinary Engineering: Select Proceedings of FLAME 2020', Springer, pp 1–10, https://doi.org/10.1007/978-981-15-9956-9_1.

- Shashi Kant Ratnakar, Subhajit Sanfui and Deepak Sharma, 2021, "GPU-Based Topology Optimization Using Matrix-Free Conjugate Gradient Finite Element Solver with Customized Nodal Connectivity Storage", *in* N. Kumar, S. Tibor, R. Sindhwani, J. Lee and P. Srivastava, eds, 'Advances in Interdisciplinary Engineering. : Select Proceedings of FLAME 2020', Springer, pp 87 – 97, https://doi.org/10.1007/978-981-15-9956-9_9.

Other Publication

- Utpal Kiran, Subhajit Sanfui, Shashi Kant Ratnakar, Sachin Singh Gautam, and Deepak Sharma, "Comparative Analysis of GPU-based Solver Libraries for A Sparse Linear System of Equations", *in* 2nd International Conference on Computational Methods in Manufacturing (ICCMM), 8 – 9 March 2019, IIT Guwahati, India.

TH-3072_146103004

TH-3072_146103004

Contents

List of Figures **xv**

List of Tables **xix**

1 Introduction **1**
1.1 Motivation 5
1.2 Objectives of the Thesis 7
1.3 Organization of Thesis 7

2 Preliminaries and Literature Survey **9**
2.1 Solid Isotropic Material with Penalization (SIMP) 9
2.2 SIMP-based Topology Optimization 10
2.3 Conjugate Gradient Solver and Preconditioning 15
2.4 GPU Computing 18
2.5 Literature Review 21
2.5.1 Parallel Computing for SIMP-based Topology Optimization . . . 21
2.5.2 GPU Acceleration of SIMP-based Topology Optimization 23
2.5.2.1 Using Structured Mesh 23
2.5.2.2 Using Unstructured Mesh 24
2.5.3 GPU Acceleration of Other Topology Optimization Methods . . . 26
2.5.4 Open-Source Parallel Topology Optimization Frameworks 27
2.5.5 Closure 27

3 GPU-based Matrix-Free FEA Solver for Structural Topology Optimization using Unstructured Mesh **29**
3.1 Matrix-Free PCG Solver on GPU 30
3.1.1 Proposed Element-by-Element SpMV Strategy '***ebe***' 34
3.1.2 Proposed Node-by-Node SpMV Strategy '***nbn***' 36
3.1.2.1 Proposed Customized Nodal Connectivity Storage Format '$\mathbf{C}_{rev}$' 36
3.1.2.2 Algorithm for '***nbn***' Strategy 38
3.2 Topology Optimization using Proposed FEA Solver 38

TH-3072_146103004

3.3 Benchmark Examples . . . 40
3.3.1 3D Cantilever Beam . . . 41
3.3.2 3D L-Beam . . . 41
3.3.3 Michell Cantilever . . . 42
3.3.4 Connecting Rod of an Auto-mobile Engine . . . 44
3.4 Results and Discussion . . . 46
3.4.1 Optimal Topology . . . 46
3.4.2 Computational Performance . . . 47
3.4.3 Closure . . . 53

4 Efficient Thread Allocation Strategies for Matrix-Free FEA Solver 55
4.1 Fine-Grained Thread Allocation Strategies for SpMV on GPU . . . 57
4.1.1 8-Threads per Element Strategy '*ebe8*' . . . 57
4.1.2 24-Threads per Element Strategy '*ebe24*' . . . 58
4.1.3 64-Threads per Element Strategy '*ebe64*' . . . 59
4.2 Computational Performance . . . 60
4.2.1 Closure . . . 69

5 Novel Data Storage and Data Access Patterns for Matrix-Free FEA Solver 71
5.1 Exploiting Symmetry of Elemental Stiffness Matrices . . . 72
5.2 Customized Data Storage Format . . . 72
5.2.1 Storage Format for Diagonal Sub-Matrices . . . 73
5.2.2 Storage Format for Off-Diagonal Sub-Matrices . . . 74
5.3 Proposed Symmetry-based SpMV Strategy '***ebeSym***' . . . 75
5.4 Computational Performance . . . 78
5.5 Closure . . . 82

6 Conclusions and Future Work 85
6.1 Conclusions . . . 85
6.2 Future Work . . . 87

References 98

TH-3072_146103004

List of Figures

2.1 Computational steps of SIMP-based structural topology optimization. . . 11

2.2 Sensitivity filtering for an element over a neighborhood defined by the filter radius '$\mathcal{R}$'. 14

2.3 Simplified model of CPU and CPU architectures. 19

3.1 Element-by-element SpMV strategy. 33

3.2 Node-by-node SpMV strategy. 33

3.3 Matrix-vector multiplication by the proposed GPU-based element-by-element SpMV strategy '***ebe***'. 34

3.4 A patch of 2D elements with one DoF per node, and their internal node numbering scheme. 37

3.5 Customized nodal connectivity storage format '$\mathbf{C}_{rev}$'. 37

3.6 Flowchart of SIMP method-based topology optimization framework using the proposed matrix-free PCG solver. 40

3.7 3D Cantilever beam example. 42

3.8 3D L-beam example. 43

3.9 Michell cantilever example. 44

3.10 Connecting rod of an auto-mobile engine example. 45

TH-3072_146103004

3.11 Evolution of optimal topology of 3D cantilever beam example corresponding to the mesh CB_5. 47
3.12 Evolution of optimal topology of 3D L-beam example corresponding to the mesh LB_5. 48
3.13 Evolution of optimal topology of Michell cantilever example corresponding to the mesh MC_5. 49
3.14 Evolution of optimal topology of connecting rod example corresponding to the mesh CR_5. 50
3.15 PCG execution time for ***nbn*** and ***ebe*** strategies and the gained speedups against their respective CPU-based single thread versions ***nbnCPU*** and ***ebeCPU***. 52
3.16 Percentage share of various computational steps of topology optimization framework from the wall-clock time, corresponding to the proposed ***ebe*** and ***nbn*** strategies. 54

4.1 Percentage share of SpMV and VeA operations from PCG execution time. 56
4.2 Matrix-vector multiplication by the proposed ***ebe8*** strategy. 58
4.3 Matrix-vector multiplication by the proposed ***ebe24*** strategy. 60
4.4 Matrix-vector multiplication by the proposed ***ebe64*** strategy. 62
4.5 PCG execution time, and speedup with respect to the ***ebe*** strategy for 3D cantilever beam example. 63
4.6 PCG execution time, and speedup with respect to the ***ebe*** strategy for 3D L-beam example. 64
4.7 PCG execution time, and speedup with respect to the ***ebe*** strategy for Michell cantilever example. 66

TH-3072_146103004

4.8 PCG execution time, and speedup with respect to the ***ebe*** strategy for Connecting rod example. 67
4.9 Percentage share of various computational steps of topology optimization framework from the wall-clock time, corresponding to the proposed fine-grained SpMV strategies. 68

5.1 A symmetric elemental stiffness matrix is shown. The entries from the upper triangular half ($\mathbf{K}_{sym}$) are used for SpMV. 73
5.2 Storage format for the entries of diagonal sub-matrices. 74
5.3 Storage format for the entries of off-diagonal sub-matrices. 75
5.4 Thread assignment and matrix-vector multiplication by the proposed ***ebeSym*** strategy for one group of sub-matrices. 76
5.5 PCG execution time and speedup of the ***ebeSym*** strategy against ***ebe*** and ***ebe8*** strategies for various mesh sizes of four benchmark examples. . 80
5.6 Percentage contributions of various computational steps of topology optimization with respect to the wall-clock time. 81
5.7 GPU memory required by the ***ebe8*** and ***ebeSym*** strategies for different mesh sizes. 82

TH-3072_146103004

TH-3072_146103004

List of Tables

2.1 Different types of memories available on a GPU device 20

3.1 Common parameters used for topology optimization of three examples. . 41

3.2 Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of 3D cantilever beam example. 42

3.3 Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of 3D cantilever beam example. 43

3.4 Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of Michell cantilever example. 44

3.5 Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of connecting rod example. 45

3.6 Average number of PCG iterations taken to converge by different mesh sizes of four benchmark examples. 51

TH-3072_146103004

TH-3072_146103004

Chapter 1

Introduction

Various industries and engineering disciplines require lightweight structures that can be manufactured with lesser amount of material while still fulfilling their strength and structural performance requirements [1, 2]. One of the viable methods to develop such structures is structural topology optimization. It is an application of optimization that enables the design engineers to optimally distribute the material within a design space, taking into consideration the given loads, boundary conditions, and other performance constraints. The ultimate result is a topology that is sufficiently robust while making efficient use of the limited material available, striking a balance between desired strength and material costs.

In the design cycle of a product or component, topology optimization can be used either at the start for exploring conceptual designs or at the end to lightweight a working design. Topology optimization as a design tool is very useful in industries where a small reduction in weight can lead to significant improvement in part performance and cost savings. Aerospace industry is a primary example of this [3, 4] where it is used to optimize the design of wing structures, turbine blades, and engine parts, etc. Automobile industry is another such example where topology optimization is used to design lightweight and structurally efficient components [5, 6]. In biomedical engineering topology optimization has been used to design custom bone replacements [1] and to perform multiscale simulations of bone remodelling [7]. In the field of multi-physics design it has been applied to the development of 3D nanophotonic devices [8] and 3D electrothermo-

TH-3072_146103004

mechanical actuators [9]. In compliance mechanism design topology optimization has been used extensively [10–12]. In the recent past, additive manufacturing has utilized topology optimization to produce designs with complex geometries and intricate internal structures, such as lattices and flow channels [13, 14].

Bendsøe and Kikutchi introduced a homogenization method for topology optimization in their pioneering study [15]. This study laid the foundation for the development of numerical methods for topology optimization of continuum structures. Several topology optimization methods have been developed over the years, which can be categorized as follows.

- **Density-based methods:** these methods work on a fixed domain of finite element. Each element is assigned an artificial density variable, and the objective function is minimized by determining whether each element should be filled with material or left empty. Two methods in this category are homogenization method [15] and solid isotropic material with penalization (SIMP) method [16, 17]. Density-based methods are widely applied for problems that require material reduction or redistribution, such as weight reduction in aerospace structures [18] or material savings in manufacturing processes [19].

- **Boundary variation methods:** these methods are based on implicit functions that determine structural boundaries. The optimal topology is achieved by moving these boundaries. This category includes the phase-field method [20, 21] and the level-set method [22–24]. A limitation of boundary variation methods is that the topology evolves from the pre-existing geometry, and no additional holes may be formed during optimization. However, Yamada et al. [25] suggest a 'relaxed level-set' method that permits the formation of new holes. These methods are suitable for problems that involve optimizing the geometry of a specific part or region of a structure, such as microstructure design [26], and cellular composite design [27].

- **Topological derivatives-based method:** Eschenauer et al. [28] were the first to apply topological derivatives with topology and shape optimization methods. The fundamental idea is to estimate the effect (derivative) of introducing an infinitesimal hole at a particular location within the design domain and use this as the control for the formation of new holes. This method allows designers to ob-

TH-3072_146103004

tain a better understanding of the influence of the design on the structure and the underlying physics.

- **Discrete methods:** these methods work by gradually removing/adding material form the design domain based on a heuristic criterion. Since finite elements are explicitly defined as 'solid' or 'void', the final topology is free of intermediate 'gray' material. The most popular methods in this category are evolutionary structural optimization (ESO) [29] and bi-directional evolutionary structural optimization (BESO) [30, 31]. These methods are often used for problems with a large number of design variables that can be computationally expensive. Additionally, discrete methods allow the manipulation of individual elements in a discrete design space. This level of control over the design makes discrete methods particularly useful in applications where high precision is required such as compliant mechanism design [32].

Density-based methods (SIMP in particular) are the most popular in the literature. First, its implementation over the domain represented by finite elements is easy. Secondly, they represent smooth, differentiable problems that may be efficiently solved by gradient-based optimization techniques such as the optimality criteria (OC) method [33], method of moving asymptotes (MMA) [34], etc.

Evidently, structural topology optimization is an important tool for determining the optimal material layout early in the design process. However, the primary challenge with its implementation is the requirement of large computation time [35]. The computational steps of a density-based structural topology optimization include meshing, finite element analysis (FEA), objective function computation, sensitivity analysis, mesh-independency filter, and design variable update. Among these steps, FEA is the most computationally expensive step since it includes computation of elemental stiffness matrices in case of unstructured mesh, assembly of these elements into a global stiffness matrix [36] and solving the linear system of equations [37].

The need for a dense finite element (FE) mesh is one of the key contributors for higher computational time required for topology optimization. There are several practical benefits of using a dense FE mesh. In the case of complex geometries, a dense FE mesh can represent the geometry accurately. This is especially true for non-uniform or irregular geometries [38]. In addition, some geometric features, such as sharp corners or narrow

TH-3072_146103004

channels, could require a finer mesh for accurate capture [39]. In some cases, dense mesh can aid in accurately capturing local phenomena such as stress concentration [40]. Using a dense FE mesh can also improve the accuracy of topology optimization results, since a finer mesh can capture more detail and provide a more accurate representation of the structure being optimized [15]. Lastly, using a coarser mesh could give rise to mesh-dependent solutions, resulting in infeasible and inaccurate solutions. Using a dense FE mesh can help avoid this issue by ensuring that the results are less dependent on mesh density [41].

Several strategies have been explored in the literature to resolve the challenge of higher computation time with FEA. Changes at the algorithmic level, including techniques such as adaptive mesh refinement and reduced order modeling, are some of the remedies. Leveraging the capabilities of high-performance computing (HPC) is another implementation-level strategy for addressing this challenge. Although HPC is defined as the the ability to process data and perform complex calculations at high speeds, this term has been used interchangeably with parallel computing.

Since the FEA solver is the primary computational bottleneck in topology optimization, the majority of published research focuses on speeding up the FEA solver through the use of parallel computing. In the literature, conventional parallel computing frameworks like distributed memory architecture [42, 43] and shared memory architectures [44] have been used successfully to accelerate the FEA solver of topology optimization. Among them, GPUs have gained popularity as the preferred HPC architecture for accelerating topology optimization among researchers in recent times. GPUs provide massive parallelism at a reasonable cost compared to conventional parallel architectures. The GPU's many-core, multi-threaded processing architecture makes it ideally suited for data-parallel applications such as topology optimization. GPUs have been used to accelerate topology optimization for a variety of applications, including the design of a tied-arc bridge and compliant gripper mechanism [45], the design of a heat sink [46], and the design of an airline passenger seat frame [47], among others.

The literature shows that GPU may significantly accelerate the FEA solver for topology optimization. It is also observed that the implementation strategy of the FEA solver on GPU is highly dependent on the type of mesh used to discretize the design domain. In numerical simulations, structured and unstructured meshes are two common categories of meshes. Structured meshes comprise identically shaped and sized cells arranged in a

TH-3072_146103004

regular grid. Unstructured meshes, on the other hand, contain an irregular arrangement of cells, with each cell having a different shape and size [48]. Structured meshes are efficient for discretizing structures with regular domain geometry. GPU implementation is also easier to handle when structured meshes are used. For the same reason, structured meshes have been used by the majority of the studies in the literature dedicated to GPU-based acceleration of structural topology optimization [45, 49, 50]. Unstructured meshes can handle complex geometries with arbitrary shapes and sizes more easily than structured meshes. However, they are more challenging to efficiently handle on GPU, and have been used by a few studies in the literature [51, 52].

1.1 Motivation

GPUs can speedup the FEA solver of topology optimization by many-folds. However, executing FEA solver efficiently on a GPU poses a number of challenges. The first significant challenge is the development of an efficient kernel that can harness the GPU's massive parallelism. This is accomplished by ensuring optimal load distribution among GPU threads and preventing thread divergence, thread idling, and race-condition issues. The second challenge is the effective use of GPU memories, such as usage of shared memory for inter-thread communications, minimization of global memory read operations, etc.

When structured meshes are utilized for discritization of the design domain, these challenges are comparatively easier to address. Generally, the same elemental stiffness matrix can be used for the entire mesh, and the connectivity information can be obtained by simple mapping, avoiding the need to store large amounts of data. In real−life applications of topology optimization, however, the domain geometry may be complicated, irregular, and have curved boundaries. In such instances, unstructured meshes are utilized to more accurately represent the design domain. Unstructured meshes pose a number of additional computational challenges on the GPU, including a huge memory space requirement for storing elemental and connectivity data, and achieving proper load balancing among GPU threads. Through a comprehensive review of the relevant literature, several research gaps are identified in the field. This thesis seeks to address these challenges by developing GPU-based implementation strategies for unstructured

TH-3072_146103004

mesh with topology optimization of large-scale 3D structures. While it is difficult to clearly define the term 'large-scale problems', considering the trends in the literature it can be termed as the problems containing a large number of design variables, often in the range of million(s).

The assembly of elemental stiffness matrices into a global stiffness matrix is a time consuming operation, according to the literature [53, 54]. This thesis aims to alleviate this issue by adopting the matrix-free or assembly-free strategy, which performs computations in parallel at the local level.

The matrix-free iterative solver comprises of two types of operations: sparse matrix-vector multiplications (SpMV) and vector arithmetic operations. Compared to vector arithmetic operations, the SpMV operations on GPU are more complicated and consumes from 80% to 99% of the solver's execution time, hence becoming the bottleneck of the FEA solver. For efficient SpMV execution on a GPU, it is essential to maintain an optimal balance of computational load among GPU threads. In this thesis, various thread allocation strategies are developed for SpMV on GPU to address this challenge.

Execution of matrix-free FEA solver using unstructured meshes on a GPU requires a huge amount of CPU to GPU data transfer, and a large number of global memory read operations. In the literature, the symmetry property of elemental stiffness matrices has been used for SpMV using 2D meshes [55] and for 2D and 3D polygonal meshes [52]. It has also been used to perform the assembly [51]. This thesis aims to reduce the amount of CPU-GPU data transfer by using only the symmetric half of elemental stiffness matrices, and to optimize the data accesses on GPU by developing novel data storage and access formats.

To summarize, the aim of this thesis is to develop an efficient GPU-based FEA solver for topology optimization of large-scale 3D continuum structures using unstructured meshes. The FEA solver used in this thesis adopts a matrix-free approach where the system of linear equations is solved at the elemental level by GPU threads. The thesis later focuses on enhancing the computational performance of GPU-based matrix-free FEA solver by tackling the challenges posed by unstructured meshes on GPU. This is accomplished through creating efficient thread allocation strategies that ensure a balanced computational load among GPU threads, as well as GPU-specific data storage formats that optimize data accesses on a GPU while reducing CPU-GPU data transfer.

TH-3072_146103004

1.2 Objectives of the Thesis

Following are the objectives of the thesis.

1. Develop a GPU-based matrix-free FEA solver for structural topology optimization using unstructured meshes.

2. Develop efficient thread allocation strategies for matrix-free FEA solver.

3. Develop novel data storage and data access patterns for matrix-free FEA solver.

4. Performance analysis of the proposed GPU-based matrix-free FEA solver over benchmark examples and comparing it with the state-of-the-art strategies on GPU.

In the following chapters, the strategies proposed to achieve the objectives of the thesis are discussed in detail.

1.3 Organization of Thesis

The rest of the thesis is organized as follows.

- **Chapter 2** discusses the theoretical and implementational aspects of the structural topology optimization. The fundamentals of GPU computing are discussed, followed by a literature review on using GPUs to accelerate density-based structural topology optimization methods.

- **Chapter 3** presents the GPU-based matrix-free FEA solver for structural topology optimization for large-scale 3D continuum structures using unstructured meshes. The popular strategies for accelerating matrix-free FEA solver on GPU, their implementation, and performance over benchmark examples are discussed in this chapter.

- In **Chapter 4** fine-grained SpMV strategies to accelerate the matrix-free FEA solver on GPU are discussed. The proposed three SpMV strategies are tested over benchmark examples and their computational performance is compared with the standard strategy form the literature.

TH-3072_146103004

- **Chapter 5** presents the ***ebeSym*** strategy that uses the symmetric half of the elemental stiffness matrices, along with two novel data storage formats to ensure optimized data accesses on GPU. The results of the proposed strategy are presented and compared with the standard strategy.

- **Chapter 6** presents the conclusions drawn from the thesis along with a note on future work.

TH-3072_146103004

Chapter 2

Preliminaries and Literature Survey

In this chapter a detailed introduction of SIMP method, topology optimization, and GPU computing is presented, followed by the relevant literature on GPU acceleration of SIMP method-based structural topology optimization.

2.1 Solid Isotropic Material with Penalization (SIMP)

In the density-based topology optimization method, an 'artificial' density variable 'ρ_e' is introduced with each mesh element. ρ_e is referred to as 'artificial' density given the fact that its value does not represent the physical density of the finite element, but rather the presence (or absence) of solid material within the finite element. ρ_e becomes design variable for the topology optimization problem, and its value lies in the range of $0 < \rho_{min} \leq \rho_e \leq 1$, where $\rho_e = 1$ represents a 'solid' and $\rho_e = \rho_{min}$ represents a 'void'. The minimum density value limit 'ρ_{min}' is imposed to prevent numerical instabilities such as singularity in finite element matrices and, in certain cases, the problem of material not reappearing in a zero density area.

The SIMP method penalizes intermediate ρ_e values iteratively to steer the topology toward a 'solid'-'void' or '0'-'1' solution. This is achieved by penalizing the density variables with a parameter 'p'. The relationship between the elemental density and material property of the 'e^{th}' element is given by the power law as shown in equation (2.1).

TH-3072_146103004

$$\{E_{ijkl}\}_e = {\rho_e}^p \; E^0_{ijkl}, \qquad p > 1,$$
$$E_{ijkl} = \begin{cases} 0, & \text{if} \quad \rho_e = 0, \\ E^0_{ijkl}, & \text{if} \quad \rho_e = 1, \end{cases} \tag{2.1}$$

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

The value of p must be chosen carefully. Too low value of p may result into too many intermediate densities, and too high value of it may result into too fast convergence to local minima. Several methods are reported in the literature for deciding the penalization parameter's value. An example of this is the 'continuation method,' in which the initial optimization iteration uses $p = 1$, and the number gradually increases in the following iterations [56].

The majority of studies from literature use $p = 3$ as the penalization parameter throughout the optimization. However, the rationale for selecting this specific number is more empirical than technical [57]. The value $p = 3$ has been found to be a good compromise between accuracy and computational efficiency [58]. Additionally, for most practical applications, values of $p > 3$ do not yield significant improvements in the optimization results [59]. In this thesis, $p = 3$ is used as penalization parameter.

2.2 SIMP-based Topology Optimization

The structural topology optimization problem is formulated as compliance minimization with volume constraint [16] that is shown in equation (2.2).

$$\begin{aligned} \min_{\boldsymbol{\rho}} \quad & C(\boldsymbol{\rho}, \boldsymbol{u}) = \boldsymbol{u}^T \mathbf{K} \boldsymbol{u}, \\ \text{subject to :} \quad & \mathbf{K}(\boldsymbol{\rho})\boldsymbol{u} = \boldsymbol{f}, \\ & V(\boldsymbol{\rho})/V_d \leq V_f, \\ & 0 < \rho_{min} \leq \rho_e \leq 1, \quad e \in \Omega, \end{aligned} \tag{2.2}$$

TH-3072_146103004

where C denotes the structural compliance, $\boldsymbol{\rho}$ is the vector of density variable, $\boldsymbol{u}$ is the nodal displacement vector, $\mathbf{K}$ is the global stiffness matrix, and $\boldsymbol{f}$ is the global load vector. $V(\boldsymbol{\rho})$ is the final volume of the structure, and V_d is the volume of original design domain. The ratio of these two volumes is defined as 'volume fraction' (V_f) and its value is decided by the user based on the permissible amount of material. The last constraint puts a bound over the design variable value.

In this thesis the optimality criteria (OC) method is used to solve the topology optimization problem given in equation (2.2). OC method is suitable for problems with a large number of design variables and a single constraint. The OC-based structural topology optimization is an iterative method that involves various computational steps which are shown in Figure 2.1. These steps include the following:

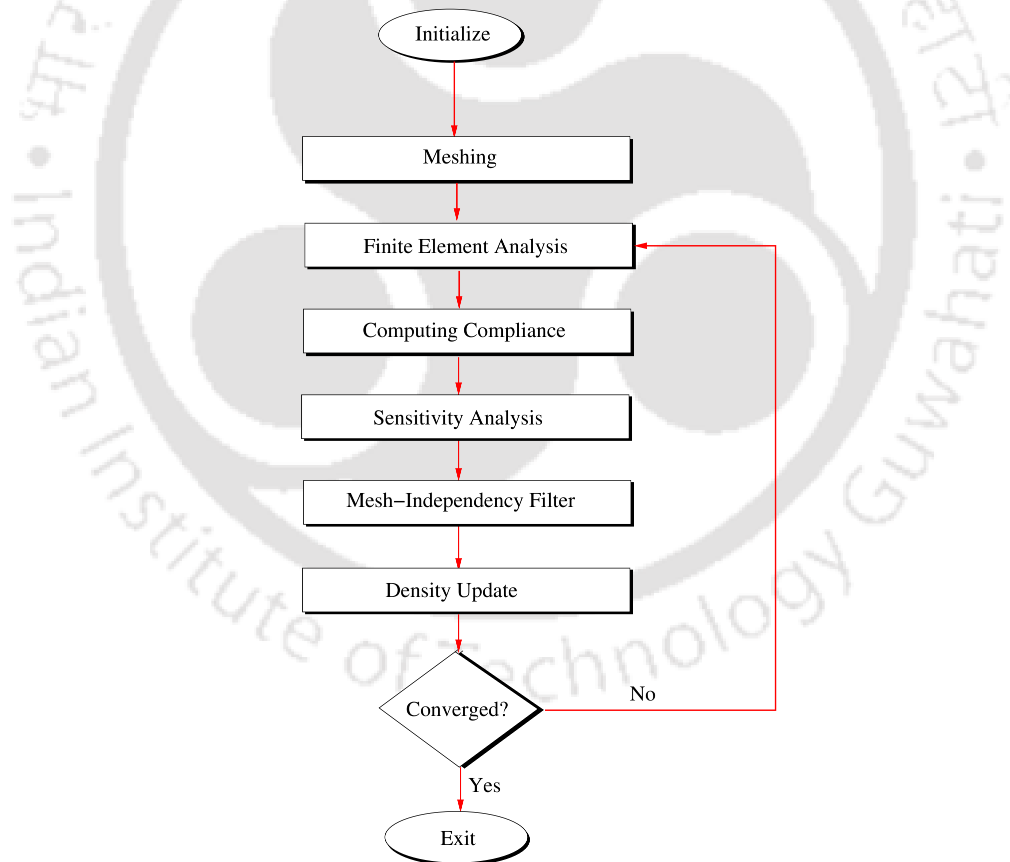


Figure 2.1: Computational steps of SIMP-based structural topology optimization.

TH-3072_146103004

Initialization:

Design domain, loading, and boundary conditions of the given problem are defined at this step. Other parameters such as ρ_{min}, V_f, and termination criteria for optimization, etc. are also decided.

Meshing:

The design domain is discretized into finite elements. For simple domain geometries, meshing can be done by the program itself. Another way is to perform meshing using a commercial software and import the mesh data into the framework. In this thesis, the Ansys APDL module is used to perform the meshing using 8-noded hexahedral elements.

Finite Element Analysis:

FEA is then performed to generate the structural response under the specified loading and boundary conditions. The nodal displacements are computed by solving the linear system of equations given in equation (2.3).

$$\left[\sum_{e=1}^{N_e} \rho_e^p \, \mathbf{K}_e\right] \boldsymbol{u} = \boldsymbol{f}, \tag{2.3}$$

where N_e is the total number of finite elements in the mesh, ρ_e^p the penalized elemental density, and $\mathbf{K}_e$ the elemental stiffness matrix of eth element.

Computing Compliance:

Once the nodal displacements are known, the structural compliance are computed as given in equation (2.4),

$$C(\boldsymbol{\rho}, \boldsymbol{u}) = \sum_{e=1}^{N_e} \rho_e^p \boldsymbol{u}_e^T \mathbf{K}_e \boldsymbol{u}_e, \tag{2.4}$$

Sensitivity Analysis:

In order to obtain the optimal solution it is essential to correctly estimate the structural response when changes in the design variables are introduced, hence the sensitivity analysis is performed. The sensitivity of the compliance with respect to the design variables is then computed [50] as given in equation (2.5).

$$\frac{\partial C(\boldsymbol{\rho})}{\partial \rho_e} = -\, p \rho_e^{p-1} \boldsymbol{u}_e^T \mathbf{K}_e \boldsymbol{u}_e. \tag{2.5}$$

TH-3072_146103004

Mesh-Independency Filter:

Topology optimization often suffers from numerical instabilities such as 'checkerboard' issue which refers to the existence of alternating solid and void elements in a region, and mesh-dependency issue which means different solutions are obtained for different mesh sizes. To alleviate these issues a filter is applied over the sensitivities [41]. The sensitivity (or mesh-independency) filter modifies the design sensitivity of an element based on the weighted average of the elemental sensitivities in a defined neighborhood as given in equation (2.6).

$$\widehat{\frac{\partial C(\boldsymbol{\rho})}{\partial \rho_e}} = \frac{\sum_{i\in\mathcal{N}} \left(\frac{\partial C(\boldsymbol{\rho})}{\partial \rho_i}\right) \cdot H_{ei} \cdot \rho_i}{\rho_e \ \sum_{i\in\mathcal{N}} H_{ei}}, \tag{2.6}$$

where $\mathcal{N}$ is the set of neighborhood elements for e^{th} element, which is decided by a user-defined filter radius $\mathcal{R}$ as shown in Figure 2.2. H_{ei} is the weighing function defined as

$$H_{ei} = \begin{cases} \mathcal{R} - d_{ei}, & \text{if} \quad ||d_{ei}|| \leq \mathcal{R}, \\ 0, & \text{otherwise}, \end{cases} \tag{2.7}$$

where d_{ei} is the Euclidean distance between the centers of element 'e' and 'i'.

Density Update:

The elemental densities are then updated as per the optimality criteria update scheme [15] given in equation (2.8).

$$\overline{\rho_e} = \begin{cases} max(\rho_{min}, \rho_e - m), & \text{if} \quad \rho_e B_e^{\eta} \leq max(\rho_{min}, \rho_e - m), \\ min(1, \rho_e + m), & \text{if} \quad min(1, \rho_e + m) \leq \rho_e B_e^{\eta}, \\ \rho_e B_e^{\eta}, & \text{otherwise}, \end{cases} \tag{2.8}$$

where m is a positive move limit, and η is the numerical damping coefficient. Values of $m = 0.2$, and $\eta = 0.5$ are suggested by Bendsøe and Sigmund [57]. B_e is computed using equation (2.9).

TH-3072_146103004

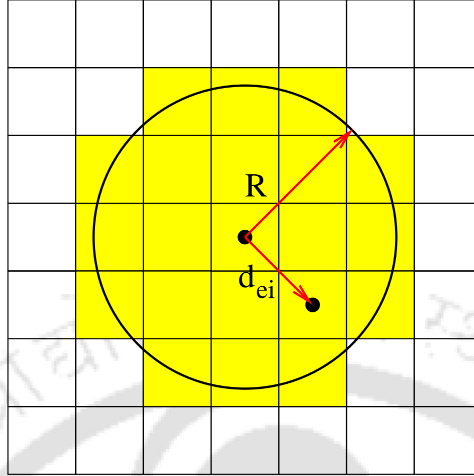


Figure 2.2: Sensitivity filtering for an element over a neighborhood defined by the filter radius '$\mathcal{R}$'.

$$B_e = \frac{-\left(\dfrac{\partial C(\boldsymbol{\rho})}{\partial \rho_e}\right)}{\lambda\left(\dfrac{\partial V(\boldsymbol{\rho})}{\partial \rho_e}\right)}. \tag{2.9}$$

The optimum of the element is found at $B_e = 1$. The value of design variable increases if $B_e > 1$, and decreases if $B_e < 1$. It can be seen from equations (2.8) and (2.9) that the updated values of the design variable ($\overline{\rho_e}$) depend upon the Lagrange multiplier (λ). Finding such value of λ which satisfies the constraint function becomes a one-dimensional optimization problem. In this thesis bisection method is used to compute λ.

Once the elemental densities are updated a convergence check is performed. Depending on the application's needs, various criteria for convergence can be used. In this thesis, two criteria have been used. First is the difference between the objective function values of two subsequent optimization iterations. Second is a limit over the number of optimization iterations. If the termination criterion is met the optimization loop gets terminated. Otherwise, FEA is performed again with the updated $\overline{\rho_e}$ values.

TH-3072_146103004

2.3 Conjugate Gradient Solver and Preconditioning

The conjugate gradient (CG) method [60] is one of the most widely used iterative algorithms for solving large-scale linear systems of equations. The method is particularly suitable for symmetric and positive definite matrices, which are often encountered in many engineering applications. Consider the following system of linear equations,

$$\mathbf{Ku} = \mathbf{f}, \tag{2.10}$$

where $\mathbf{K}$ is an $n \times n$ symmetric positive definite matrix. This system of linear equations can also be expressed as minimization problem of the following convex quadratic function,

$$min\ \phi(\mathbf{u}) = \frac{1}{2}\mathbf{u}^T\mathbf{Ku} - \mathbf{f}^T\mathbf{u}. \tag{2.11}$$

Using this equivalence, the process of solving the system of linear equations in equation (2.10) is same as minimizing the convex quadratic function equation in equation (2.11). The gradient of $\phi(\mathbf{u})$ also equals the residual of the linear system as given in the following equation,

$$\nabla\phi(\mathbf{u}) = \mathbf{f} - \mathbf{Ku} = \mathbf{r}(\mathbf{u}). \tag{2.12}$$

At the start of algorithm, an initial residual is computed using a guess solution '$\mathbf{u}_0$' which gives,

$$\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{f} - \mathbf{Ku}_0, \tag{2.13}$$

where $\mathbf{p}_0$ is the initial search direction. The CG method finds the solution by minimizing the residual error over a Krylov subspace [61]. The solution is found iteratively, by conjugating the current search direction with the previous one at each iteration 'k' [62]. The solution vector is updated at each iteration using the following equation,

$$\mathbf{u}_k = \mathbf{u}_{k-1} + \alpha_k\mathbf{p}_k, \tag{2.14}$$

where $\mathbf{p}_k$ is the set of conjugate directions, and α_k is the step size along the search direction at iteration 'k'. The conjugate search direction is obtained by imposing or-

TH-3072_146103004

thogonality with respect to the previous search directions. This ensures that the search directions are conjugate with respect to the matrix, which improves the convergence rate of the method. The CG method terminates when the residual error reaches below a user defined tolerance or when a maximum number of iterations is reached. The detailed explanation of the method can be found in Hestenes and Stiefel [60], and Nocedal and Wright [62].

Iterative methods such as CG are known to be sensitive to rounding off errors which can generate wrong outputs. In such cases the problem is termed to be 'ill-conditioned', meaning a small perturbation in the inputs of the problem can lead to a huge change in it's output [63]. Additionally, if the problem is ill-conditioned it can deteriorate the convergence speed of the CG method as well [64]. The conditioning of a matrix '**K**' is measured by its 'condition number' which is calculated as,

$$cond\ (\mathbf{K}) = \frac{\|\mathbf{K}\|}{\|\mathbf{K}^{-1}\|}. \tag{2.15}$$

A large value of condition number indicates that the matrix is ill-conditioned. In practice, we use 'preconditioning' to improve the convergence of the solution [65]. Preconditioning refers to transforming the system given in equation (2.10) to a system with more favourable properties for the iterative solver [66], as given in equation (2.16).

$$\mathbf{M}^{-1}\mathbf{K}\mathbf{u} = \mathbf{M}^{-1}\mathbf{f}, \tag{2.16}$$

where $\mathbf{M}$ is a nonsingular matrix of size $n \times n$ known as 'preconditioner'. The linear system given equation (2.16) has the same solution as system in equation (2.10) but is easier and faster to solve. Now, the convergence of the solution becomes dependent on the choice of the preconditioner. The preconditioner should be chosen in such a way that, the preconditioner should be computationally inexpensive to compute, and the preconditioned system should be easier to solve compared to the original system [65].

There are several types of preconditioners available in the literature for iterative solvers; some of the most commonly used preconditioners are discussed below.

- **Jacobi Preconditioner:** also known as diagonal preconditioner [67] is the simplest preconditioner, in which the diagonal of the matrix is used to construct a diagonal preconditioner as,

TH-3072_146103004

$$\mathbf{M} = diag\ (\mathbf{K}). \tag{2.17}$$

The advantage of Jacobi preconditioner is its simplicity and low computational cost.

- **Incomplete LU (ILU) Preconditioner:** ILU preconditioner approximates the factorization of the matrix '**K**' into a lower triangular matrix '**L**' and an upper triangular matrix '**U**'. The preconditioner is then computed as given in equation (2.18).

$$\mathbf{M} = \mathbf{LDU}, \tag{2.18}$$

where **D** is the diagonal matrix. The advantage of the ILU preconditioner is its effectiveness for matrices with high condition numbers and extreme diagonal dominance. However, it may be computationally expensive for large matrices [65].

- **Multigrid Preconditioner:** constructs the multigrid hierarchy of coarser and coarser approximations such that a problem of manageable size is obtained, and then solve the coarse problem exactly using a direct solver. The solution of the coarse problem is then used to improve the solution of the finer problem, through interpolation [68].

  Multigrid preconditioner can be further categorized as geometric multigrid (GMG) and algebraic multigrid (AMG) preconditioners. The main difference between them is in the way they construct the multigrid hierarchy. In GMG, the hierarchy is constructed by successively coarsening the grid geometry to reduce the problem size, while maintaining important features of the solution. This requires knowledge of the geometry of the problem, and hence GMG is more suited to problems with a structured mesh [68]. AMG, on the other hand constructs the hierarchy using only the matrix of the linear system, and hence can be used for problems with unstructured meshes and complex geometry [69]. However, the construction of the multigrid hierarchy and the determination of the interpolation operators can be computationally challenging.

TH-3072_146103004

- **Domain Decomposition (DD) Preconditioner:** decomposes the domain of the problem into subdomains, each of which can be solved independently [70]. The main advantage of DD preconditioners is their ability to handle problems with irregular geometries efficiently. DD preconditioners can be used with any iterative solver, such as the conjugate gradient (CG) method. However, the convergence rate of the iterative solver may depend on the size of the subdomains, the number of subdomains, and the overlap between them [71].

The CG method is known for its fast convergence rate for symmetric and positive definite matrices, which makes it ideally suited for problems in FEA [72]. In addition, it is memory-efficient for large-scale problems. The CG method is easily parallelizable, making it scalable for the solution of very large linear systems on modern high-performance computing platforms [73].

Considering these advantages, CG method with diagonal preconditioning is chosen for solving the system of linear equations in this thesis. Diagonal preconditioner is chosen because it is computationally efficient and easy to implement, as it only requires the diagonal elements of the matrix to be computed and stored. This makes it particularly attractive for problems with a large number of unknowns, as the computational cost of the preconditioner is proportional to the size of the matrix [67].

2.4 GPU Computing

GPUs were first developed for graphics processing applications. Since NVIDIA's introduction of general purpose graphics processing units (GPGPUs) in 2007, they have been utilized in a variety of scientific computing applications. GPUs are usually employed in heterogeneous computing environments, in which parts of an application are executed on CPU and compute-intensive, data-parallel parts are executed on GPU. CPU retains primary control over the application and is responsible for copying the necessary data to GPU and invoking GPU kernels.

As shown in Figure 2.3, GPU's architecture is vastly different from that of CPU. In general, CPU is optimized for sequential code performance. It is equipped with a high-capacity, sophisticated control unit that is connected to a small number of arithmetic logic units (ALUs). CPU is equipped with a large cache memory to reduce the access

TH-3072_146103004

latency of data and instructions. Whereas, the bulk of a GPU's chip area is dedicated to a large number of parallel computing threads. Individual GPU processors are called called streaming processors (SP) and each has its own register memory. A collection of SPs constitutes a streaming multiprocessor (SM). A control unit and cache are shared among SPs within SM.

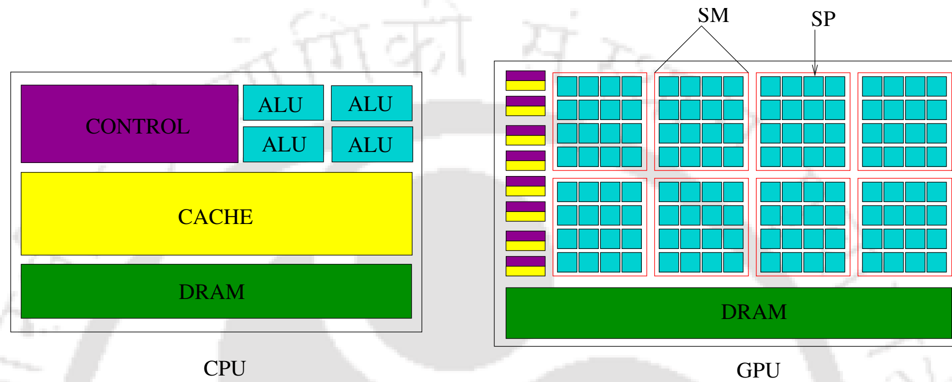


Figure 2.3: Simplified model of CPU and CPU architectures.

The parallel program that runs on GPU is defined by a function called 'kernel'. The kernel is invoked by the host (CPU) after specifying the number of threads. These threads are grouped into thread 'blocks.' Threads within a block can communicate and share information. However, threads within one block cannot communicate directly with threads within other block. The total number of threads specified during kernel launch is divided into these thread blocks, which are referred to as a 'grid.' GPU architecture clubs the individual threads into a group of 32 called as a 'warp'. At a time a single instruction is executed by the all threads of a warp. Programmer can control only the size of the grid and blocks, while GPU handles the formation of warps.

A GPU device is comprised of various memory types, each with different characteristics and functions. Any application's performance is highly dependent on the optimal utilization of these memories. Table 2.1 lists the memory types, their location on the device, and their scope of access.

The register and local memories are private to each thread. The shared memory is exclusive to a thread block and generally used by the threads of a block to communicate and share data. Access to on-chip memories are faster. Global memory is the largest in the size and accessible to all the thread throughout the application run time. However,

TH-3072_146103004

Table 2.1: Different types of memories available on a GPU device

Memory	Location	Accessed by
Register	On-chip	Thread
Local	On-chip	Thread
Shared	On-chip	Block
Global	Off-chip	Grid
Constant	Off-chip	Grid

transactions between threads and global memory are 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.

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. Since CUDA allows a maximum control over the available GPU resource, the responsibility of achieving a good performance and accurate results lies with the programmers. To accomplish this, the following considerations must be taken into account.

Race Condition:

A race condition occurs when two or more threads with access to the same memory location attempt to modify the same data at the same time. Multiple threads can therefore have different values for the same memory address at the same time. This occurs because the programmer cannot anticipate the order in which threads will access the data. Consequently, race conditions might lead to inconsistent and inaccurate results and should be avoided.

Thread Divergence:

The GPU architecture is designed in such a way that all the threads in a warp execute the same instruction. However, if the threads in a warp execute different instructions due to instruction level branching, this is known as 'thread divergence.' It wastes computing resources and must be avoided.

TH-3072_146103004

Memory Coalescing:

Memory coalescing is a technique that enables the GPU's global memory to be utilized optimally. It is achieved when threads within a block access consecutive global memory addresses. In this scenario, GPU bundles all memory accesses into a single memory transaction. For optimal performance, global memory accesses must be coalesced.

2.5 Literature Review

In this section, the relevant literature related to the GPU acceleration of density-based topology optimization is discussed.

2.5.1 Parallel Computing for SIMP-based Topology Optimization

SIMP is one of the most well-known and widely employed topology optimization methods in the literature. In order to reduce the computation time of large-scale SIMP-based topology optimization, parallel computing has been extensively used in the literature. The conventional parallel computing architectures include shared memory architecture and distributed memory architecture. A shared memory architecture provides all processors access to a centralized memory system, and each processor's memory utilization is dependent on its current requirement. OpenMP is the programming interface used with this architecture. In a distributed memory architecture, each processor has access to its own memory system. A communication channel is required for processors to share data. The Message Passing Interface (MPI) is the most often utilized communication interface.

Borrvall and Petersson [42] accelerated the topology optimization of large-scale 3D continuum structures on a CRAY T3E system utilizing MPI and the domain decomposition method (DDM). Kim et al. [74] accelerated the topology optimization of eigenvalue problems using DDM over a cluster system. Vemaganti and Lawrence [43] used a decomposition technique based on Hilbert curves for SIMP-based topology optimization of 2D structures on a cluster. In each of these studies, the linear system of equations was solved using a parallel version of the preconditioned conjugate gradient (PCG) algorithm

TH-3072_146103004

during FEA. Borrvall and Petersson [42] solved the optimization problem using sequential convex programming, whereas Kim et al. [74] and Vemaganti and Lawrence [43] used the optimality criteria (OC) method. Aage and Lazarov [75] developed a parallel framework for topology optimization using the method of moving asymptotes (MMA).

Mahdavi et al. [44] used a shared memory system to accelerate the OC method-based topology optimization of 2D continuum structures. The system of linear equation was solved using a PCG algorithm on a shared memory system following a master-slave programming paradigm. París et al. [76] used OpenMP to develop a parallel framework for the stress-constrained topology optimization of 2D structures on a multi-core system.

Shared and distributed memory architectures, which are commonly referred to as traditional parallel architectures, have been applied successfully in numerous studies to accelerate the various computational steps of topology optimization. However, traditional architectures have several practical limitations.

Traditional parallel architectures have limited memory bandwidth, which may limit the performance of applications that require frequent data transfers between the compute core and memory [77]. They have limited scalability, and scaling to larger systems can be costly and time-consuming. In addition, traditional architectures have limited floating-point performance, which hinders their ability to handle applications that involve intensive floating-point arithmetic operations [77]. The set-up and maintenance costs of conventional architectures are also quite high. Finally, traditional parallel architectures, particularly distributed memory architectures, can consume a lot of power, resulting in high energy costs and negative environmental impacts [78].

GPUs have emerged as a remedy to these limitations of traditional architectures in recent years. GPUs are highly parallel, many-core processor architecture. They have a high memory bandwidth, allowing them to efficiently process large amounts of data. GPU cores are also specifically designed to handle a large number of floating-point calculations, making them a suitable choice for arithmetic-intense operations [79]. GPUs are designed to be highly scalable and easy to integrate into existing systems in order to provide additional processing capacity [80]. GPUs are compact, low-maintenance, and energy-efficient architectures with a high level of parallelism, making them an ideal choice for data-parallel and arithmetic-intensive applications like topology optimization [81].

TH-3072_146103004

2.5.2 GPU Acceleration of SIMP-based Topology Optimization

In recent years, GPUs have been favored over conventional parallel computing architectures due to their numerous advantages. The GPU acceleration strategy for topology optimization depends heavily on the type of mesh used to discretize the design domain. Therefore, the relevant literature is categorized according to the type of mesh used in the study.

2.5.2.1 Using Structured Mesh

The earliest use of GPU to accelerate the topology optimization was by Wadbro and Berggren [49]. Their problem statement was finding the optimal distribution of two materials with different heat conduction properties in order to obtain an even temperature field. A 2D mesh of identical finite elements was used to discretize the domain. In this study, GPUs were used to accelerate the FEA part of topology optimization, and matrix-free PCG was used as the FEA solver. The SpMV operations of PCG solver were performed on-the-fly following the element-by-element (*ebe*) strategy. cuBLAS library [82] was used for linear algebraic operations of vectors. For data read operations, one-thread per element was used, and for write operations, one-thread per node was used. The elemental and nodal data were stored in the GPU's shared memory to enable faster memory access. For the given problem, a maximum speedup of 20× over a single-threaded CPU implementation was observed.

Later, Schmidt and Schulz [50] accelerated topology optimization of 3D linear elastic structures by employing the node-by-node (*nbn*) SpMV strategy for the matrix-free conjugate gradient (CG) solver on GPU. Since node-wise computations are independent of each other, the issue of race condition does not arise. The design domain was meshed using 3D structured mesh of identical elements, and a single element stiffness matrix ($\mathbf{K}_e$) was used for the entire FE mesh which was stored in the GPU's constant memory. The authors used the shared memory of the GPU to store the nodal data of three successive 2D slices of nodes for each thread block. By successively moving in the third direction, SpMV for the entire mesh can be efficiently performed. Using an *nbn* thread allocation-based matrix-free CG solver, the authors achieved a speedup of 1.7× over a shared memory system with 48 CPU cores for a 3D cantilever beam example. The strategy of using GPU's constant memory to store a single $\mathbf{K}_e$ for identical elements was also

TH-3072_146103004

used by Martínez-Frutos and Herrero-Pérez [46] for accelerating topology optimization of large-scale linear elastic structures on multi-GPU systems. Multiple GPUs were used to accelerate the FEA, sensitivity analysis, and mesh filtering of topology optimization. This study used a multi-granular strategy for computations on the GPU. The GPU-based matrix-free PCG solver was developed using the DoF-by-DoF (*dbd*) thread allocation strategy, and sensitivity analysis and mesh filtering were performed using the *ebe* thread allocation strategy. The atomics operation of the CUDA toolkit was used to avoid the race condition issue during write operations. The FEA solver showed maximum speedups of 8× and 14× for the double hook design problem and the heat sink design problem, respectively, for a mesh having 1.7 million DoFs.

Another multilevel granularity-based acceleration of complete topology optimization was presented by Martínez-Frutos et al. [45]. The acceleration of FEA was achieved by implementing a *dbd* thread allocation-based matrix-free PCG solver on the GPU. A single $\mathbf{K}_e$ was used for the entire mesh in this study as well. Sensitivity analysis, mesh filtering, and density update were computed using the *ebe* strategy. GPU's shared memory was used to store the elemental data of a thread block. For vector arithmetic operations on GPU the Thrust library [83] of CUDA toolkit was used. Atomics were used to avoid the race condition issue. The GPU-based PCG solver showed speedups of 19.9×, 19.4×, and 22.5× on the tied-arch bridge design, the 3D gripper design, and the heat sink design problems, respectively.

2.5.2.2 Using Unstructured Mesh

Zegard and Paulino [51] were the first to develop a GPU-based topology optimization framework considering 2D unstructured meshes. A parallel Cholesky decomposition-based direct solver was used for solving the linear system of equations. The direct solver requires the assembly of a global stiffness matrix. The assembly was performed in parallel using the *ebe* thread allocation strategy. In this study, the symmetry property of elemental stiffness matrices was exploited to reduce the memory space requirement. Only the lower triangular entries of each elemental stiffness matrix were stored. Furthermore, only the lower triangular half of the global stiffness matrix was used by the solver. The mesh filtering and sensitivity analysis were performed by following the *ebe* thread allocation strategy. A greedy-graph coloring algorithm was implemented to avoid race

TH-3072_146103004

conditions. 'Coloring' is a popular technique that divides the finite elements into colors, such that the elements belonging to one color do not share a boundary or a node. The computations of all the elements belonging to one color can be performed in parallel without facing the race condition issue. GPU-based direct solver showed speedups in the range of 15× to 20× over the serial version for three linear elasticity examples.

Later, Duarte et al. [52] presented a polygonal mesh-based topology optimization framework named 'PolyTop++' for 2D and 3D structures on GPU. The FEA step was accelerated by using both a parallel direct solver (UMFPACK) and a parallel matrix-free iterative solver (PCG). The matrix-free PCG solver uses an *ebe* thread allocation strategy, and a greedy-graph coloring algorithm to avoid race conditions. In polygonal meshes, elements have different numbers of vertices. The framework uses two successive subdivisions of elements, they are first divided by the number of vertices, and then by color. A single $\mathbf{K}_e$ was computed at the start for similar elements, and only the lower symmetric half of $\mathbf{K}_e$ was stored. For a problem size of 10 million polygonal elements, the GPU-based matrix-free PCG solver showed 13× speedup over its CPU counterpart and outperformed the direct solver by 1.5× for a mesh with 1 million elements. The authors also noted that for unstructured meshes, even though $\mathbf{K}_e$ of all finite elements need to be computed and explicitly stored, their memory requirement is still less than that of an assembled global stiffness matrix.

Recently, Herrero-Pérez and Castejón [84] presented a multi-GPU based implementation for acceleration of large-scale density-based topology optimization using both structured and unstructured meshes. The finite element mesh was partitioned into non-overlapping sub-domains, which were then allocated to a distributed memory system consisting of 8 GPU cards. A standard message passing interface (MPI) was used to exchange elemental data among the GPUs. A distributed version of the PCG solver preconditioned with the aggregation-based AMG method was used to solve the system of linear equations on GPU. The parallel computation of FEA, objective function, sensitivity analysis, and the design variable update were performed at the elemental level (*ebe*). The largest finite element mesh used in this study consisted of $106,348,544$ tetrahedral elements. A speedup of 6.1× was observed with respect to a distributed system with 32 CPU cores. A maximum speedup of 7.8× was reported for a finite element model with $15,630,336$ hexahedral elements using the same computational setup.

TH-3072_146103004

2.5.3 GPU Acceleration of Other Topology Optimization Methods

A GPU-based topology optimization framework based on the level-set method was developed by Challis et al. [85] to solve the problem of the design of isotropic materials with maximized bulk modulus. The results show that the GPU implementation performs $9\times$ to $13\times$ faster for mesh sizes containing $64,000$ and $4,096,000$ elements, respectively, when compared to the serial implementation. Liu et al. [86] presented a fully parallel implementation of the parameterized level-set method for large-scale structural topology optimization. Using open source libraries, the entire topology optimization, including mesh generation, FEA, sensitivity analysis, parameterization of the level-set function, and updating of the level set function, were carried out in parallel for uniform and non-uniform structured meshes. Recently, Li et al. [87] proposed a two-grid method for level-set based topology optimization of 2D problems on GPU. The proposed method was reported to be $20\times$ faster than the benchmark method.

The GPU was used to accelerate the topological sensitivity method-based topology optimization by Suresh [47]. In this study, instead of a single topology, a pareto set of multiple optimal topologies with varying volume fractions is generated. The proposed algorithm was reported to be $10\times$ to $100\times$ faster than the previously published literature [42, 57].

Multiple research works have used GPUs to accelerate evolutionary structural optimization (ESO). Martínez-Frutos and Herrero-Pérez [88] proposed a multi-granular GPU implementation of the evolutionary topology optimization method driven by stress isosurfaces. A matrix-free PCG solver using the fixed-grid FEA method was proposed. Speedups in the range of $7\times$ to $19\times$ were achieved for GPU implementations over CPU implementations. Ram and Sharma [89] implemented evolutionary algorithm on GPU to generate geometrically feasible structures by adopting triangular representation for 2D continuum structures. The proposed strategy was $5\times$ faster than the CPU implementation. Bi-directional ESO (BESO) was implemented on GPU by Munk et al. [90] to accelerate the lattice Boltzmann method for multi-physics topology optimization. Speedups between $18\times$ and $67\times$ were achieved compared to the serial implementation (CPU).

TH-3072_146103004

2.5.4 Open-Source Parallel Topology Optimization Frameworks

There are a few open-source parallel topology optimization implementations available in the literature. PETSc [91] is likely the most well-known of them all. It is a fully parallel framework based on the MMA method for solving large-scale topology optimization problems over structured meshes. The framework is written in C++, and its parallelism is easily scalable across CPUs with multiple cores. The framework is flexible and can be applied to a variety of problems, including compliance minimization, material design, etc. Recently, Smit et al. [92] have extended the functionality of PETSc to include passive domains and local volume constraints, along with a Python wrapper to expand the user base.

Topo-fenics [93] is another such framework which is a 55-line Python code for 2D and 3D topology optimization based on open-source finite element software FEniCS [94]. This framework follows SIMP method and can be run on HPC workstations and clusters. With some modifications, topo-fenics is capable of solving complex structural problems. PolyTop++ [52] is another parallel framework for parallel 2D and 3D topology optimization which is developed using C++ and CUDA. It has been discussed in detail in previous section.

A GPU-based open-source topology optimization framework is developed by Schmidt and Schulz [50] using C and CUDA. The framework can be used for 3D compliance minimization problem over structured meshes. Recently, an OpenMP-GPU and OpenMP-CPU based parallel framework for large-scale topology optimization problems was presented by Träff et al. [95]. The fraemework can also be extended to solve problems of nonlinear elasticity.

2.5.5 Closure

The literature survey indicates that parallel computing has been used extensively to reduce the total computation time of SIMP-based structural topology optimization. Different parallel architectures have been employed to accelerate the major computational steps of topology optimization and the entire optimization framework. In the recent past, GPUs have been the preferred choice as an HPC setup among researchers. Mainly because they offer high memory bandwidth, and high level of parallelism at the reasonable cost. Most of the GPU-based studies in the literature have focused on developing

TH-3072_146103004

strategies for accelerating various computational steps of topology optimization. While, hardly any of them discuss the practical challenges of implementing these strategies on GPUs. Secondly, there is a lack of focus on how to optimize the existing algorithms to fully utilize the available computational resources. In the majority of GPU-based studies, structured meshes are used due to their computational simplicity on GPU devices. However, as far as the simulation accuracy is concerned, unstructured meshes are more suited to handle the complex geometries with arbitrary shapes. In order to optimally use the available GPU resources, unstructured meshes necessitate efficient load distribution and data handling strategies. Regardless of the type of mesh used, the primary goal of the vast majority of studies is to efficiently execute the FEA solver on GPU, as it is the primary computational bottleneck of topology optimization. Some studies have used direct solvers, although they require a huge memory space. Assembly-based iterative solvers encounter the similar problem. PCG solvers with preconditioning are the most popular in the literature for the same reason that matrix-free (assembly-free) iterative solvers are better suited for GPU devices. For matrix-free iterative solvers employing large-scale 3D unstructured meshes, the literature still lacks efficient thread allocation and data storage strategies that can maximally utilize the available GPU resources.

TH-3072_146103004

Chapter 3

GPU-based Matrix-Free FEA Solver for Structural Topology Optimization using Unstructured Mesh

Since FEA is the main computational bottleneck in topology optimization, we execute it on GPU to speed up topology optimization. The first objective of this thesis is to develop an efficient GPU-based FEA solver tailored to efficiently handle the unstructured meshes. In the literature, there are primarily two types of solvers: direct solvers and iterative solvers. The exact solution to the system of linear equations is obtained by direct solvers. In this method, the solution is obtained by solving the system of linear equations directly, either by elimination or decomposition. The assembly of elemental stiffness matrices into a global stiffness matrix is required for direct solvers. The process of assembly for large-scale structures is computationally expensive [36, 53]. In addition, they require a large memory space for storing the global stiffness matrix, whose size can be enormous, as well as additional memory for factorization. Even though they are more robust, direct solvers are infeasible for large-scale structures for these reasons. Iterative

TH-3072_146103004

solvers start with an initial guess solution and 'iteratively' refine it until it converges within a specified tolerance. One of the primary advantages of these methods is that the convergence tolerance can be adjusted based on the application requirements. For computation, iterative solvers require less memory. However, storage memory is still necessary for assembly-based iterative solvers. In the literature, a novel category of iterative solvers known as matrix-free or assembly-free iterative solvers exists [96]. The matrix-free iterative solvers perform computations locally at the elemental or nodal level 'on-the-fly,' therefore the assembly of a global stiffness matrix is avoided. Matrix-free iterative solvers are the ideal choice for GPU-based acceleration of topology optimization due to their inherent parallelism [55]. In this thesis, a GPU-based matrix-free PCG solver is developed to handle unstructured meshes efficiently. The proposed matrix-free PCG solver is discussed in the following section.

3.1 Matrix-Free PCG Solver on GPU

The computational steps of matrix-free PCG solver are discussed in Algorithm 1. The input data required by the solver are the global stiffness matrix '$\mathbf{K}$' and force vector '$\boldsymbol{f}$', initial guess solution '$\boldsymbol{u_0}$', elemental density vector '$\boldsymbol{\rho}$', and the preconditioner '$\mathbf{M}^{-1}$'. Since matrix-free PCG solver does not assemble elemental stiffness matrices, the required calculations are done 'on-the-fly' at the elemental level. A diagonal preconditioner is also used with the PCG solver. Two termination conditions are used that are maximum number of iterations '$iter$' and tolerance value for the residual 'ε'. When both the conditions are satisfied, the iterative solver gets terminated. The output of the solver is stored in the nodal displacement vector '$\boldsymbol{u}$'.

Algorithm 1 starts by computing the residual '$\boldsymbol{r}$' at line 2. This step requires SpMV operation between $\mathbf{K}$ and $\boldsymbol{u_0}$. In line 3, the vector product of residual and the preconditioner '$\mathbf{M}^{-1}$' is computed, and the result is copied to vector '$\boldsymbol{z}$'. Line 4 copies $\boldsymbol{z}$ into $\boldsymbol{d}$. The variable 'δ^{new}' in line 5 is computed by taking the inner product of $\boldsymbol{r}^T\boldsymbol{z}$. Line 6 loops over the residual tolerance value 'ε' and the maximum number of PCG iterations '$iter$'. Another SpMV operation is performed in line 7 to compute '$\boldsymbol{q}$', which is then used in line 8 to compute the variable 'α'. The $\boldsymbol{u}$ and $\boldsymbol{r}$ vectors are updated in lines 9 and 10, respectively. The vector product of $\boldsymbol{r}$ and $\mathbf{M}^{-1}$ is then performed in line 11. The value

TH-3072_146103004

of δ^{new} is copied to δ^{old} in line 12, and δ^{new} is recomputed in line 13. Line 14 evaluates variable 'β', which is subsequently used in line 15 to update the vector '$\boldsymbol{d}$'. The iteration counter is incremented in line 16, and the loop continues until the convergence criteria in line 6 are not met.

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

```
   Data: K, f, u_0, ρ, M^-1, iter, ε
   Output: u
 1 i ← 0
 2 r ← f − K u_0                                // SpMV kernel
 3 z ← M^-1 r                                   // CUDA Thrust
 4 d ← z                                        // CUDA Thrust
 5 δ^new ← r^T z                                // CUDA Thrust
 6 while i < iter and δ^new > ε do
 7     q ← K d                                  // SpMV kernel
 8     α ← δ^new / d^T q                        // CUDA Thrust
 9     u ← u + α d                              // CUDA Thrust
10     r ← r − α q                              // CUDA Thrust
11     z ← M^-1 r                               // CUDA Thrust
12     δ^old ← δ^new
13     δ^new ← r^T z                            // CUDA Thrust
14     β ← δ^new / δ^old
15     d ← z + β d                              // CUDA Thrust
16     i ← i + 1
17 end
```

The PCG solver detailed in Algorithm 1 shows the computational steps of a conventional PCG solver from the literature. This solver must be modified, however, so that it can be applied to unstructured meshes and used to solve the topology optimization problems used in this thesis. The proposed PCG solver takes a generalized approach and treats every mesh as an unstructured one. Hence, the stiffness contribution of each individual element is taken into account during FEA. Furthermore, the matrix-vector multiplications in lines 2 and 7 of the standard PCG algorithm require the assembly of a global stiffness matrix and force vectors. For large-scale problems, storing and moving data at such a scale becomes impractical. Therefore, the PCG solver is modified to perform matrix-vector multiplications at the local (elementary or nodal) level. In addition,

TH-3072_146103004

the efficient distribution of the computational load and optimization of the CPU-GPU data transfer are addressed by developing customized SpMV strategies and storage patterns, which are discussed in the subsequent sections. These customizations make the PCG solver matrix-free, enabling it to function efficiently with unstructured mesh.

The matrix-free PCG FEA solver shown in Algorithm 1 consists of two types of operations: sparse matrix-vector multiplications (SpMV) and vector arithmetic operations. Initial experiments show that SpMV operations on GPU are more complex and can consume 80% – 99% of the solver's execution time, as compared to other vector arithmetic operations. In this thesis vector arithmetic operations are performed on GPU by using Thrust library of CUDA toolkit. For SpMV operations on GPU, the following two kinds of strategies can be found in literature.

Element-by-Element (*ebe*) SpMV Strategy:

In the standard *ebe*-based SpMV strategy, one compute thread of GPU is assigned to each element of the FE mesh [52]. Figure 3.1 demonstrates the *ebe*-based multiplication for a 2D mesh element 'e'. $\mathbf{K_e}$ is the elemental stiffness matrix, and $\boldsymbol{u}_i$ is the multiplying vector for this element. The thread assigned to 'e' multiplies the $\mathbf{K_e}$ entries of node '1' and writes the product to the output vector. This operation is repeated for nodes 2, 3, and 4 of 'e'. The matrix-vector multiplication is performed for the entire FE mesh by executing the same operations in parallel for each element of the mesh by its respective thread. It can be observed from Figure 3.1 that while writing multiplication results, multiple threads may attempt to modify the same output vector location, resulting in a race condition issue. In the majority of cases, the coloring method or atomic operations of CUDA is used to address this issue.

Node-by-Node (*nbn*) SpMV Strategy:

In *nbn* strategy, a single GPU thread computes the state for all degree-of-freedoms (DoF) of a node [50] as shown in Figure 3.2. The node's multiplying vector is $\boldsymbol{u}_e$, and the stiffness matrices of the elements attached to it are $\mathbf{K_i}$. The thread must have access to the elemental data of adjacent elements. Since each node's computation is performed independently, no race condition is observed [55].

In order to identify the best GPU-based SpMV strategy for large-scale unstructured 3D meshes, both of these strategies are tested in this thesis. The proposed '***ebe***' and

TH-3072_146103004

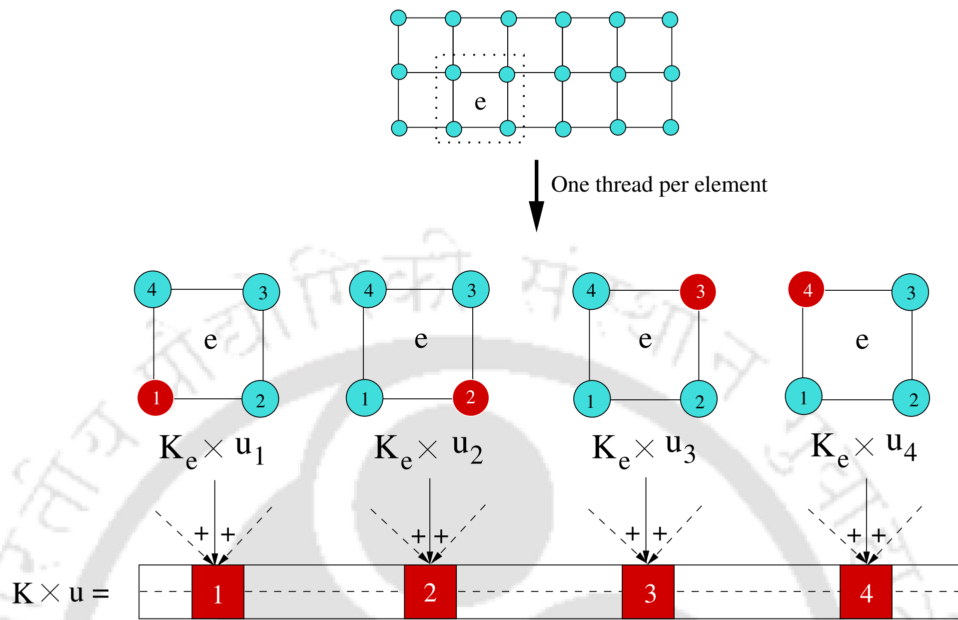


Figure 3.1: Element-by-element SpMV strategy.

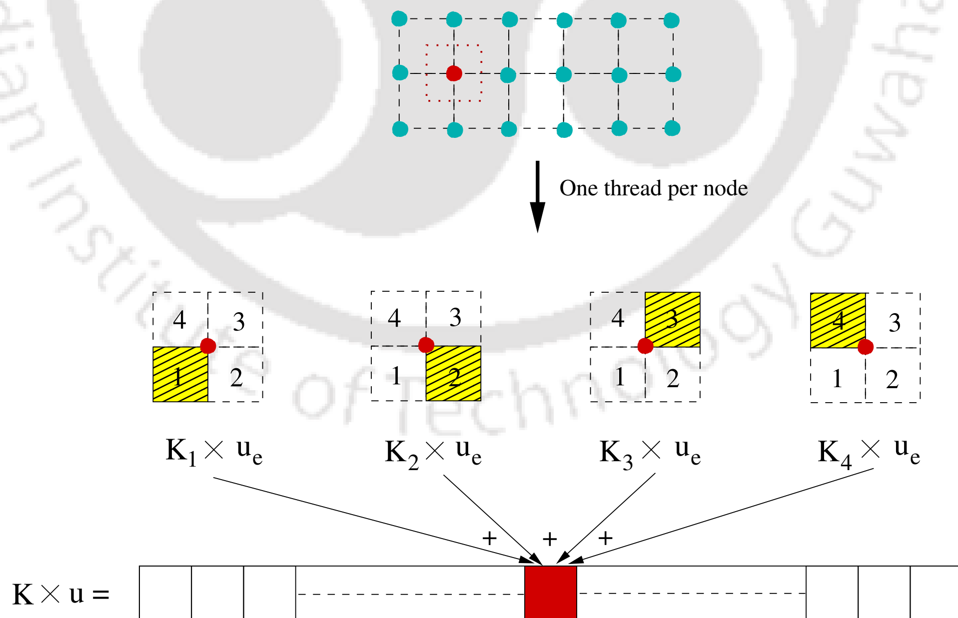


Figure 3.2: Node-by-node SpMV strategy.

TH-3072_146103004

'***nbn***' SpMV strategies are tailored to efficiently handle unstructured meshes to be used with the matrix-free PCG solver. The two proposed SpMV strategies are discussed in the following subsections.

3.1.1 Proposed Element-by-Element SpMV Strategy '*ebe*'

In this thesis the design domain of the structures are meshed using 8-noded hexahedral finite element with 3 DoFs per node, hence the size of $\mathbf{K}_e$ and $\boldsymbol{u}_e$ are 24×24 and 24×1, respectively, as shown in Figure 3.3. All entries of $\mathbf{K}_e$ and $\boldsymbol{u}_e$ are multiplied by a single compute thread.

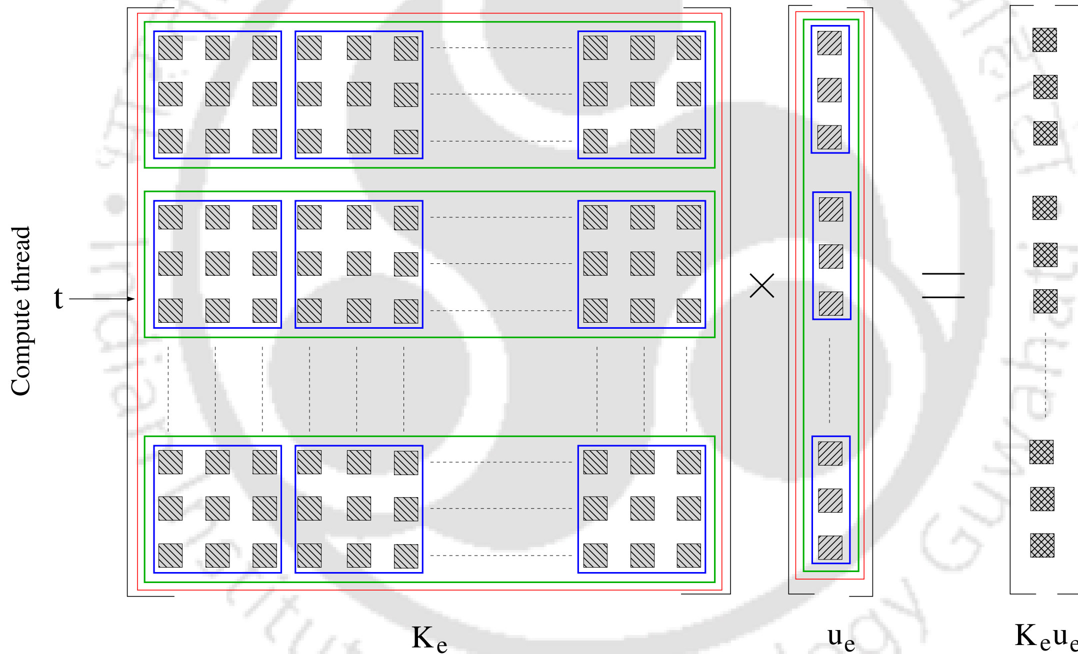


Figure 3.3: Matrix-vector multiplication by the proposed GPU-based element-by-element SpMV strategy '***ebe***'.

The data required by the proposed '***ebe***' strategy are the elemental stiffness matrices ($\mathbf{K}_e$) of all elements, connectivity matrix ($\mathbf{C}$), nodal displacements ($\boldsymbol{u}$), and the elemental density vector ($\boldsymbol{\rho}$). The proposed '***ebe***' strategy is given in Algorithm 2 that starts by assigning a global index to a compute thread that represents an element in the mesh. The thread '*t*' is responsible for reading the required data, performing computations, and

TH-3072_146103004

writing the results for the element it is allocated to. 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 3.3. 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 $\mathbf{K}_e$. Finally, in line 12, the thread performs the multiplication between $\mathbf{K}_e$ 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 3.3. In line 13, the product of multiplication is stored in the output array $\boldsymbol{r}$. To avoid the race condition in line 13 atomics operation of CUDA is used.

Algorithm 2: Proposed GPU-based element-by-element SpMV strategy '***ebe***.'

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

```
1  t = blockIdx.x × blockDim.x + threadIdx.x;
2  if t < Elem then
3      C_ℓ [8];                                        // local memory space
4      C_ℓ ← copy from C;
5      ρ_e = ρ[t]^p;
6      for i ← 0 to 7                                  // nodes of 't'
7       do
8          id_1 ← C_ℓ[i];
9          val = 0.0;
10         for j ← 0 to 7 do
11             id_2 ← C_ℓ[j];
12             val + = ρ_e × {K_e[i][j] × u_e[id_2]};  // matrix-vector product
13         r[id_1] + = val;                            // Atomic add
14 end
```

Standard *ebe*strategy for structured meshes use a single $\mathbf{K}_e$ for the entire FE mesh. In addition, connectivity information can be generated on-the-fly using simple arithmetic

TH-3072_146103004

computations instead of being explicitly stored.

3.1.2 Proposed Node-by-Node SpMV Strategy '***nbn***'

The data requirements of the proposed node-by-node spMV strategy '***nbn***' is largely similar to ***ebe*** strategy. However, the access pattern of threads to connectivity data is different. As shown in Figure 3.2, each thread requires access to the data of the adjacent element, and hence must be aware of the global numbering of the node's neighboring elements. This information is retrievable through a search of the connectivity matrix '**C**'; however, running search operations on GPU is inefficient and wastes computing resources. As a remedy, this thesis proposes a customized nodal connectivity storage format referred to as the 'reverse-connectivity matrix' ($\mathbf{C}_{rev}$), which is explained in detail in the following subsection.

3.1.2.1 Proposed Customized Nodal Connectivity Storage Format '$\mathbf{C}_{rev}$'

The purpose of the 'reverse-connectivity matrix' ($\mathbf{C}_{rev}$) is to make the read operation of connectivity data on GPU more efficient in order to maximally utilize the GPU's computing resources. $\mathbf{C}_{rev}$ provide each compute thread the global numbering of elements attached to the node to which the thread is allocated. It is created by performing a search operation through the connectivity matrix for each node. Although conducting a search operation for each node consumes a lot of time, $\mathbf{C}_{rev}$ is created only once as the part of preprocessing step at the beginning of the optimization. Consider a patch of 2D finite elements with one DoF per node, as depicted in Figure 3.4. Figure 3.5 shows the connectivity matrix (**C**) for this patch and the procedure to generate $\mathbf{C}_{rev}$ from **C** for a particular node 'N_4'. Four elements e_0, e_1, e_2, and e_3 are shared by node N_4, as revealed by a search through the matrix **C**. The first four entries of $\mathbf{C}_{rev}$ for node N_4 contain the global *ids* of these elements, followed by an integer value representing the node's member count. The following four entries represent the respective local numbering of node N_4 in its neighboring elements. The same procedure is followed for all the nodes in FE mesh.

TH-3072_146103004

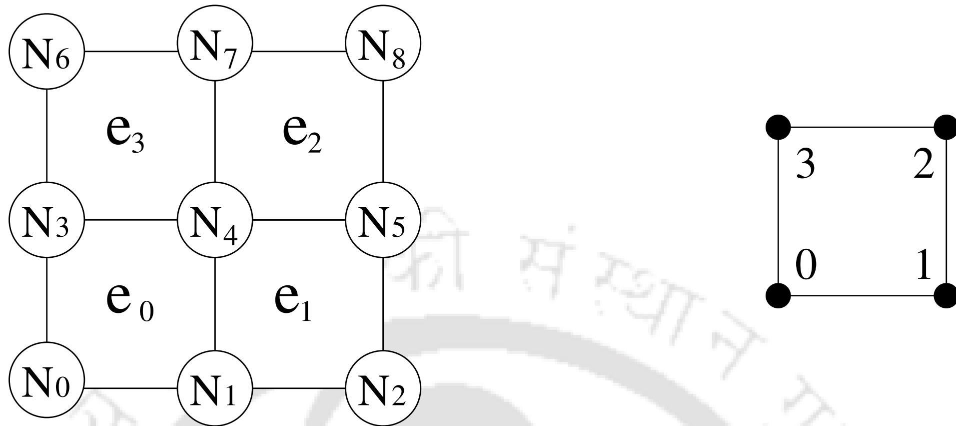


Figure 3.4: A patch of 2D elements with one *DoF* per node, and their internal node numbering scheme.

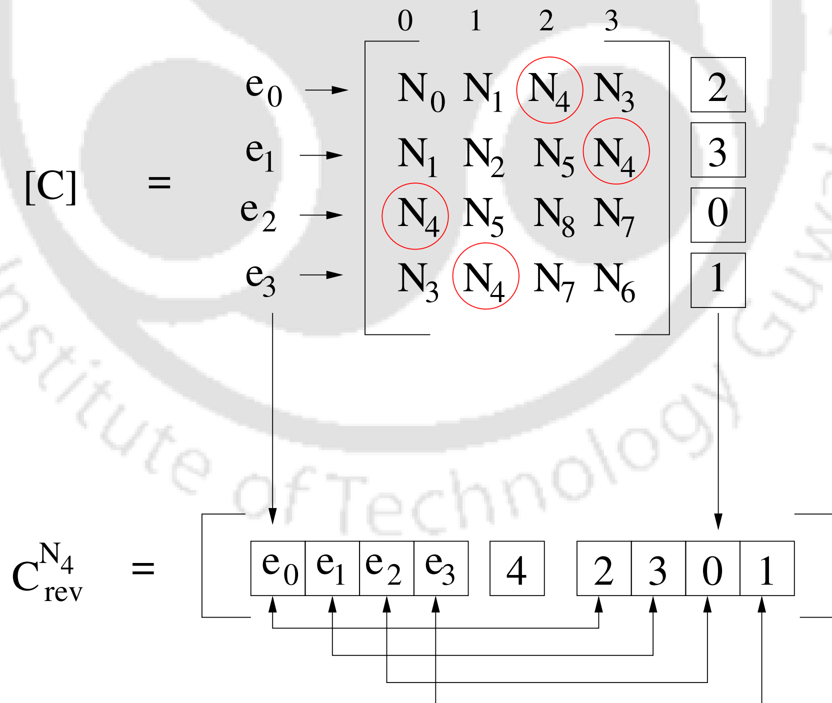


Figure 3.5: Customized nodal connectivity storage format '$\mathbf{C}_{rev}$'.

TH-3072_146103004

3.1.2.2 Algorithm for '*nbn*' Strategy

The proposed ***nbn*** strategy is shown in Algorithm 3 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 $\mathbf{C}_{rev}$ 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 $\mathbf{C}_{rev}$ 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 $\mathbf{C}$ for each node of an element 'e'. The index id_2 represents the global index of node 'j'. Finally, the matrix-vector multiplication between $\mathbf{K}_e$ 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 $\mathbf{C}_{rev}$ and $\mathbf{C}$ for reading the connectivity indices. In addition, each thread must read $\mathbf{K}_e$ 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.

3.2 Topology Optimization using Proposed FEA Solver

The matrix-free PCG solver introduced in Algorithm 1 is then incorporated into a SIMP method-based structural topology optimization framework for large-scale 3D continuum structures discretized with unstructured meshes. Figure 3.6 shows the topology optimization framework, the first step of which is the initialization of allocated memories. The mesh data are then imported into the framework. The proposed framework takes a generalized approach by considering unstructured mesh regardless of domain geometry.

TH-3072_146103004

Algorithm 3: Proposed GPU-based node-by-node SpMV strategy '***nbn***.'

Data: $\mathbf{K}$, $\mathbf{C}$, $\mathbf{C}_{rev}$, $\boldsymbol{u}$, $\boldsymbol{\rho}$, p, $Node$

Output: $\boldsymbol{r}$

```
1  t = blockIdx.x × blockDim.x + threadIdx.x;
2  if t < Node then
3      val = 0.0;
4      ξ_t ← read from C_rev;                              // neighborhood elements
5      foreach e ∈ ξ_t do
6          id_1 ← read from C_rev;
7          ρ_e = ρ[e]^p;
8          for j ← 0 to 7                                  // nodes of 'e'
9          do
10             id_2 ← read from C;
11             val += ρ_e × {K_e[id_1][j] × u_e[id_2]};    // matrix-vector product
12     r[t] = val;
13 end
```

The meshing is performed using the Ansys R16.1 APDL module with 8-noded hexahedral elements for all of the examples in the thesis. During this step, the various program parameters are also defined. The subsequent stage is data preprocessing, which includes the computation of $\mathbf{K}_e$ for all finite elements in the mesh, the preparation of the list of neighbouring elements for the mesh-independency filter, etc. This step includes the construction of $\mathbf{C}_{rev}$ when ***nbn*** strategy is used. Then, the nodal displacement vector ($\boldsymbol{u}$) and density vector ($\boldsymbol{\rho}$) are initialized. FEA is executed on the GPU, whilst the remaining steps are executed on CPU. To perform FEA, all elements' $\boldsymbol{\rho}$ and $\mathbf{K}_e$ must be copied to GPU. The framework allocates GPU memory and copies data to the GPU's global memory. FEA is then performed by following the Algorithm 1. After computing the nodal displacements, the vector $\boldsymbol{u}$ is copied back to CPU. The compliance of the structure is then determined using equation (2.4), followed by the sensitivity analysis given by equation (2.5). Filtering the elemental sensitivities according to equations (2.6)–(2.7) eliminates the checkerbording issue. The elemental densities ($\boldsymbol{\rho}$) are then updated based on the filtered sensitivities using equations (2.8)–(2.9), and convergence is checked. If the convergence criterion is met, the framework writes the findings and vti files. If not, the FEA is re-executed using the updated $\boldsymbol{\rho}$ values. Until the convergence criterion is not met, the loop will continue.

TH-3072_146103004

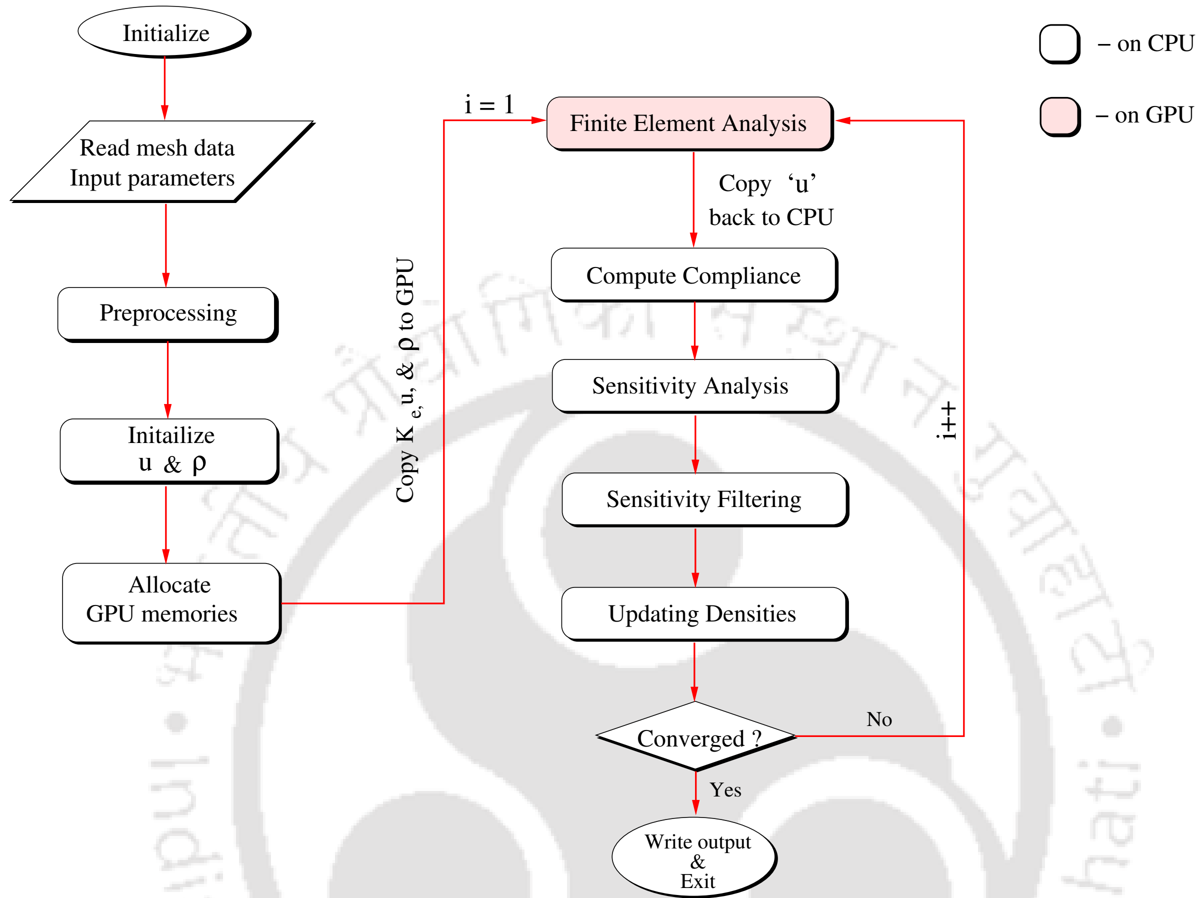


Figure 3.6: Flowchart of SIMP method-based topology optimization framework using the proposed matrix-free PCG solver.

3.3 Benchmark Examples

Four benchmark examples of structural topology optimization are used to test the performance of the proposed GPU-based matrix-free PCG solver discussed in Algorithm 1, using both ***ebe*** and ***nbn*** strategies given in Algorithm 2, and 3 respectively. For each benchmark example, five different mesh sizes are used to evaluate the scalability of the proposed strategies.

The simulations are performed on a CPU with Intel Xeon E5-1650 processor equipped with 6 cores and 12 threads. It has clock speed of 3.2 GHz and offers maximum memory bandwidth of 51.2 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. The computational

TH-3072_146103004

steps performed on the CPU were written in C, whilst GPU kernels were developed using CUDA 10.0. For topology optimization an artificial material is considered and its properties are given in Table 3.1. It is noted that the implementation is made unit-independent, and changes in material properties do not affect the topology. The other common parameters are also listed in Table 3.1. The optimization loop terminates when the difference between compliance value of two subsequent iterations falls below 10^{-2} or the number of optimization iterations exceeds 150.

Table 3.1: Common parameters used for topology optimization of three examples.

Parameter	Young's modulus E	Poisson's ratio ν	SIMP penalty parameter p	Min. density ρ_{min}	PCG tolerence ε	Change in compliance $\Delta = C_i - C_{i-1}$
Value	1.0	0.3	3.0	10^{-2}	10^{-5}	10^{-2}

The four benchmark examples are discussed in the following subsections.

3.3.1 3D Cantilever Beam

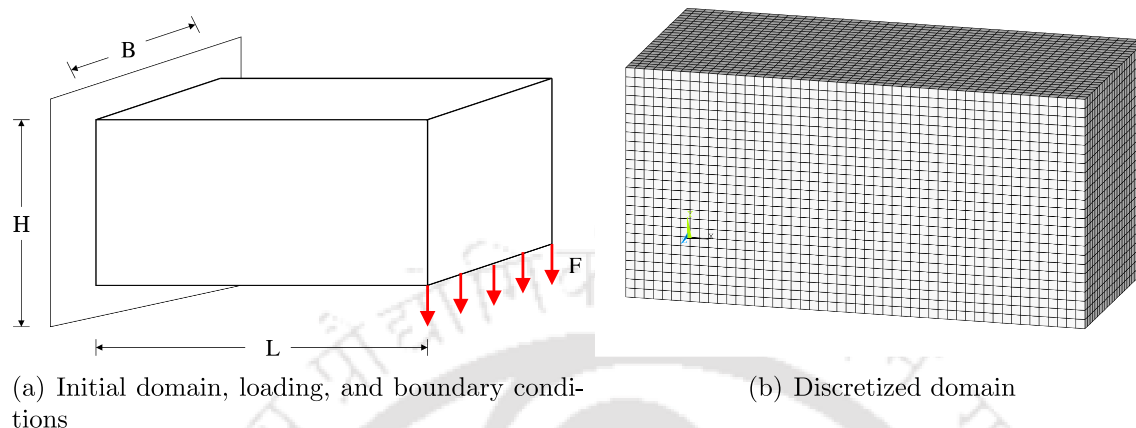
The first example is a 3D cantilever beam [50] and its domain and boundary conditions are shown in Figure 3.7(a). The $L : B : H$ ratio of the cantilever beam is $2 : 1 : 1$. Each node of the left face is subjected to the Dirichlet boundary condition, while each node at the lower edge of the right face is subjected to the Neumann boundary condition. Figure 3.7(b) shows the discretized domain of the beam. It can be observed that the mesh is structured and composed of identical 8-noded hexahedral elements.

The volume fraction (V_f) for this example is taken as 30% of the initial domain volume. The total number of elements, nodes, DoFs and the sensitivity filter radius ($\mathcal{R}$) in the five mesh sizes used for this example are given in Table 3.2.

3.3.2 3D L-Beam

The second benchmark example is a 3D L-beam that is shown in Figure 3.8(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

TH-3072_146103004



(a) Initial domain, loading, and boundary conditions

(b) Discretized domain

Figure 3.7: 3D Cantilever beam example.

Table 3.2: Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of 3D cantilever beam example.

Mesh	# Elements	# Nodes	# DoFs	Filter radius ($\mathcal{R}$)
CB_1	$31,250$	$34,476$	$103,428$	3.0
CB_2	$85,750$	$92,016$	$276,318$	3.5
CB_3	$182,250$	$192,556$	$577,668$	4.0
CB_4	$432,000$	$450,241$	$1,350,723$	4.5
CB_5	$1,024,000$	$1,056,321$	$3,168,963$	5.0

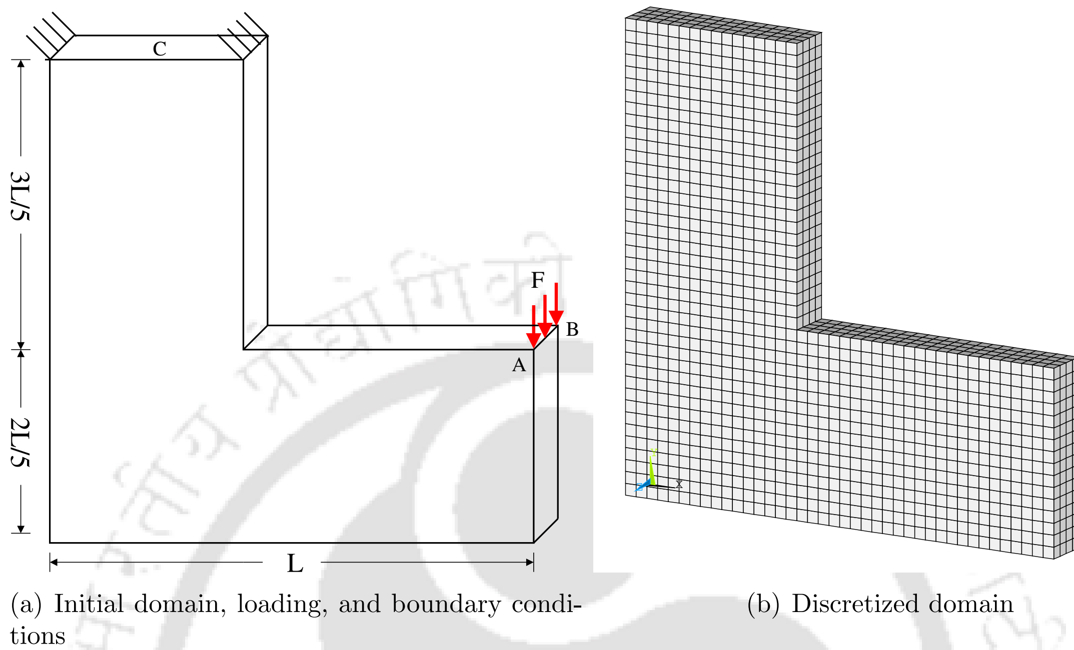
optimization, a unit load is applied at each node of the edge 'AB'. The discretized design domain of 3D L-beam example is shown in Figure 3.8(b). As can be seen, this example is also meshed using a structured mesh of 8-noded hexahedral elements.

For optimization, $V_f = 45\%$ is considered. The details of the mesh sizes used for 3D L-beam example are given in Table 3.3.

3.3.3 Michell Cantilever

In topology optimization, the Michell cantilever is a well-known benchmark example [97]. This example has a semi-circular boundary that is supported as shown in Figure 3.9(a). The supported semi-circular boundary is subjected to the Dirichlet boundary condition, whereas the loaded points in Figure 3.9(a) are subjected to the Neumann

TH-3072_146103004



(a) Initial domain, loading, and boundary conditions

(b) Discretized domain

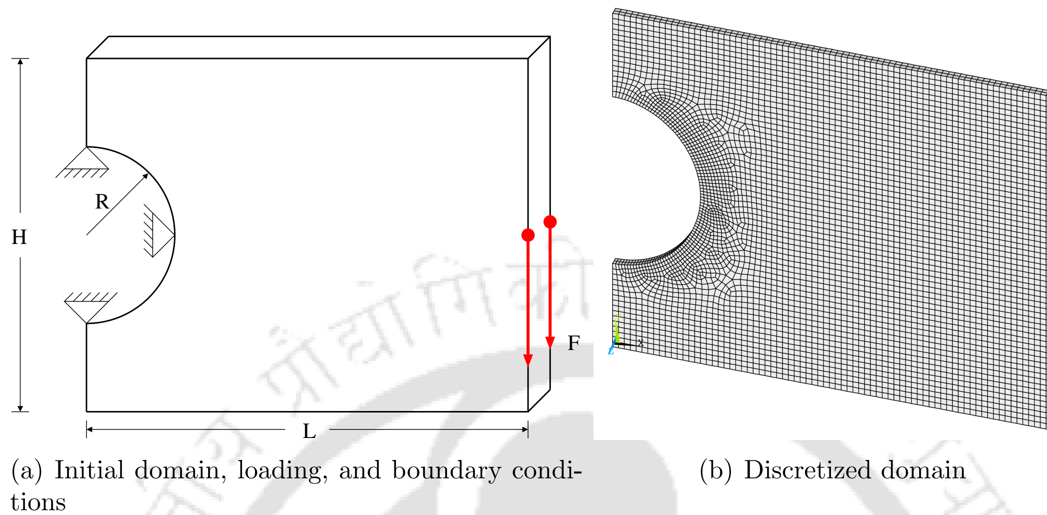
Figure 3.8: 3D L-beam example.

Table 3.3: Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of 3D cantilever beam example.

Mesh	# Elements	# Nodes	# DoFs	Filter radius ($\mathcal{R}$)
LB_1	$32,768$	$38,313$	$114,939$	3.0
LB_2	$85,184$	$95,580$	$286,740$	3.5
LB_3	$188,384$	$206,205$	$618,615$	4.0
LB_4	$438,976$	$469,700$	$1,409,100$	4.5
LB_5	$1,000,000$	$1,053,026$	$3,159,078$	5.0

boundary condition. The $L : H : R$ ratio is taken as $5 : 4 : 1$. V_f is limited to 45% of the design domain volume. Since this example has a semi-circular boundary, unstructured mesh can be seen in Figure 3.9(b). The five mesh sizes used for this example are listed in Table 3.4.

TH-3072_146103004



(a) Initial domain, loading, and boundary conditions

(b) Discretized domain

Figure 3.9: Michell cantilever example.

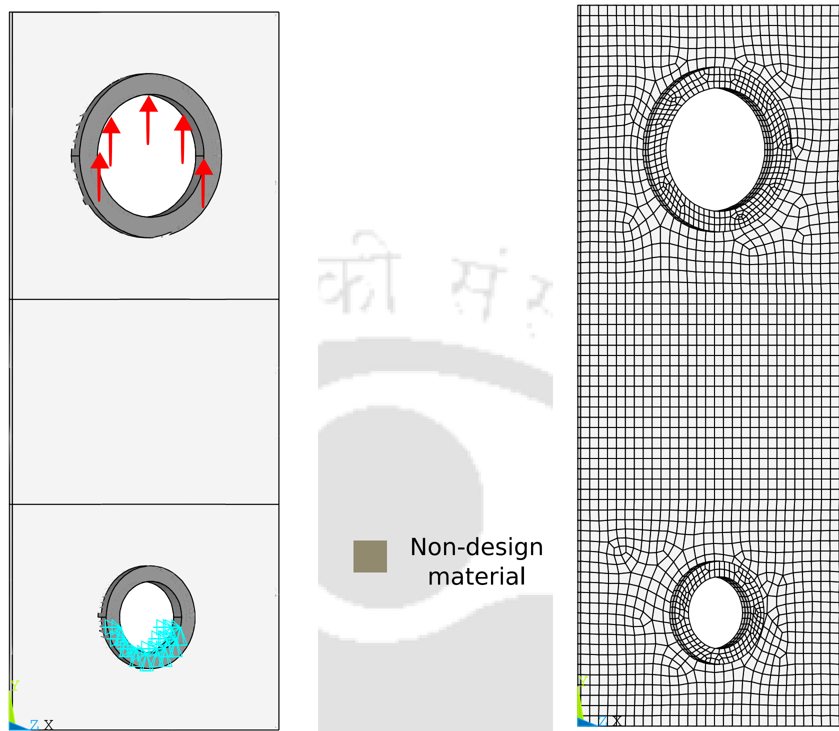
Table 3.4: Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of Michell cantilever example.

Mesh	# Elements	# Nodes	# DoFs	Filter radius ($\mathcal{R}$)
MC_1	$38,148$	$52,008$	$156,024$	3.0
MC_2	$85,674$	$116,012$	$348,036$	3.5
MC_3	$183,432$	$232,125$	$696,375$	4.0
MC_4	$453,175$	$526,998$	$1,580,994$	4.5
MC_5	$1,170,894$	$1,374,415$	$4,123,245$	5.0

3.3.4 Connecting Rod of an Auto-mobile Engine

The fourth benchmark example is a connecting rod of an auto-mobile engine taken from Nana et al. [98]. Figure 3.10(a) shows the design domain and its loading and boundary conditions. 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 3.10(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. $V_f = 30\%$ is considered for this example.

TH-3072_146103004



(a) Initial domain, loading, and boundary conditions

(b) Discretized domain

Figure 3.10: Connecting rod of an auto-mobile engine example.

Since the design domain has two bores, unstructured meshes can be seen in Figure 3.10(b). The mesh sizes used for connecting rod are listed in Table 3.5.

Table 3.5: Number of elements, nodes, DoFs and corresponding filter radius for mesh sizes of connecting rod example.

Mesh	# Elements	# Nodes	# DoFs	Filter radius ($\mathcal{R}$)
CR_1	33, 178	48, 248	144, 744	3.0
CR_2	85, 376	124, 305	372, 915	3.5
CR_3	182, 637	238, 578	715, 734	4.0
CR_4	465, 708	572, 978	1, 718, 934	4.5
CR_5	1, 033, 820	1, 223, 392	3, 670, 176	5.0

TH-3072_146103004

3.4 Results and Discussion

In this section, the results of numerical experiments performed using four benchmark examples are discussed.

3.4.1 Optimal Topology

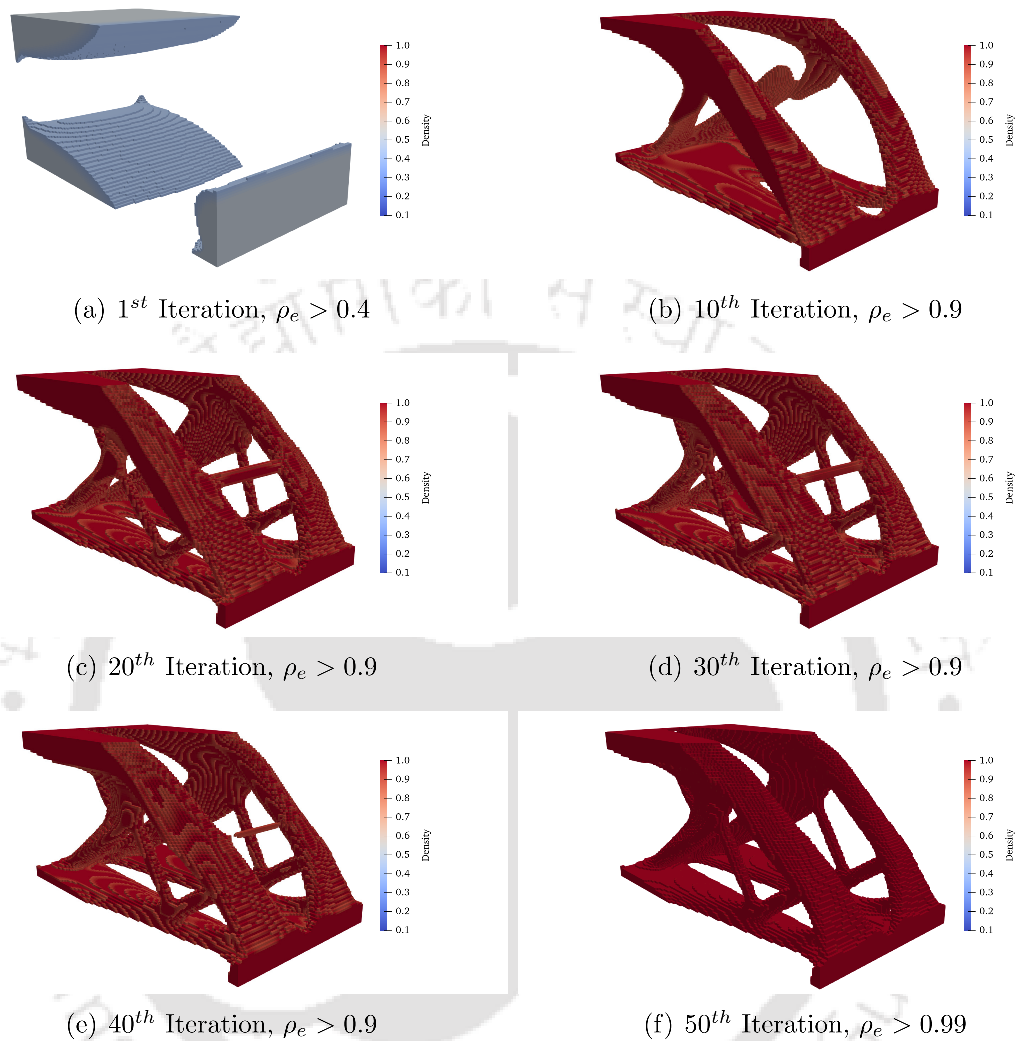
Figure 3.11 depicts the evolution of optimal topology for a 3D cantilever beam example through optimization iterations. At the start of optimization, the density vector is initialized with a small value and hence, the topology consists of only low-density elements, as can be observed in Figure 3.11(a). As the iterations progress, the optimal topology with solid elements ($\rho_e > 0.9$) begins to emerge, as shown in Figure 3.11(b). At the initial stages of optimization, a horizontal member can be seen forming, which later vanishes around 40th iteration. Figure 3.11(f) depicts the optimal topology after 50 optimization iterations, with lower element densities filtered out and only $\rho_e > 0.99$ elements are shown. The optimal topology of 3D cantilever beam example is similar to the one reported by Schmidt and Schulz [50].

For 3D L-beam example the evolution of topology is shown in Figure 3.12. Similar to the previous example, the initial iterations consist of only low-density elements. However, the optimal topology starts to take shape rather quickly, as can be observed in Figure 3.12(b). The optimal topology after 50 iterations is shown in Figure 3.12(f). The lower element densities are filtered out, and the figure shows for $\rho_e > 0.9$.

The changes in topology for Michell cantilever example is shown in Figure 3.13. The low-density elements start to vanish from the topology at the early stages of optimization. Figure 3.13(f) shows the optimal topology with $\rho_e > 0.9$ after 50 iterations.

Figure 3.14 shows the evolution of topology for the connecting rod example. As shown in Figure 3.14(a), since the bores are considered non-design parts of the domain, they already consist of solid material. As optimization proceeds, the material is positioned around the two bores to join them. As depicted in Figures 3.14(b) and 3.14(c), the formation of cross connections between the bores can be observed in the early stages. These intermediary linkages are eliminated after 30th iteration. Figure 3.14(f) shows the optimal topology after 50 iterations, which is identical to the topology reported by Nana et al. [98].

TH-3072_146103004



(a) 1st Iteration, $\rho_e > 0.4$ (b) 10th Iteration, $\rho_e > 0.9$

(c) 20th Iteration, $\rho_e > 0.9$ (d) 30th Iteration, $\rho_e > 0.9$

(e) 40th Iteration, $\rho_e > 0.9$ (f) 50th Iteration, $\rho_e > 0.99$

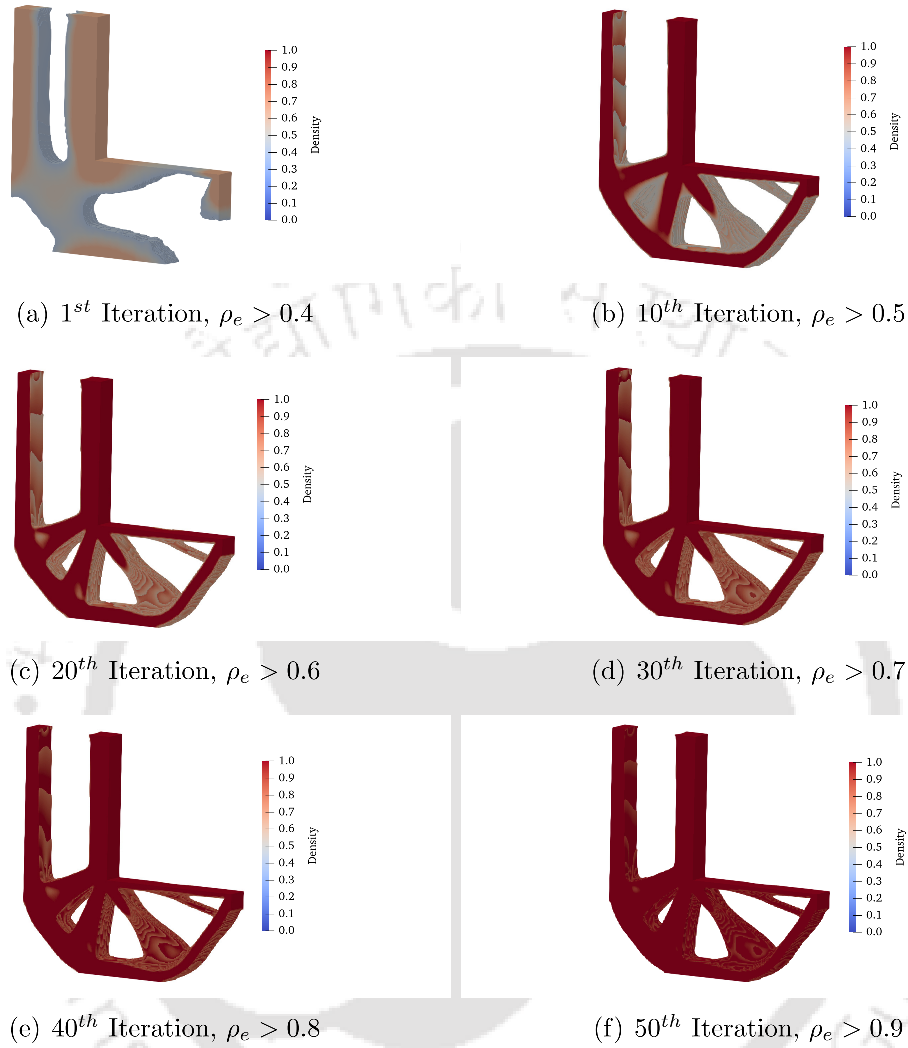
Figure 3.11: Evolution of optimal topology of 3D cantilever beam example corresponding to the mesh CB_5.

3.4.2 Computational Performance

The matrix-free PCG solver is the main computational bottleneck of the topology optimization framework, and it is performed on the GPU. The computational performance of the proposed matrix-free PCG solver using both ***nbn*** and ***ebe*** strategies is discussed here.

The first performance parameter is the PCG solver's execution time and the corresponding speedups. Generally, the speedup is defined in terms of the relative performance

TH-3072_146103004



(a) 1st Iteration, $\rho_e > 0.4$ (b) 10th Iteration, $\rho_e > 0.5$

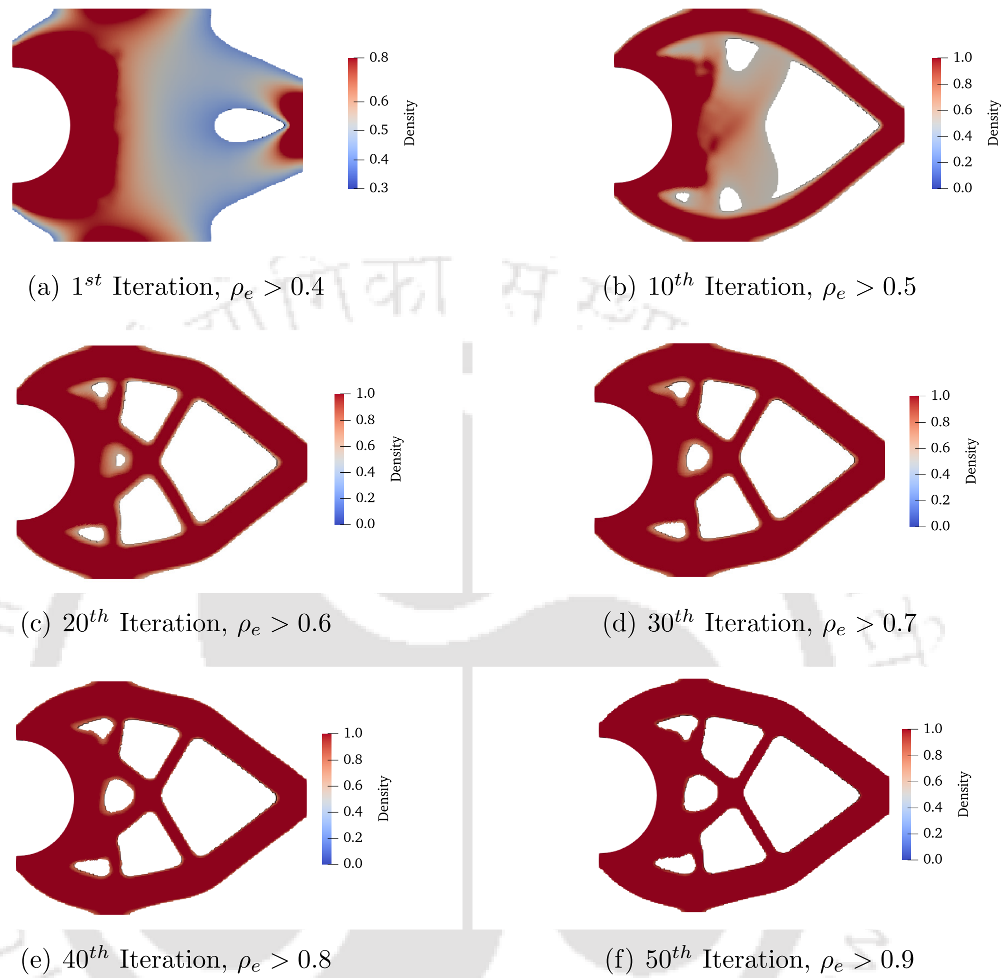
(c) 20th Iteration, $\rho_e > 0.6$ (d) 30th Iteration, $\rho_e > 0.7$

(e) 40th Iteration, $\rho_e > 0.8$ (f) 50th Iteration, $\rho_e > 0.9$

Figure 3.12: Evolution of optimal topology of 3D L-beam example corresponding to the mesh LB_5.

of two systems processing the same task. Both ***nbn*** and ***ebe*** strategies are compared to their respective CPU-based single thread versions, referred to as '***nbnCPU***' and '***ebeCPU***'. Figure 3.15 shows the PCG execution time of both ***nbn*** and ***ebe*** GPU-based SpMV strategies and their respective ***nbnCPU*** and ***ebeCPU*** CPU versions for five mesh sizes of all four benchmark examples. Figure 3.15 shows that for all mesh sizes, ***nbn*** strategy requires a significantly higher execution time than ***ebe*** strategy. On the right axes of the plots, the speedups are shown, and it can be seen that for all mesh sizes, the ***ebe*** strategy achieves a higher speedup than the ***nbn*** strategy. The speedups

TH-3072_146103004



(a) 1st Iteration, $\rho_e > 0.4$ (b) 10th Iteration, $\rho_e > 0.5$

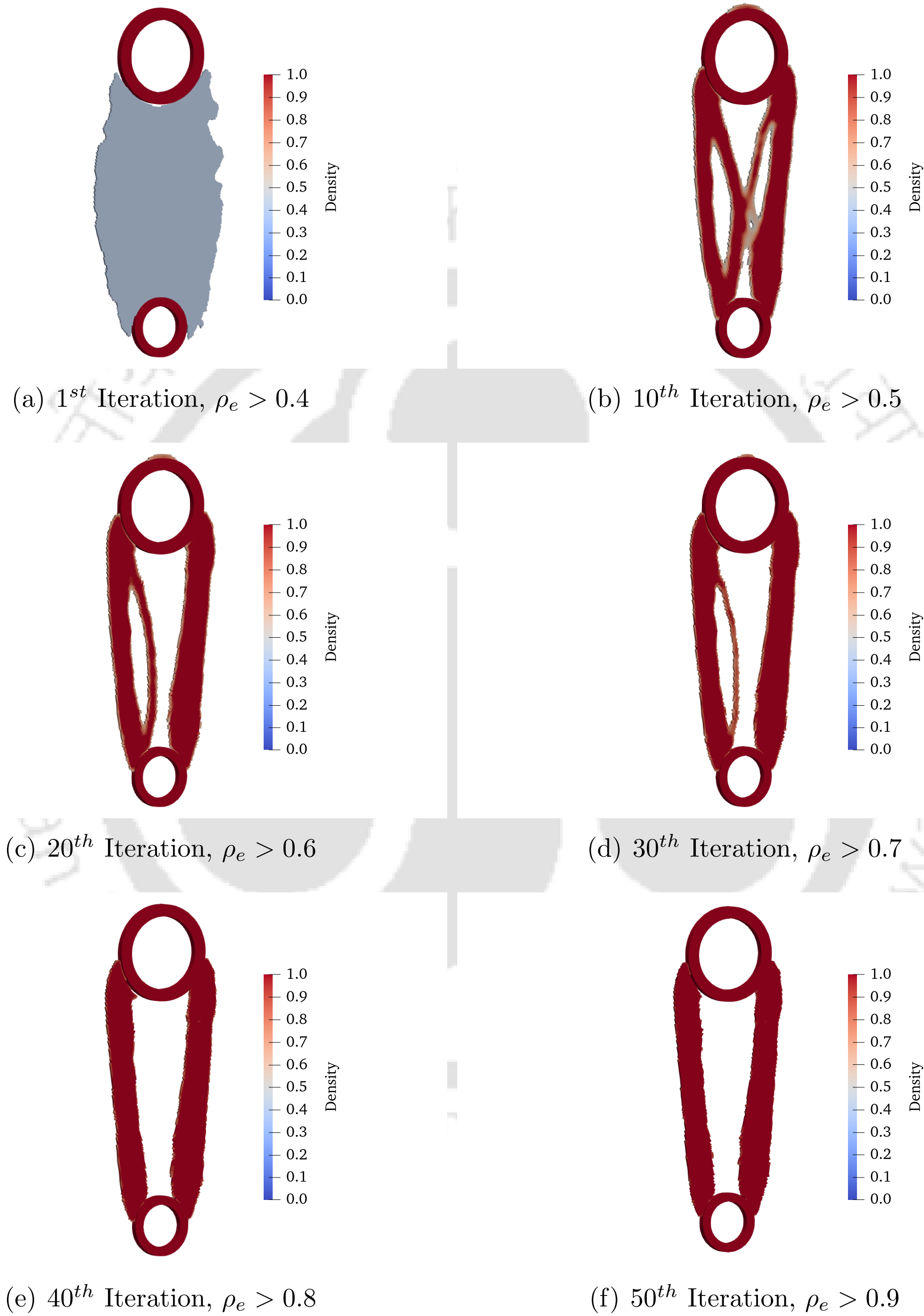
(c) 20th Iteration, $\rho_e > 0.6$ (d) 30th Iteration, $\rho_e > 0.7$

(e) 40th Iteration, $\rho_e > 0.8$ (f) 50th Iteration, $\rho_e > 0.9$

Figure 3.13: Evolution of optimal topology of Michell cantilever example corresponding to the mesh MC_5.

achieved by the ***ebe*** strategy for the smallest mesh sizes of four benchmark problems varies in the range of 1.6× to 2.3×. For the four benchmark examples, ***ebe*** strategy achieves the highest speedups of 3.7×, 3.9×, 4×, and 3.9×, respectively. Initially, the speedup of ***ebe*** strategy scales up with the increase in mesh sizes, however for significantly larger meshes, the speedup is found to decrease slightly. SpMV on GPU requires $\mathbf{K}_e$ of all elements stored in the global memory of the GPU. In addition, the SpMV of a finite element is executed by a single GPU thread, involving several memory read operations between the thread and global memory. For large meshes, the number of memory transactions becomes notably high; hence, the exhibited speedup decreases for

TH-3072_146103004



(a) 1st Iteration, $\rho_e > 0.4$

(b) 10th Iteration, $\rho_e > 0.5$

(c) 20th Iteration, $\rho_e > 0.6$

(d) 30th Iteration, $\rho_e > 0.7$

(e) 40th Iteration, $\rho_e > 0.8$

(f) 50th Iteration, $\rho_e > 0.9$

Figure 3.14: Evolution of optimal topology of connecting rod example corresponding to the mesh CR_5.

TH-3072_146103004

larger meshes.

The speedup attained by the ***nbn*** strategy for the smallest meshes ranges from 1.4× to 2×, while the highest speedups are 3.9×, 2.9×, 2.4×, and 2.5×. Similar to ***ebe*** strategy, the speedup of ***nbn*** strategy first increases with the increment in mesh size and then decreases for larger meshes. In ***nbn*** strategy, the compute thread reads the $\mathbf{K}_e$ of every element shared by the node it is allocated to. These global memory read operations are not coalesced, wasting the GPUs' computing resources. In addition, ***nbn*** strategy requires $\mathbf{C}_{rev}$, which substantially increases the number of memory transactions. In ***nbn*** strategy a node may be connected to different number of elements, especially in case of boundary nodes. This creates a computational load imbalance among the threads. These factors contribute to the inferior performance of the ***nbn*** strategy relative to the ***ebe*** strategy.

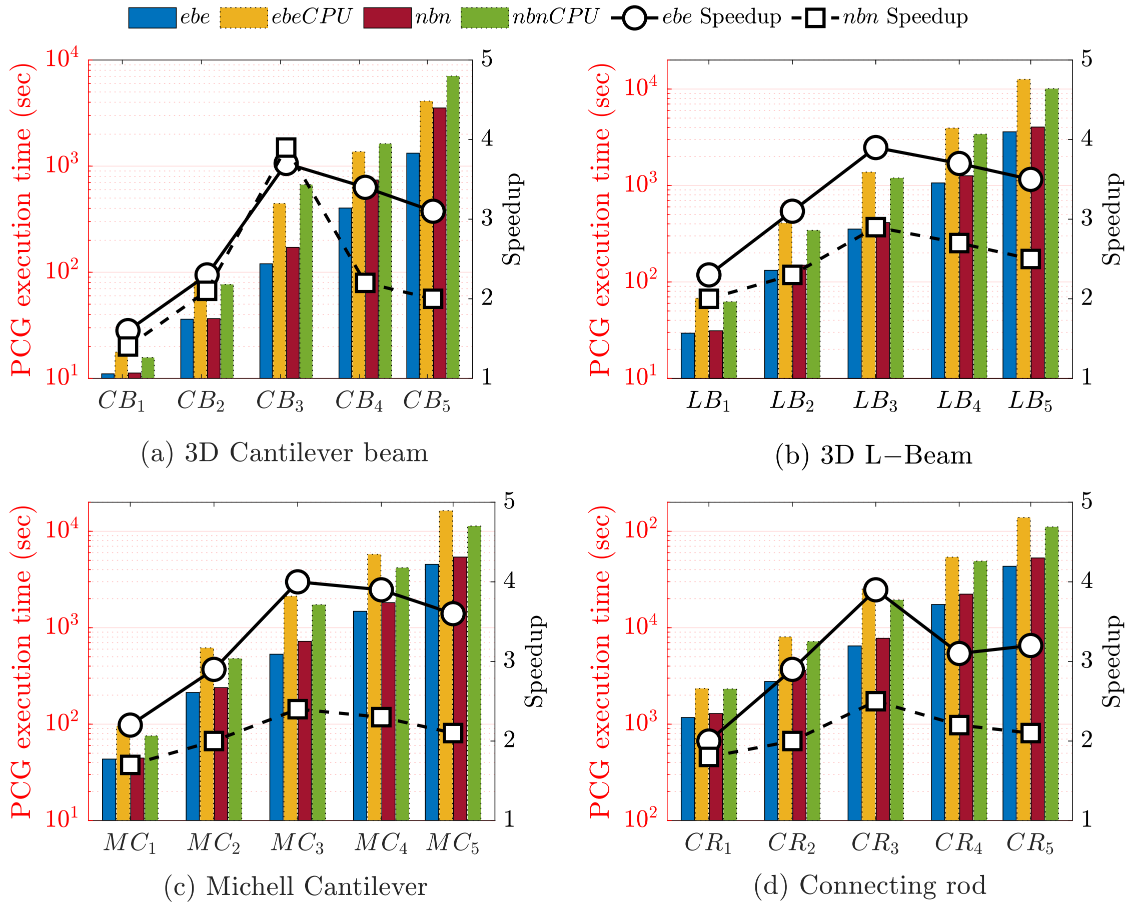
Table 3.6 displays the average number of PCG iterations required to converge for the four benchmark examples using various mesh sizes. Notably, the proposed ***nbn*** and ***ebe*** strategies only alter the manner in which SpMV computations are performed. Since the algorithm of the PCG solver remains the same, the selection of the SpMV strategy has no effect on the number of PCG iterations required for an example to converge. It can be observed that the benchmark examples with unstructured meshes require more number of solver iterations to converge.

Table 3.6: Average number of PCG iterations taken to converge by different mesh sizes of four benchmark examples.

Example	Mesh 1	Mesh 2	Mesh 3	Mesh 4	Mesh 5
CB	887	1107	1682	2305	3140
LB	2382	4671	5836	7754	11224
MC	3357	7132	8364	10064	11636
CR	8046	9468	10246	11135	12695

Next, the execution time of various computational steps of the GPU-based topology optimization framework shown in Figure 3.6 is analyzed. The purpose of this analysis is to identify the computational bottleneck of the proposed topology optimization framework. Figure 3.16 shows the percentage of total wall-clock time ($\%WC$) attributed to these computational steps. The wall-clock time is the actual time from the start to the finish of a computer program. The percentages are computed relative to the wall-clock

TH-3072_146103004



(a) 3D Cantilever beam

(b) 3D L−Beam

(c) Michell Cantilever

(d) Connecting rod

Figure 3.15: PCG execution time for ***nbn*** and ***ebe*** strategies and the gained speedups against their respective CPU-based single thread versions ***nbnCPU*** and ***ebeCPU***.

time of the complete topology optimization for each of the four benchmark examples' five mesh sizes. '*preComp*' encompasses all preliminary computations, including reading mesh and connectivity data from files, allocating and initializing GPU memory, and computing $\mathbf{K}_e$. This step for ***nbn*** strategy includes the time required to construct the $\mathbf{C}_{rev}$. The '*PCG*' step includes the total $\%WC$ required by the proposed GPU-based PCG solver. '*Sensitivity*' includes the computation of structural compliance and the sensitivity analysis. '*Filter*' includes the amount of time required by the mesh-independence filter. '*Update*' refers to the time needed to update the design variables, whereas '*PostProcess*' reflects the time required to write results to files.

TH-3072_146103004

For smaller meshes, the percentage share of *preComp* for ***ebe*** strategy varies in the range of 0.09% to 0.64%, whereas for ***nbn*** strategy, it ranges from 0.2% to 1.1%. For the largest mesh, these values ranges from 2% to 3%, and from 4% to 5%. The additional time required to generate $\mathbf{C}_{rev}$ increases the percentage of *preComp* assigned to ***nbn*** strategy.

As seen in Figure 3.16, the *PCG* step holds the largest percentage share across all mesh sizes. Depending on the mesh size, the $\%WC$ value for ***ebe*** strategy ranges from 92% to 95% of the total wall-clock time. Comparatively, the range for ***nbn*** strategy is from 95% to 99%. For the same mesh size, it is also noted that when ***nbn*** strategy is used, the $\%WC$ of *PCG* is higher.

The *Sensitivity* step for ***ebe*** strategy requires from 0.63% to 5.9% for the smallest meshes of four benchmark examples and between 0.52% and 1.5% for the largest meshes. In comparison, the same numbers for ***nbn*** strategy range from 0.58% to 2.4% and from 0.43% to 6.3%. The $\%WC$ of *Filter* step requires from 0.12% to 0.74% and varies minimally between SpMV strategies. The $\%WC$ of the *Update* step exhibits the similar trend and falls within the range of 0.01% to 0.17%. The *PostProcess* step requires between 0.01% and 0.03% of wall-clock time.

The analysis of results from Figure 3.16 reveals that even when FEA is executed on GPU, it remains the computational bottleneck of the topology optimization framework. In addition, other computational steps require only a small amount of time compared to FEA, so their parallelization will not significantly reduce the total computation time.

3.4.3 Closure

It is noticed that the *PCG* step, even when executed on GPU, consumes the bulk of computing time and is, therefore, the topology optimization framework's bottleneck. The computational time required by the other steps is negligible compared to that required by the *PCG* step. The analysis of computational performance indicates that the ***ebe*** strategy is the better suited SpMV strategy for large-scale unstructured meshes. However, the computational load over a single compute thread in ***ebe*** strategy is still very high. The granularity of the SpMV operation can be further improved by developing suitable strategies to allocate more number of threads to each finite elements.

TH-3072_146103004

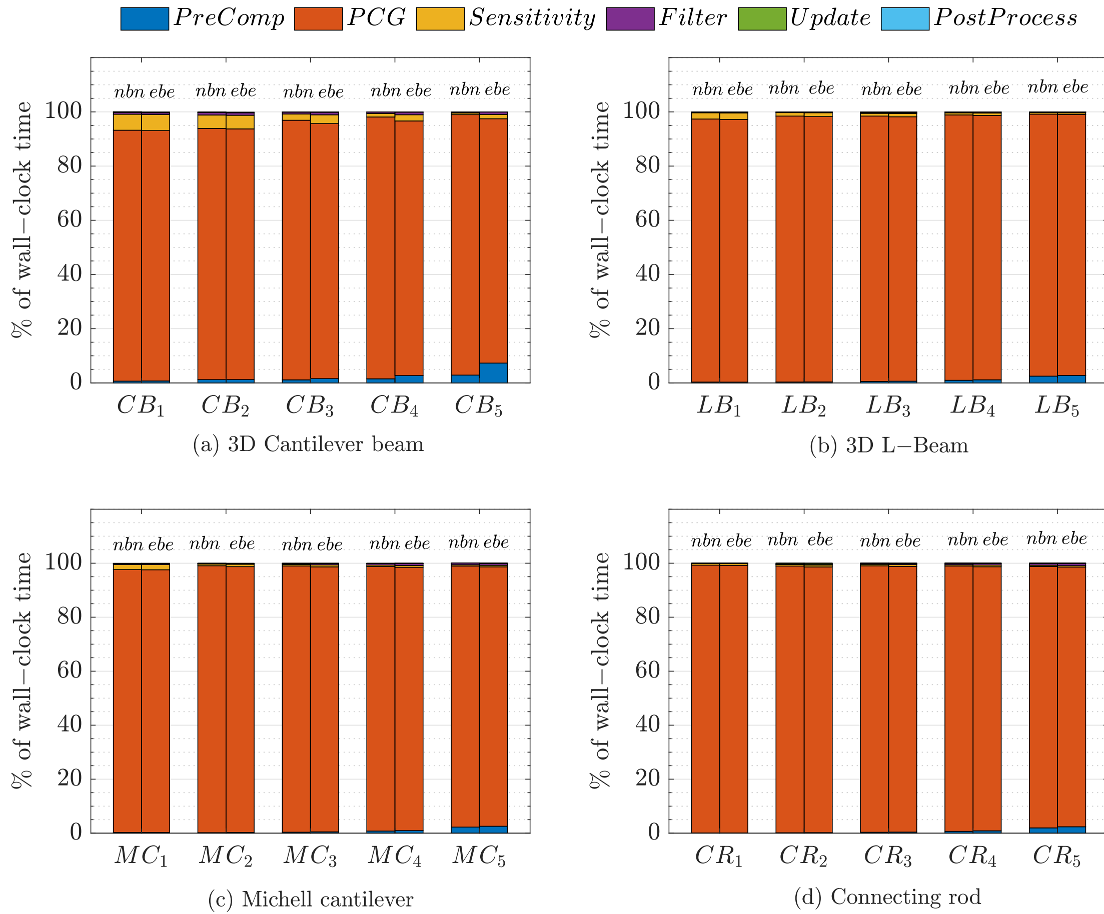


Figure 3.16: Percentage share of various computational steps of topology optimization framework from the wall-clock time, corresponding to the proposed ***ebe*** and ***nbn*** strategies.

Chapter 4

Efficient Thread Allocation Strategies for Matrix-Free FEA Solver

The analysis of the results from the previous chapter revealed that the PCG solver step is the main computational bottleneck of the topology optimization framework, even when executed on GPU. The matrix-free PCG solver consists of SpMV and vector arithmetic (VeA) operations. The percentage of computation time required by both out of PCG computation time is shown in Figure 4.1 for all four benchmark examples. It can be observed that the majority of PCG computation time is taken by the SpMV operation. For smallest mesh SpMV takes between 98.7% and 99% of PCG time, whereas for largest mesh it varies in the range of 99.7% to 99.9%. This is mainly due to the involvement of a large number of floating-point operations as well as memory read operations to access the connectivity and elemental data. Hence, SpMV is the computational bottleneck of the PCG solver step.

The results of the previous chapter also demonstrated that for large unstructured meshes, ***ebe*** is the better suited SpMV strategy. However, the computational load on GPU threads is still very high. In order to further improve the computational performance of the proposed matrix-free PCG solver it is important to develop appropriate

TH-3072_146103004

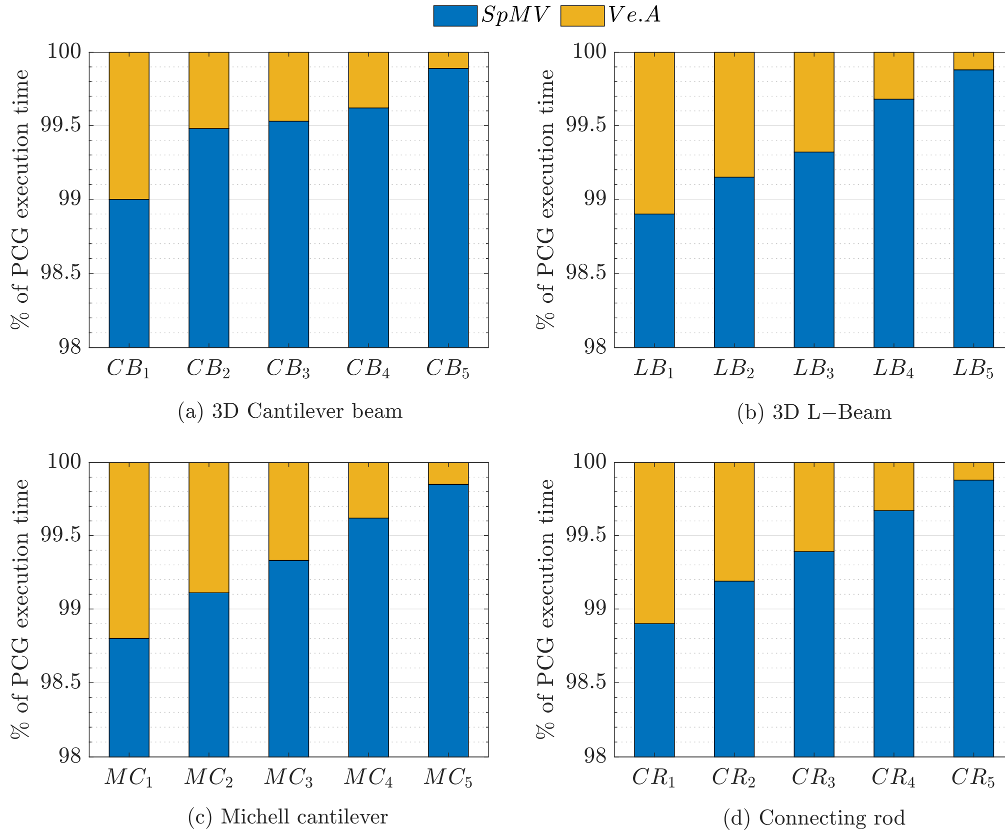


Figure 4.1: Percentage share of SpMV and VeA operations from PCG execution time.

thread allocation strategies for the ***ebe*** strategy to ensure better load distribution among GPU threads.

There are 24×24 entries in the elemental stiffness matrix ($\mathbf{K}_e$) of an 8-noded hexahedral element. The standard ***ebe*** strategy allocates a single GPU thread to perform multiplication for all the entries of $\mathbf{K}_e$ with their corresponding vector elements, as discussed in the previous chapter. In the second objective of the thesis we develop fine-grained SpMV strategies that use various levels of granularity present in $\mathbf{K}_e$ of an element. The proposed strategies are expected to reduce the computational load of a compute thread by allocating more number of threads to an element.

TH-3072_146103004

4.1 Fine-Grained Thread Allocation Strategies for SpMV on GPU

In this thesis three fine-grained GPU-based SpMV strategies are proposed that are discussed in detail in the following subsections.

4.1.1 8-Threads per Element Strategy '*ebe8*'

In the proposed strategy, eight GPU threads are assigned to an element for the ***ebe*** strategy. Since 8-noded hexahedral element is used for meshing, this strategy thus divides the workload of the standard ***ebe*** strategy among eight compute threads of GPU. The multiplication between $\mathbf{K}_e$ and $\boldsymbol{u}_e$ of one finite element 'e' is shown in Figure 4.2. It can be seen that the red boxes divide 24 rows of $\mathbf{K}_e$ into eight groups and each group contains three rows. In this strategy, 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 ***ebe*8** strategy.

The computational steps of the ***ebe*8** strategy are given in Algorithm 4. The data requirement is similar to the standard ***ebe*** strategy. 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 $\mathbf{C}$ 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 $\mathbf{K}_e$. 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 $\mathbf{K}_e$ 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 4.2 show the elements that are multiplied in one iteration of *for-loop* in line 10.

TH-3072_146103004

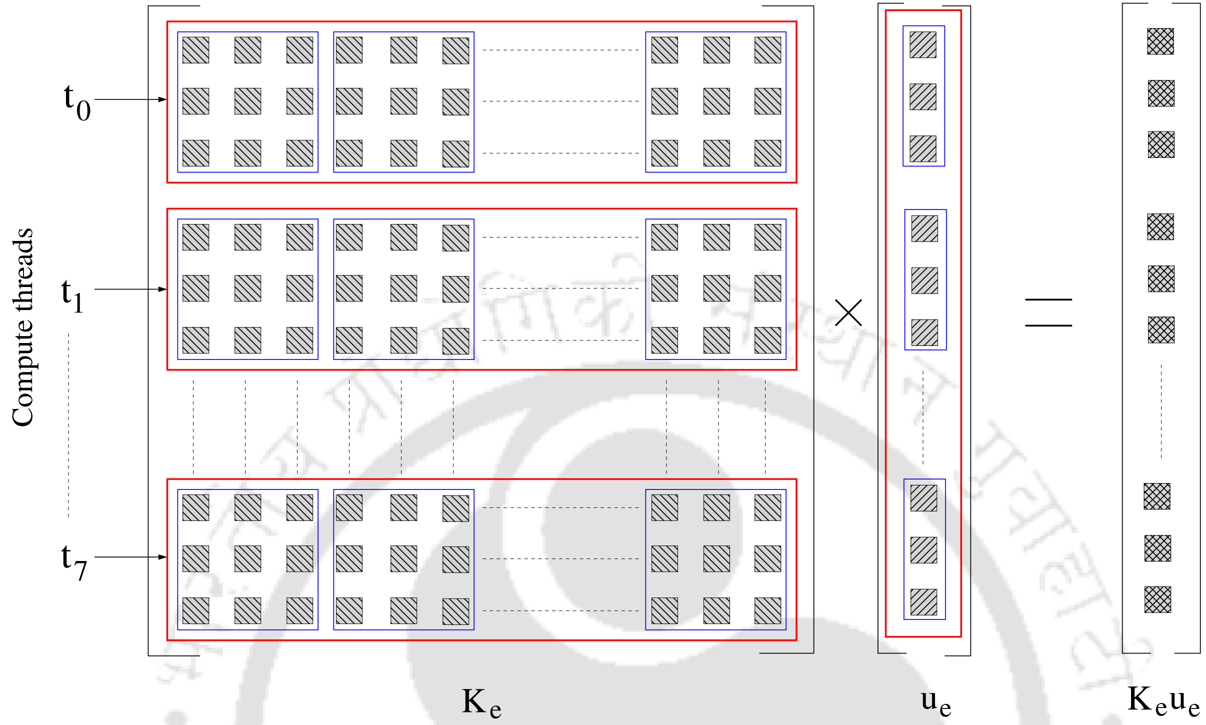


Figure 4.2: Matrix-vector multiplication by the proposed ***ebe*8** strategy.

4.1.2 24-Threads per Element Strategy '*ebe24*'

The computational load is further divided in this strategy by assigning one row of $\mathbf{K}_e$ to a thread. The strategy thus allocates 24 threads to each 8-noded hexahedral finite element in the mesh. Figure 4.3 shows grouping of $\mathbf{K}_e$ entries among GPU threads. The $\mathbf{K}_e$ 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 ***ebe*24** strategy.

The computational steps of the ***ebe*24** 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 $\mathbf{K}_e$, 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 $\mathbf{K}_e$ in line 12. Index id_2 is read from $\boldsymbol{C}_s$ for each value of col. Finally, in line 15, the thread computes $\mathbf{K}_e \times \boldsymbol{u}_e$, and the penalized density (ρ_e) is multiplied by the product. By using the CUDA's atomic operation, the output of

TH-3072_146103004

Algorithm 4: Proposed GPU-based fine-grained SpMV strategy '***ebe*8**.'

```
    Data: K, C, u, ρ, p, Elem
    Output: r
 1  t = blockIdx.x × blockDim.x + threadIdx.x;
 2  __shared__ C_s[size];                                   // shared memory space
 3  if t < (Elem * 8) then
 4  |   e = (int) (t/8);                                    // global element index
 5  |   C_s ← copy from C;
 6  |   __syncthreads( );
 7  |   ρ_e = ρ[e]^p;
 8  |   row = t − (e × 8);                                  // row number
 9  |   id_1 ← C_s;
10  |   for col ← 0 to 7                                    // columns of K_e
11  |    do
12  |   |   id_2 ← C_s;
13  |   |   val = 0.0;
14  |   |   val + = ρ_e × {K_e[row][col] × u_e[id_2]};      // matrix-vector product
15  |   |   r[id_1] + = val;                                // Atomic add
16  __syncthreads( );
17  end
```

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 4.3. Mat-vec operation for the entire row is done by looping over these boxes.

4.1.3 64-Threads per Element Strategy '*ebe64*'

This strategy is proposed for further reducing the computational load of the standard ***ebe*8** strategy by assigning 64 threads to an element. As shown in Figure 4.4, the entries of $\mathbf{K}_e$ are now grouped into 3×3 tiles shown in different colors. Each tile is then assigned to a compute thread that multiplies 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 strategy. This strategy is referred to as the ***ebe*64** strategy.

The computational steps of the ***ebe*64** strategy are shown in Algorithm 6. The initial steps up to line 5 are similar to Algorithm 4. The temporary variable '*val*' is declared

TH-3072_146103004

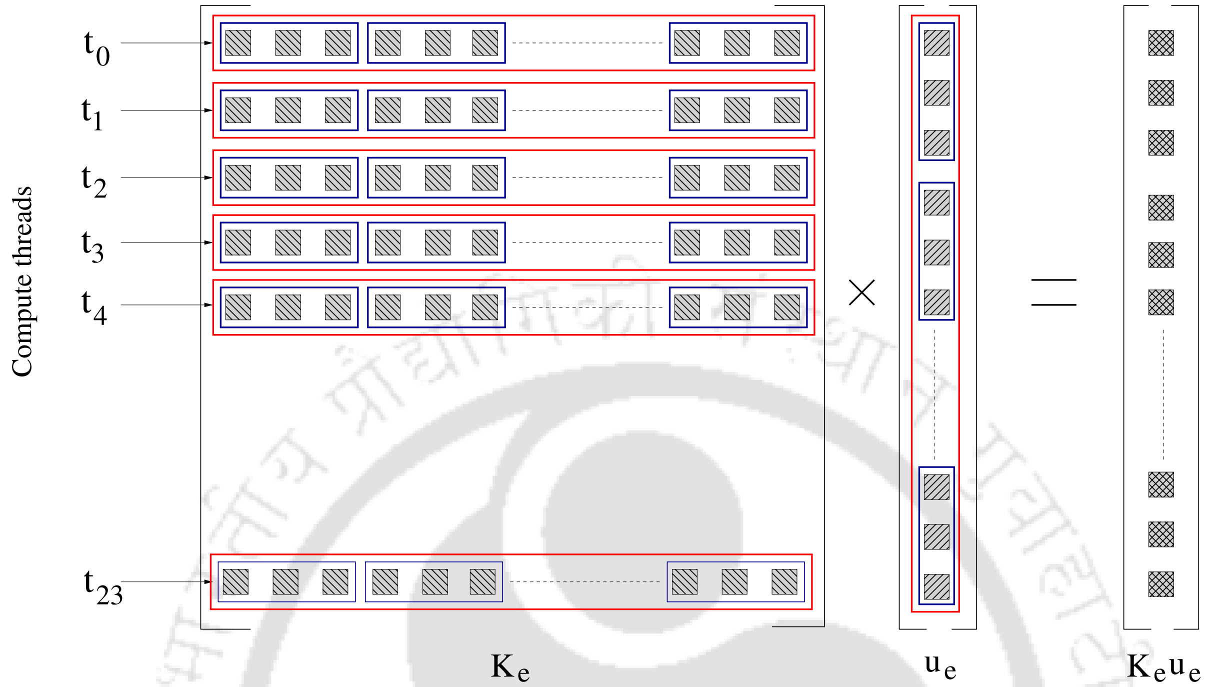


Figure 4.3: Matrix-vector multiplication by the proposed ***ebe*24** strategy.

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** strategies have one *for-loop*, while the ***ebe*64** strategy has none. In addition, shared memory is used in all three fine-grained SpMV strategies to store the connectivity data required by a finite element for reducing the global memory transactions.

4.2 Computational Performance

The computational performance of all three proposed fine-grained SpMV strategies are evaluated using the same benchmark examples discussed in Section 3.3, and mesh sizes

TH-3072_146103004

Algorithm 5: Proposed GPU-based fine-grained SpMV strategy '***ebe*24**.'

```
   Data: K, C, u, ρ, p, Elem
   Output: r
1  t = blockIdx.x * blockDim.x + threadIdx.x;
2  __shared__ C_s[size];                              // shared memory space
3  if threadIdx.x < size then
4  |  C_s[threadIdx.x] ← copy form C;
5  __syncthreads( );
6  if t < (Elem * 24) then
7  |  e = (int) (t/24);                               // global element index
8  |  ρ_e = ρ[e]^p;
9  |  row = t − (e * 24);                             // row index
10 |  id_1 ← C_s;
11 |  val = 0.0;
12 |  for col ← 0 to 7                                // columns of K_e
13 |   do
14 |   |  id_2 ← C_s;
15 |   |  val + = ρ_e * {K_e[row][col] * u_e[id_2]};  // matrix-vector product
16 |  r[id_1] + = val;                               // Atomic add
17 __syncthreads( );
18 end
```

given in Tables 3.2-3.5. The execution time of the PCG solver for each of the three proposed SpMV strategies is measured. The speedup is computed with respect to the ***ebe*** strategy on GPU discussed in Section 3.1.1. For the GPU instances, one dimensional grid of thread blocks were launched with 512 threads per block. For the ***ebe*24** strategy, 504 threads per block were launched since 512 is not a multiple of 24. This block dimension for the ***ebe*24** strategy can help transferring data from global memory to shared memory on GPU.

The PCG execution time of all three fine-grained SpMV strategies for 3D cantilever beam example is shown in Figure 4.5. It can be seen that the ***nbn*** strategy consumes the most time for all mesh sizes, followed by the ***ebe*** strategy. The ***ebe*64** strategy, on the other hand, requires the least amount of time. As a result, the ***ebe*64** strategy shows the highest speedup, followed by the ***ebe*8** strategy. For the ***ebe*8** and ***ebe*64** strategies, there is an increase in speedup with increasing mesh size. In addition, the ***ebe*24** strategy outperforms the ***ebe*** strategy by 3× to 4.5× times for various mesh sizes. Despite the

TH-3072_146103004

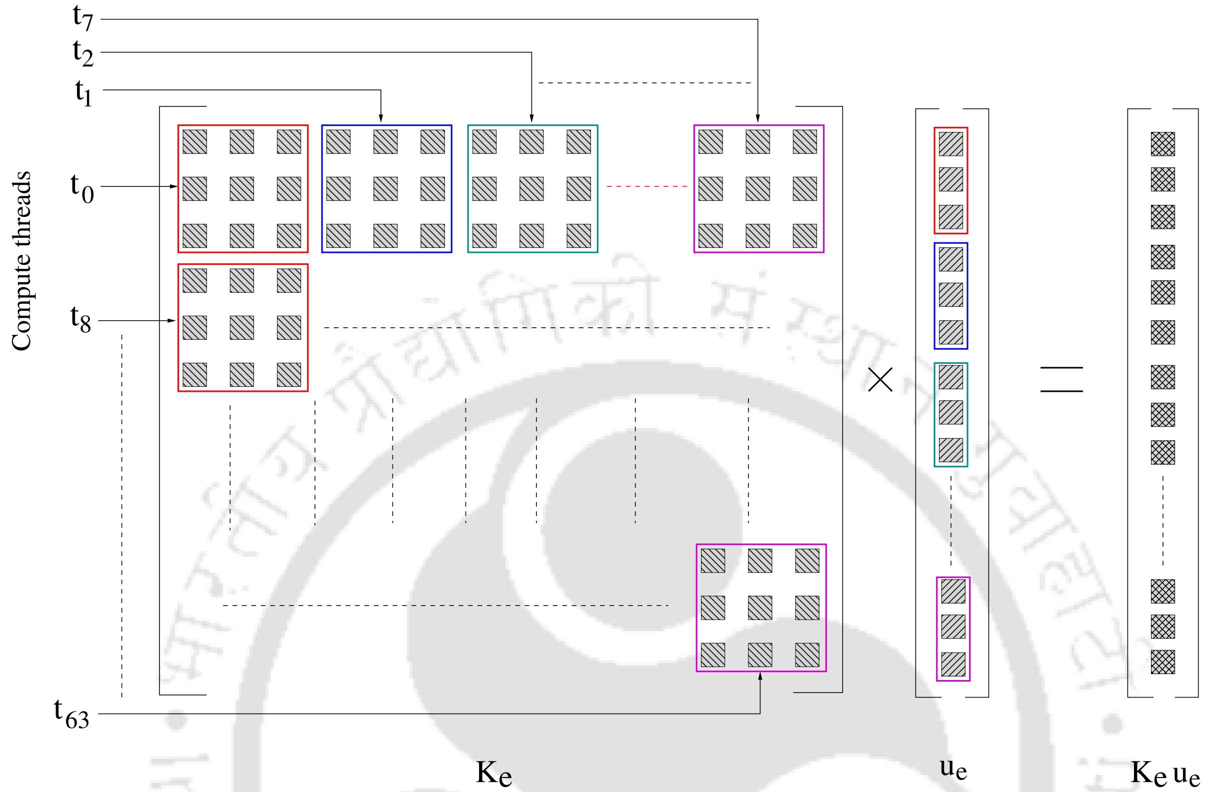


Figure 4.4: Matrix-vector multiplication by the proposed ***ebe*64** strategy.

fact that the ***ebe*24** strategy assigns more threads per element, it is still outperformed by the ***ebe*8** strategy for all mesh sizes. The mesh CB_5 with the ***ebe*64** strategy shows the highest speedup of 8.2×.

Figure 4.6 shows the PCG execution time and speedup for the 3D L-beam example. Similar to the previous example, the ***nbn*** strategy consumes the most time, followed by the ***ebe*** strategy. It is clear from Figure 4.6 that the ***ebe*8** strategy outperforms all other strategies. A linear trend in speedup can be seen with respect to different mesh sizes. The ***ebe*64** strategy is the second best performing strategy, but its speedup is nearly constant as the mesh size increases. The ***ebe*8** and ***ebe*64** strategies outperform the ***ebe*24** strategy here as well. The ***ebe*8** strategy for LB_5 mesh shows the maximum speedup of 8.2× for the 3D L-beam example.

The PCG execution time and speedup for the Michell cantilever example are shown in Figure 4.7. The ***nbn*** strategy has the highest execution time. Over different mesh

TH-3072_146103004

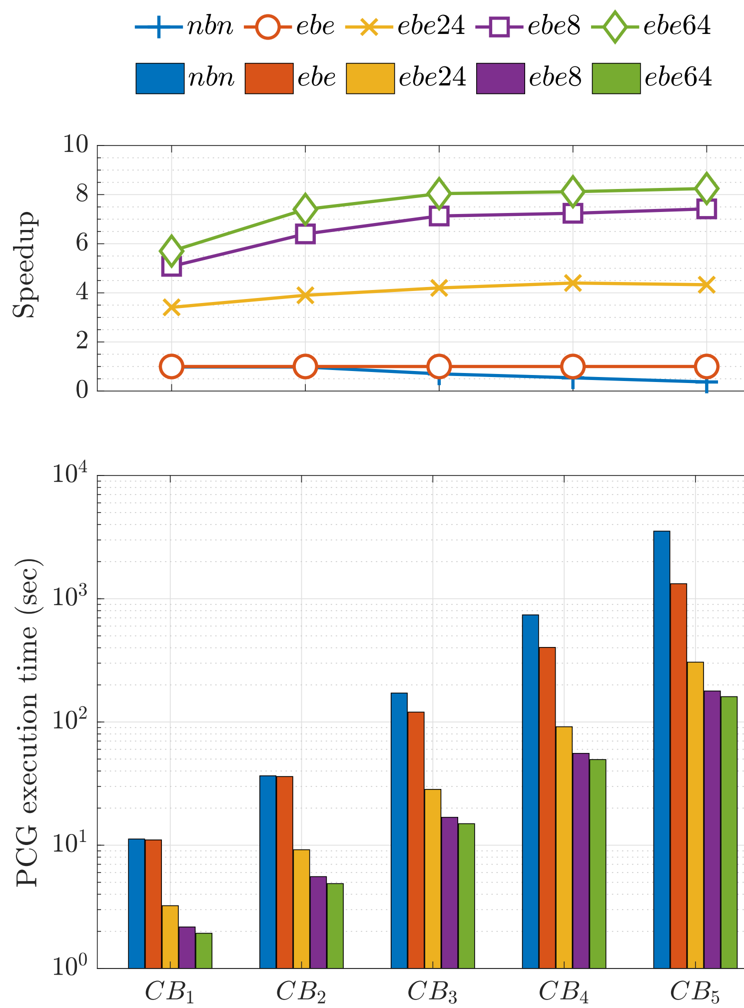


Figure 4.5: PCG execution time, and speedup with respect to the ***ebe*** strategy for 3D cantilever beam example.

TH-3072_146103004

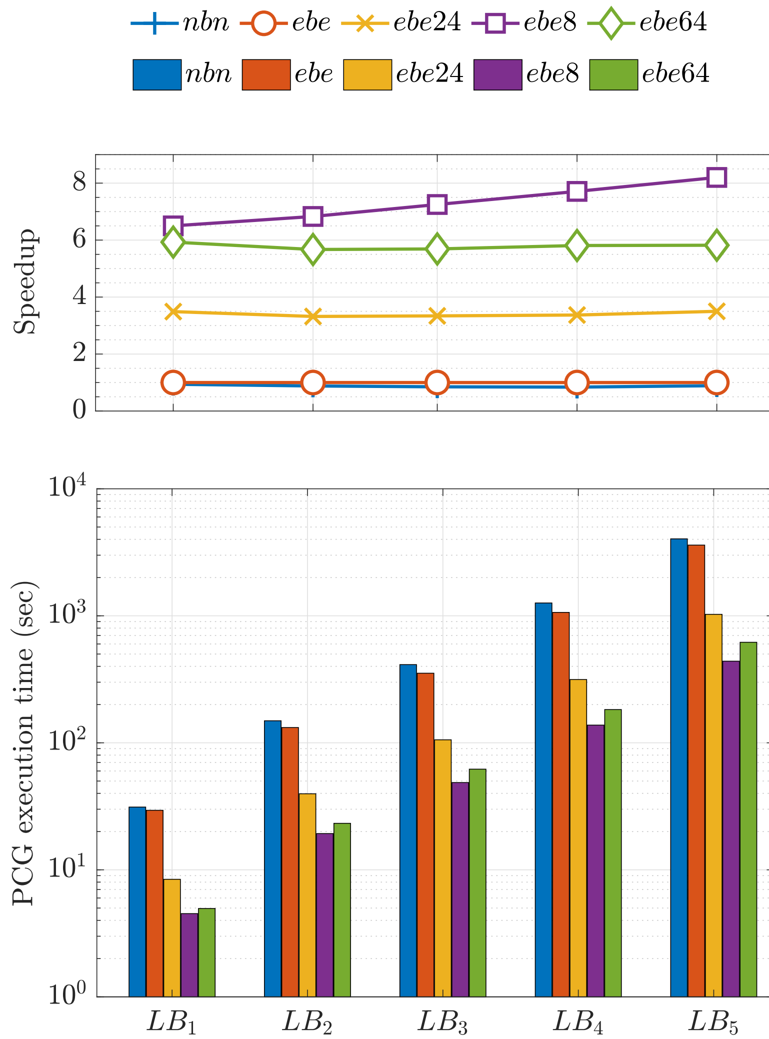


Figure 4.6: PCG execution time, and speedup with respect to the ***ebe*** strategy for 3D L-beam example.

Algorithm 6: Proposed GPU-based fine-grained SpMV strategy '***ebe*64**.'

```
   Data: K, C, u, ρ, p, Elem
   Output: r
1  t = blockIdx.x * blockDim.x + threadIdx.x;
2  __shared__ C_s[size];                                  // shared memory space
3  if threadIdx.x < size then
4  |  C_s[threadIdx.x] ← copy from C
5  __syncthreads( );
6  if t < (Elem * 64) then
7  |  val = 0.0;
8  |  e = (int) (t/64);
9  |  ρ_e = ρ[e]^p;
10 |  row = (t − e * 64)/8;                                // row index
11 |  col = t − (8 * (t/8));                               // column index
12 |  id_1 ← C_s;
13 |  id_2 ← C_s;
14 |  val + = ρ_e * {K_e[row][col] * u_e[id_2]};          // matrix-vector product
15 |  r[id_1] + = val;                                     // Atomic add
16 __syncthreads( );
17 end
```

sizes, the speedup of ***ebe*24** strategy ranges from 3× to 3.6×. It can be seen that the ***ebe*64** strategy performs slightly better than the ***ebe*8** strategy for smaller mesh sizes (MC_1 and MC_2). However, the performance of the ***ebe*8** strategy is better for larger meshes. For the largest mesh MC_5, the ***ebe*8** strategy achieves the highest speedup of 7.2×.

Figure 4.8 shows the PCG execution time and speedup for the connecting rod example. The PCG execution time follows the similar trend with the previous examples. For smaller meshes, the ***ebe*64** strategy outperforms the ***ebe*8** strategy. However, the ***ebe*8** strategy outperforms the ***ebe*64** strategy for CR_4 and CR_5 meshes. The speedup of the ***ebe*24** strategy ranges from 3.3× to 7×. In the connecting rod example, the highest speedup of 7.4× is observed with the ***ebe*8** strategy. This speedup, however, corresponds to the second largest CR_4 mesh. The same strategy shows a speedup of 7.1× 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 strategy in terms of speedup and scaling. The ***ebe*24**

TH-3072_146103004

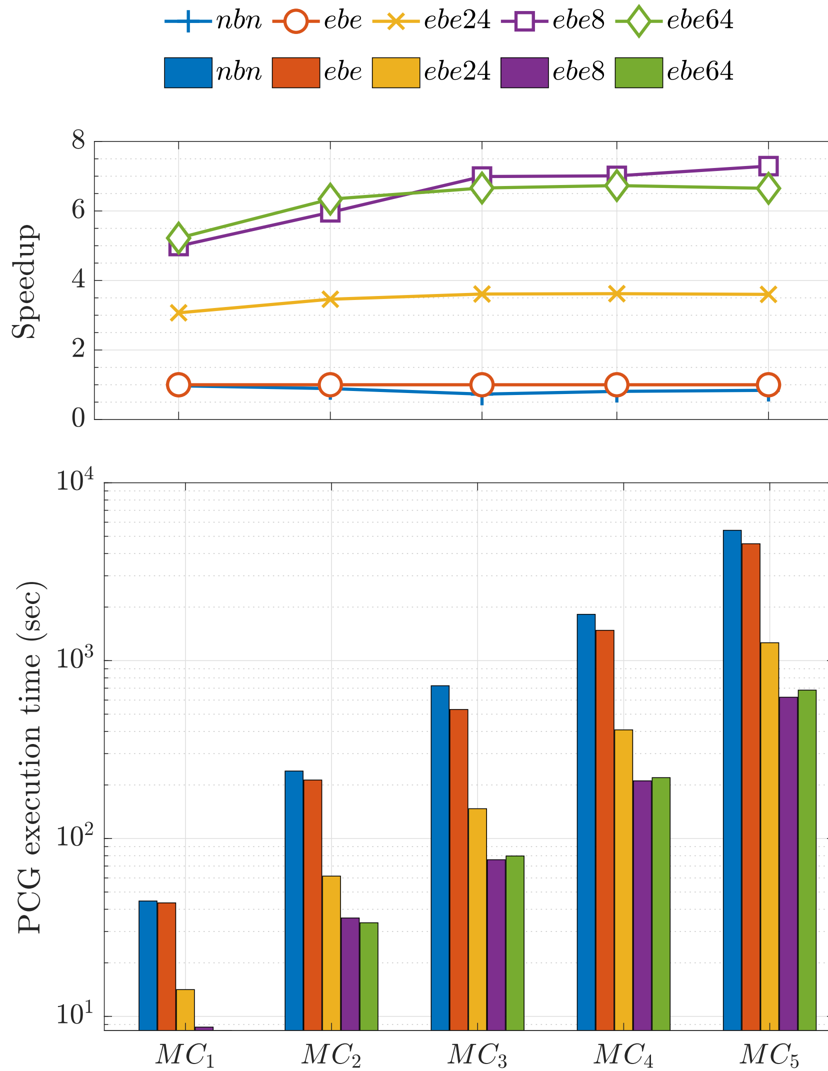


Figure 4.7: PCG execution time, and speedup with respect to the ***ebe*** strategy for Michell cantilever example.

strategy is outperformed by both the ***ebe8*** and ***ebe64*** strategies in all examples. Since the ***ebe24*** strategy 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 ***ebe8*** strategy outperforms the

TH-3072_146103004

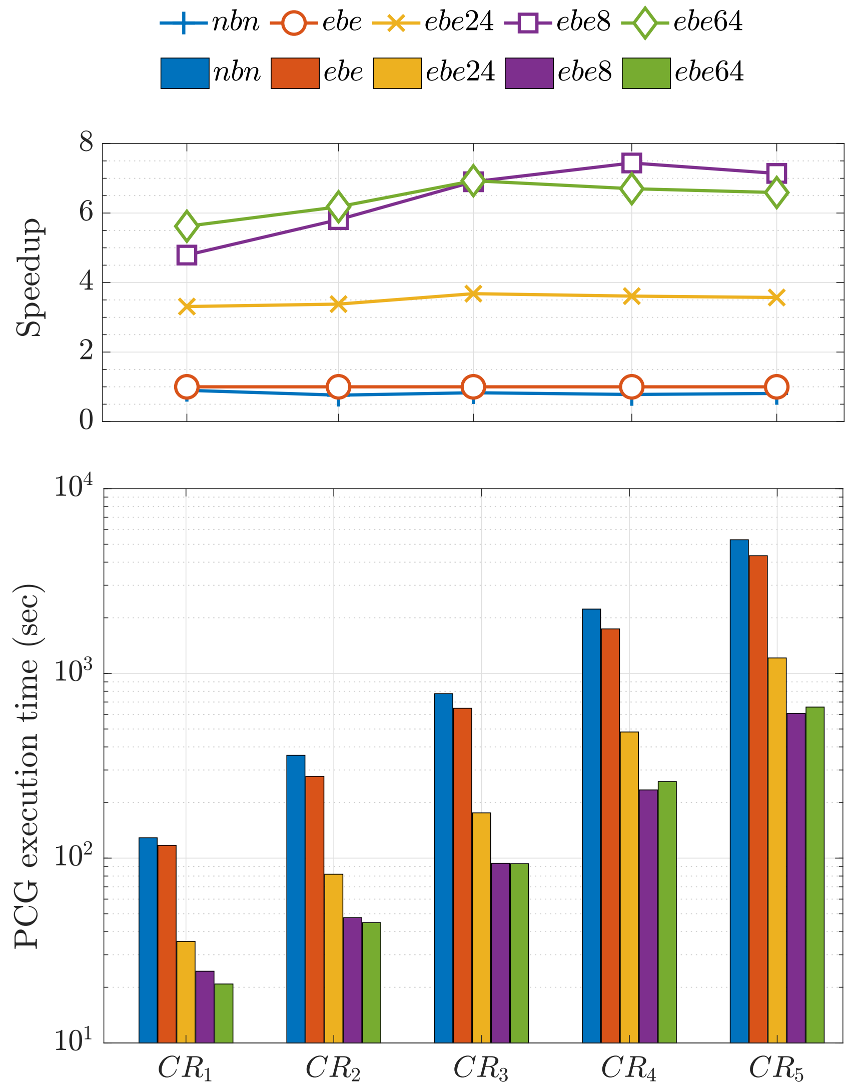


Figure 4.8: PCG execution time, and speedup with respect to the ***ebe*** strategy for Connecting rod example.

TH-3072_146103004

ebe64 strategy for larger unstructured meshes. The ***ebe8*** strategy allocates 8 threads for reading 24×24 entries of $\mathbf{K}_e$ from global memory. However, the ***ebe64*** strategy 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*** strategy.

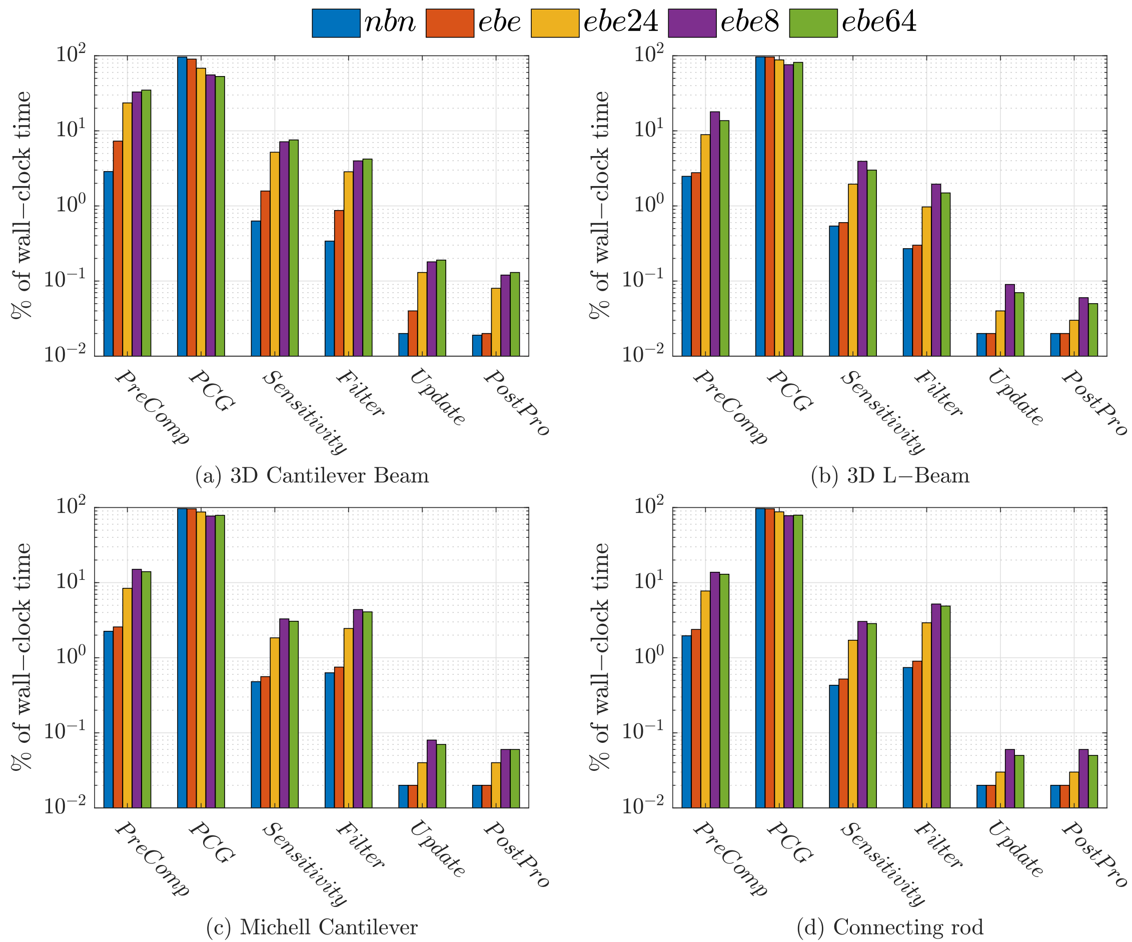


Figure 4.9: Percentage share of various computational steps of topology optimization framework from the wall-clock time, corresponding to the proposed fine-grained SpMV strategies.

Figure 4.9 shows the percentage of total wall-clock time ($\%WC$) attributed to the various computational steps of the topology optimization framework for the largest meshes of all four examples. It can be observed from Figure 4.9 that even when the PCG solver

TH-3072_146103004

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

4.2.1 Closure

The results show that among the proposed GPU-based fine-grained SpMV strategies, the ***ebe*8** strategy was found to be the best performing one. 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. 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. All three proposed SpMV strategies store the connectivity data in shared memory of GPU, thereby reducing number of global memory read operations. To access the elemental stiffness matrices a large number of global memory read operations is still needed. Additionally, during each optimization iteration, the data must be copied from CPU to GPU. Both the challenges degrade the performance of the matrix-free PCG solver. The computational performance of the proposed matrix-free PCG solver can be further improved by reducing the amount of CPU-GPU data transfer, and optimizing the data accesses on GPU.

TH-3072_146103004

TH-3072_146103004

Chapter 5

Novel Data Storage and Data Access Patterns for Matrix-Free FEA Solver

The proposed matrix-free PCG solver for topology optimization requires the elemental stiffness matrices of all finite elements to be explicitly computed and stored. For large-scale unstructured meshes, a collective size of elemental stiffness matrices can be huge. Even if these matrices are generated on CPU and transferred to GPU, this data transfer can consume a substantial amount of time. Moreover, storing these matrices on GPU's global memory requires a lot of space. Furthermore, GPU threads have to perform several read and write operations on global memory of GPU that can slow down the performance of the solver. The final objective of the thesis is to develop efficient data storage and data access patterns on GPU to further enhance the performance of the proposed matrix-free PCG solver for topology optimization.

First, we focus on reducing the amount of data transfer between CPU and GPU. In most of the cases, the elemental stiffness matrices are symmetric. Using this property, SpMV can be performed using only the symmetric half of the elemental stiffness matrices that can significantly reduce the amount of data copied to GPU. We develop a novel GPU-strategy '***ebeSym***' which performs SpMV of the proposed matrix-free PCG solver

TH-3072_146103004

by utilizing only a symmetric half of elemental stiffness matrices. Instead of using 576 entries per element we use only 300 entries. This significantly reduces the amount of data copied to GPU, thereby can allow solving larger-scale examples on GPU. A reduction in the amount of input data for computation on memory bound processor architectures like GPU is expected to improve its performance.

Next, we develop two novel data storage formats that ensure coalesced read and write operations in order to optimize the data accesses on GPU. The entries from the symmetric half of $\mathbf{K}_e$ are divided into 'diagonal' and 'off-diagonal' parts that are stored separately in two arrays. The storage is performed in such a manner that when consecutive threads are assigned to consecutive elements, threads make coalesced access to entries in the respective arrays.

5.1 Exploiting Symmetry of Elemental Stiffness Matrices

The elemental stiffness matrix for an 8-noded hexahedral element with three DoFs per node is shown in Figure 5.1. Each tile in the figure represents a sub-matrix of 3×3 entries, resulting into a total of 64 sub-matrices. In the proposed strategy, entries from the upper triangular half are used for SpMV, henceforth, the matrix is referred to as '$\mathbf{K}_{sym}$'. It can be seen from Figure 5.1 that the sub-matrices can be categorized as diagonal and off-diagonal sub-matrices. These sub-matrices are further grouped for developing the kernel for the proposed strategy. The eight diagonal sub-matrices ($X_1 - X_8$) form one group (X_i), while the off-diagonal sub-matrices are grouped into seven groups ($A_i, \ldots, G_i$), each containing four sub-matrices. For example, the off-diagonal group 'A_i' consists of four sub-matrices A_1, A_2, A_3, and A_4 that can be seen in Figure 5.1. In the following sections, the customized data storage format and the proposed SpMV strategy are discussed.

5.2 Customized Data Storage Format

The aim of developing a customized data storage format is to store symmetric part of stiffness matrices for coalesced read and write operations on GPU. The entries of diagonal

TH-3072_146103004

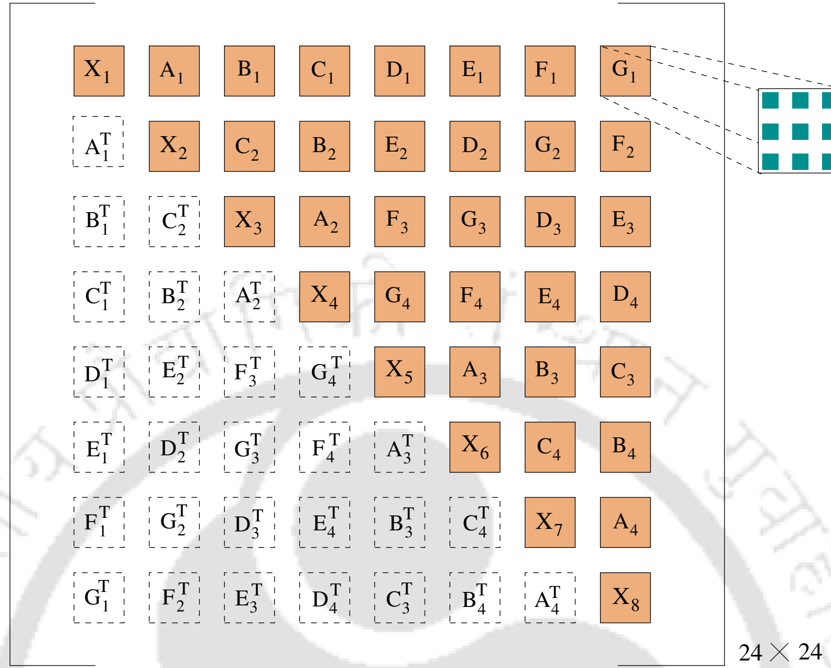


Figure 5.1: A symmetric elemental stiffness matrix is shown. The entries from the upper triangular half ($\mathbf{K}_{sym}$) are used for SpMV.

and off-diagonal groups are stored separately in vectors '$\boldsymbol{\kappa_d}$' and '$\boldsymbol{\kappa_{od}}$', respectively.

5.2.1 Storage Format for Diagonal Sub-Matrices

The storage pattern used to store the entries of the diagonal sub-matrices into vector $\boldsymbol{\kappa_d}$ is shown in Figure 5.2. At the top of the figure, all diagonal sub-matrices of an elemental stiffness matrix are shown with their entries. The vector $\boldsymbol{\kappa_d}$ is generated by storing entries of the first element (E_1) followed by other elements ($E_2, \ldots, E_{Nelem}$). For the first element (E_1), the left most entries from sub-matrices X_1 to X_8 such as ($a^j, \forall\ j \in 1, \ldots, 8$), etc. are stored. The same entries are then copied from the second element (E_2). The same procedure is followed for the rest of the elements in the mesh. Thereafter, entries b^j to f^j are copied one-by-one in a similar way as a^j entries are copied. It is noted that the vector $\boldsymbol{\kappa_d}$ is a one-dimensional vector as shown in Figure 5.2. When consecutive threads are assigned to consecutive elements, threads make coalesced access to entries in diagonal groups by accessing consecutive memory locations.

TH-3072_146103004

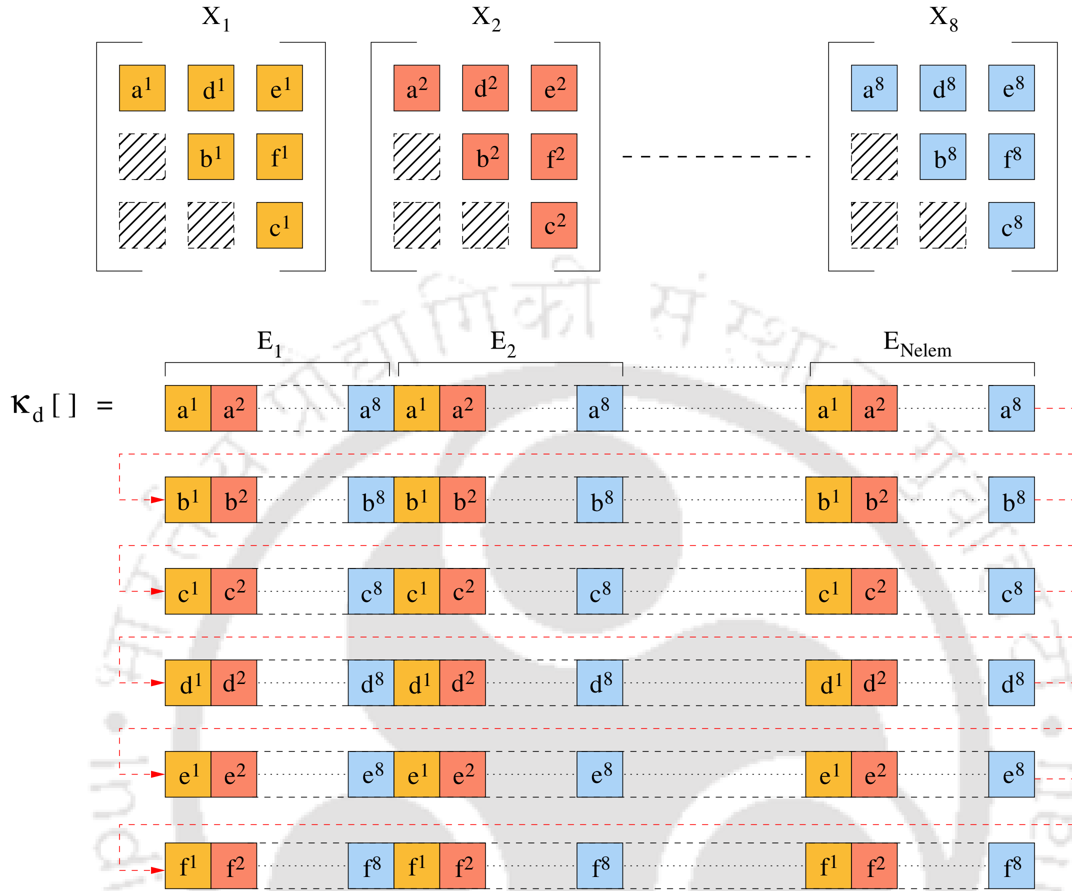


Figure 5.2: Storage format for the entries of diagonal sub-matrices.

5.2.2 Storage Format for Off-Diagonal Sub-Matrices

The off-diagonal matrices are stored in the '$\boldsymbol{\kappa_{od}}$' vector. The pattern of storage is shown in Figure 5.3 in which diagonal entries ($a^k, b^k, c^k, \forall\ k \in 1, \ldots, 4$) are stored first, followed by off-diagonal entries (d^k to g^k). It can been that the diagonal entry ($a^k, \forall\ k \in 1, \ldots, 4$) is stored for the first element (E_1), followed by other elements. The same procedure is followed for other diagonal elements (b^k and c^k). Thereafter, the off-diagonal entries are stored in the '$\boldsymbol{\kappa_{od}}$' vector as a 2-tuple. As shown in Figure 5.3, a 2-tuple of ($d^k, e^k, \forall\ k = 1, \ldots, 4$) is copied for the first element followed by other elements. The other 2-tuples that are copied in a sequence are (f^k, g^k) and (h^k, i^k). These 2-tuples are copied element-by-element. Since the corresponding entries of the off-diagonal group are stored in consecutive memory locations, a simultaneous reading of values for all elements

TH-3072_146103004

results in coalesced memory access.

Figure 5.3: Storage format for the entries of off-diagonal sub-matrices.

Using the symmetry property of $\mathbf{K}_e$ and the data storage patterns discussed in Section 5.2.1 and Section 5.2.2 a symmetry-based SpMV strategy is developed which is discussed in the following section.

5.3 Proposed Symmetry-based SpMV Strategy '*ebeSym*'

The GPU kernel of the proposed symmetry-based SpMV Strategy '***ebeSym***' is discussed here. The proposed strategy uses a greedy graph coloring scheme for handling race condition [51]. The elements are categorized into different colors during pre-processing, and thereafter, the kernel is launched color-wise. In one kernel launch, the matrix-vector multiplication for the elements of a single color is performed. The ***ebeSym***

TH-3072_146103004

strategy allocates eight compute threads to each element in the mesh because the strategy is developed for 8−noded hexahedral elements. The eight threads first multiply the entries of the diagonal group 'X_i' with the corresponding entries of the vector '$\boldsymbol{u}_e$'. Thereafter, the same operation is performed by each thread for an off-diagonal group. The multiplication performed for one of off-diagonal groups is shown in Figure 5.4. For example, the four sub-matrices 'D_1'-'D_4' of group (D_i) are allocated to compute threads 't_1'-'t_4', and their corresponding missing parts are allocated to threads 't_5'-'t_8'. Instead of reading the same sub-matrix from global memory twice, a broadcast operation is used, and the sub-matrix is read by two threads at the same time. For example, the sub-matrix 'D_2' is positioned at the 2nd row and 6th column in Figure 5.4. As a result, threads 't_2' and 't_6' are allocated to it. Now, 't_2' multiplies 'D_2' with the corresponding entries from '$\boldsymbol{u}_e$', whereas 't_6' performs transpose of 'D_2' and multiplies its entries with the corresponding entries from '$\boldsymbol{u}_e$'. The same procedure is followed for every off-diagonal sub-matrix.

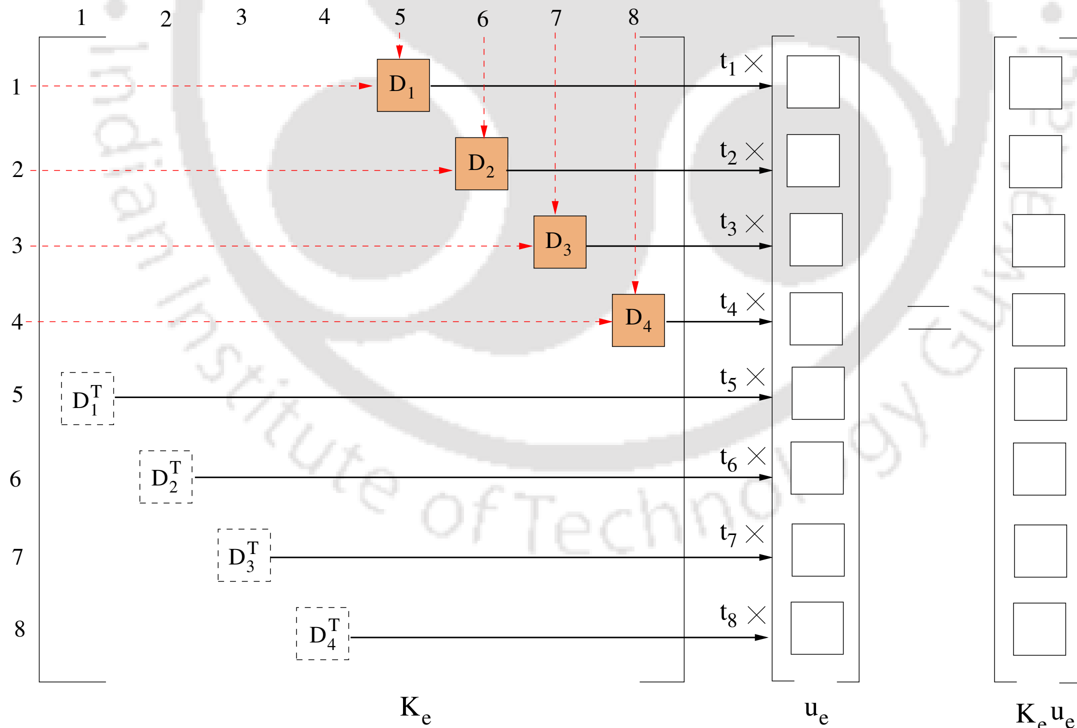


Figure 5.4: Thread assignment and matrix-vector multiplication by the proposed ***ebeSym*** strategy for one group of sub-matrices.

TH-3072_146103004

The $\boldsymbol{ebeSym}$ strategy is represented in Algorithm 7. The data required are vector '$\boldsymbol{\kappa_d}$', vector '$\boldsymbol{\kappa_{od}}$', elemental density vector '$\boldsymbol{\rho}$', displacement vector '$\boldsymbol{u}$', connectivity matrix '$\mathbf{C}$', total number of elements in mesh 'N_{elem}', number of nodes per element 'npe', number of off- diagonal groups 'N_{od}', and DoF per node 'dof'. The output is stored in the vector '$\boldsymbol{r}$'.

Algorithm 7: Proposed symmetry-based SpMV strategy '***ebeSym***.'

```
    Data: κ_d, κ_od, ρ, u, C, N_elem, npe, N_od, dof
    Output: r
 1  t = blockIdx.x × blockDim.x + threadIdx.x;            // global thread index
 2  if t < (N_elem * 8) then
 3      global_dof ← C;                 // index of product vector to write result
 4      __shared__ s_u ← u;                               // multiplying vector
 5      ρ_e = ρ[e]^p;              // penalized elemental density of this element
 6      __syncthreads( );                            // synchronization barrier
        /* for diagonal group */
 7      grp_id;                                // group index of this sub-matrix
 8      for i ← 1 to dof do
 9          idx_1 ← comp_stiffness_index(t, grp_id, npe);          // index of κ_d
10          idx_2 ← comp_vector_index(t, grp_id, npe);             // index of s_u
11          val[i] += ρ_e × κ_d[idx_1] × s_u[idx_2];  // matrix-vector multiplication
        /* for off-diagonal group */
12      for i ← 1 to N_od do
13          grp_id;                            // group index of this sub-matrix
14          for j ← 1 to dof do
15              idx_1 ← comp_stiffness_index(t, grp_id, npe);     // index of κ_od
16              idx_2 ← comp_vector_index(t, grp_id, npe);         // index of s_u
17              val[j] += ρ_e × κ_od[idx_1] × s_u[idx_2];        // matrix-vector
                 multiplication
18      r[global_dof]     += val[0];        // writing results in output vector
19      r[global_dof + 1] += val[1];
20      r[global_dof + 2] += val[2];
21  end
22  __syncthreads( );                                // synchronization barrier
```

The global thread index 't' is computed in line 1. Thereafter, the index '$global_dof$' is read from the connectivity matrix, which represents the location in the output vector for writing the results by thread 't'. The '$\boldsymbol{u}$' entries corresponding to the elements in

TH-3072_146103004

the thread block are copied from global memory to shared memory '$\boldsymbol{s_u}$' in line 4. Line 5 retrieves and penalizes the corresponding elemental density. Next, the matrix-vector multiplication for the diagonal group is performed. Line 8 loops over the number of DoFs per node. Lines 9 and 10 compute the indices 'idx_1' and 'idx_2', respectively. The index 'idx_1' is the multiplying position in '$\boldsymbol{\kappa_d}$', whereas the index 'idx_2' is the multiplying position in '$\boldsymbol{s_u}$'. Using these indices, the multiplication is performed in line 11. Thereafter, the multiplications for the off-diagonal groups are performed. Since there are multiple off-diagonal groups such as $A_i, \ldots, G_i$, line 12 loops over the number of off-diagonal groups 'N_{od}'. Inside this loop, there is another loop over DoFs in line 14. Indices 'idx_1' and 'idx_2' are computed in lines 15 and 16, where 'idx_1' is the multiplying position in '$\boldsymbol{\kappa_{od}}$' and 'idx_2' is the multiplying position in '$\boldsymbol{s_u}$'. The multiplications of the respective entries of '$\boldsymbol{\kappa_{od}}$' and '$\boldsymbol{s_u}$' are performed in line 17. The same procedure is followed for all off-diagonal groups. Finally, lines 18 - 20 write the results of the multiplications into the output vector '$\boldsymbol{r}$' for all three DoFs.

Exploiting the symmetry property of elemental matrices for SpMV operation is an idea which already exists in the literature. It was earlier used by Zegard and Paulino [51] but for 2D meshes and used a Cholesky-based FEA solver. The symmetry idea was also explored by Duarte et al. [52] for polygonal meshes, and by Kiran et al. [55] for FEA of 2D unstructured meshes. The strategy proposed in this thesis is more general in terms of its applicability to unstructured finite elements. The proposed ***ebeSym*** strategy addresses the implementational challenge associated with 3D finite elements and presents data structure which can efficiently handle large elemental matrices related with 3D elements. The proposed ***ebeSym*** strategy also has no inter-thread dependency.

5.4 Computational Performance

The proposed ***ebeSym*** strategy is tested using the benchmark examples discussed in Section 3.3, and mesh sizes given in Tables 3.2-3.5.

Figure 5.5 shows the PCG execution time and the corresponding speedups for the given benchmark examples. The speedups for the ***ebeSym*** strategy are computed with respect to the base strategy '***ebe***', and also the 8-thread per element strategy '***ebe8***'. Both ***ebe*** and ***ebe8*** strategies use full $\mathbf{K}_e$. The left Y-axis shows the execution time

TH-3072_146103004

consumed by the PCG solver in one optimization iteration for the ***ebeSym***, ***ebe8***, and ***ebe*** strategies. The speedup gained by the ***ebeSym*** strategy over the ***ebe***, and ***ebe8*** strategies are shown in the right Y-axis of the same plots.

It can be observed that for all mesh sizes of the four benchmark problems, the ***ebeSym*** strategy requires much lesser time than both ***ebe*** and ***ebe8*** strategy. For the smallest mesh CB_1 of 3D cantilever beam example, ***ebeSym*** strategy shows speedups of 2.1× and 10.9× against ***ebe8*** and ***ebe*** strategies respectively. For the largest mesh CB_5 the achieved speedups are 3.6× and 26.7×. For 3D L-beam example the speedups of 2.2× and 11× are observed for the smallest mesh LB_1, whereas for the largest mesh LB_5 speedups of 3.6× and 26.4× are noted. A similar trend is noted for the Michell cantilever example where maximum speedups of 3.3× and 24× are achieved for mesh MC_5 corresponding to ***ebe8*** and ***ebe*** strategies, respectively. For connecting rod example the maximum speedups of 3.1× and 22.2× are observed for the largest mesh CR_5. It can be seen from the results that the proposed ***ebeSym*** strategy shows good scalability with the increment in the problem size.

The execution time of all computational steps of topology optimization for ***ebeSym*** strategy are now analyzed. The execution time of computational steps are measured, and their % contributions are calculated with respect to the wall-clock time as shown in Figure 5.6. In the same plot the % of wall-clock time of the various computational steps corresponding to the symmetry-based ***ebeSym*** strategy are compared against the 8-threads per element strategy '***ebe8***' which uses full $\mathbf{K}_e$. It can be observed from Figure 5.6 that '*PCG*' step of the ***ebeSym*** strategy requires between 67% and 97% of the wall-clock time even when it is performed on GPU. However, the ***ebe8*** strategy requires even more time than the proposed ***ebeSym*** strategy. Next, the '*Filter*' step consumes 2% to 26% of the wall-clock time. It is found that this step requires less time for smaller mesh size. However, the computation time increases significantly with large mesh sizes. The '*Sensitivity*' step then consumes between 1% and 5% of the wall-clock time that is quite less with respect to the previous steps. The percentage contributions of other steps such as '*PreComp*', '*Update*', and '*PostProcess*' are quite less. It can be observed from the computational time of these steps that the solver is still the computational bottleneck of topology optimization. Another observation is that since the ***ebeSym*** strategy requires less time for the solver, the total time of optimization gets reduced. This reduction increases percentage contribution of the '*Filter*' step. However, when

TH-3072_146103004

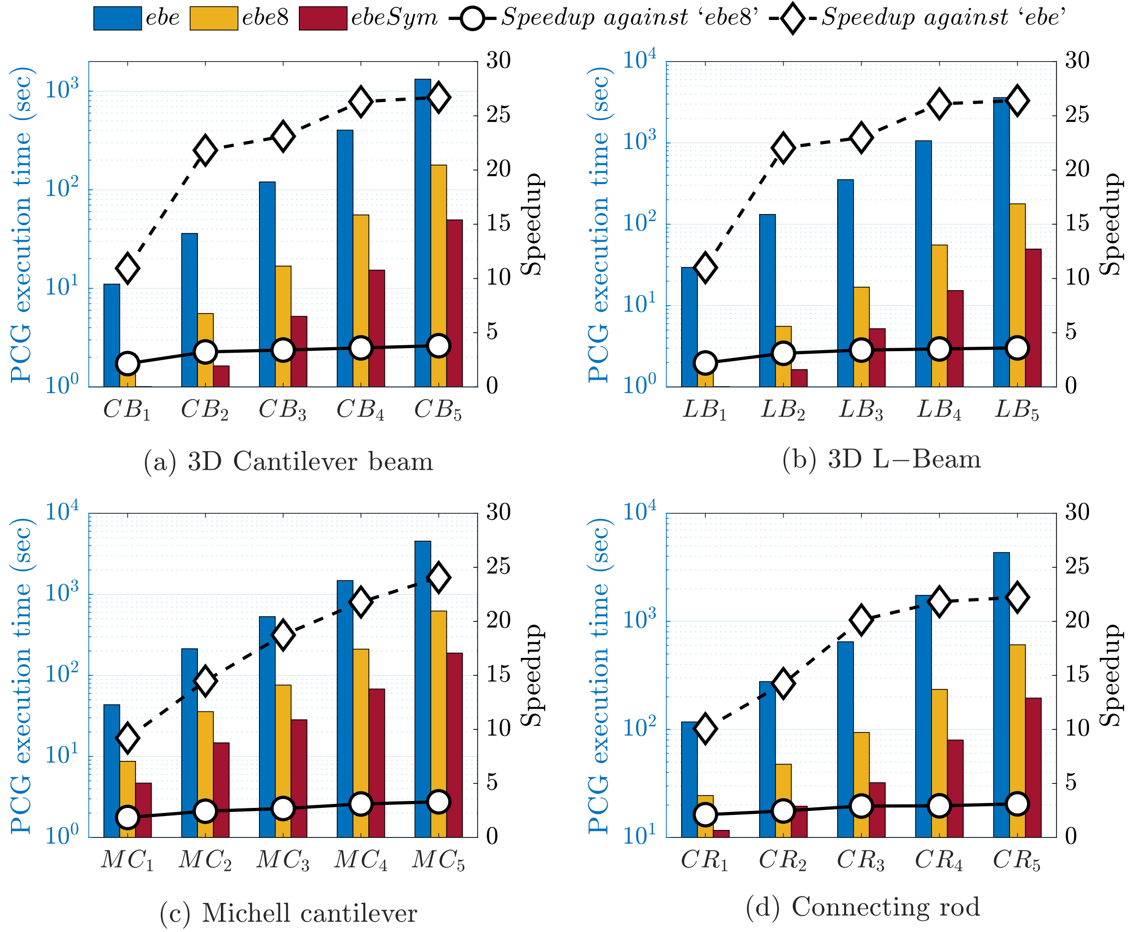


Figure 5.5: PCG execution time and speedup of the ***ebeSym*** strategy against ***ebe*** and ***ebe8*** strategies for various mesh sizes of four benchmark examples.

the ***ebe8*** strategy consumes 97% of the wall-clock time, the percentage for the '*Filter*' step reduces to 2%.

Figure 5.7 shows the amount of GPU memory required by the both ***ebe8*** and ***ebeSym*** strategies for each mesh size. It can be seen that for smaller mesh sizes of the four examples, both strategies require almost similar amounts of memory. However, the requirement of memory keeps on increasing with increment in mesh size. For example, the ***ebe8*** strategy requires 2500 MB of GPU memory for mesh CB_5 of 3D cantilever beam example whereas, the ***ebeSym*** strategy requires 1400 MB, thereby reducing the memory requirement by 1.8×. Similarly, for the largest mesh sizes of other

TH-3072_146103004

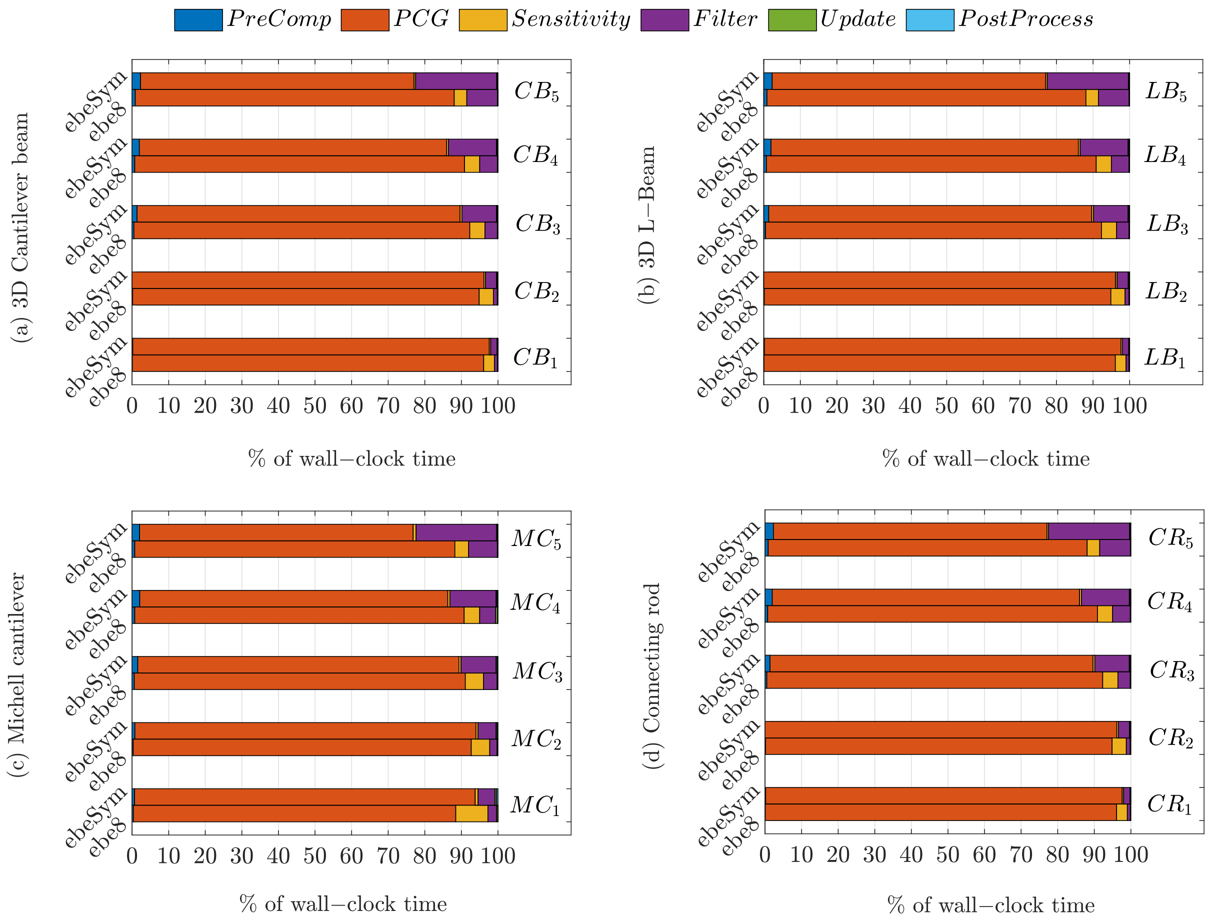


Figure 5.6: Percentage contributions of various computational steps of topology optimization with respect to the wall-clock time.

Institute of Technology Guwahati

three examples, the proposed ***ebeSym*** strategy requires 1.8×, 1.75×, and 1.78× lesser memory compared to the ***ebe8*** strategy for 3D L-beam, Michell cantilever, and connecting rod examples, respectively. Since the memory requirement is less with the proposed ***ebeSym*** strategy, topology optimization problems with much larger mesh sizes can be solved.

TH-3072_146103004

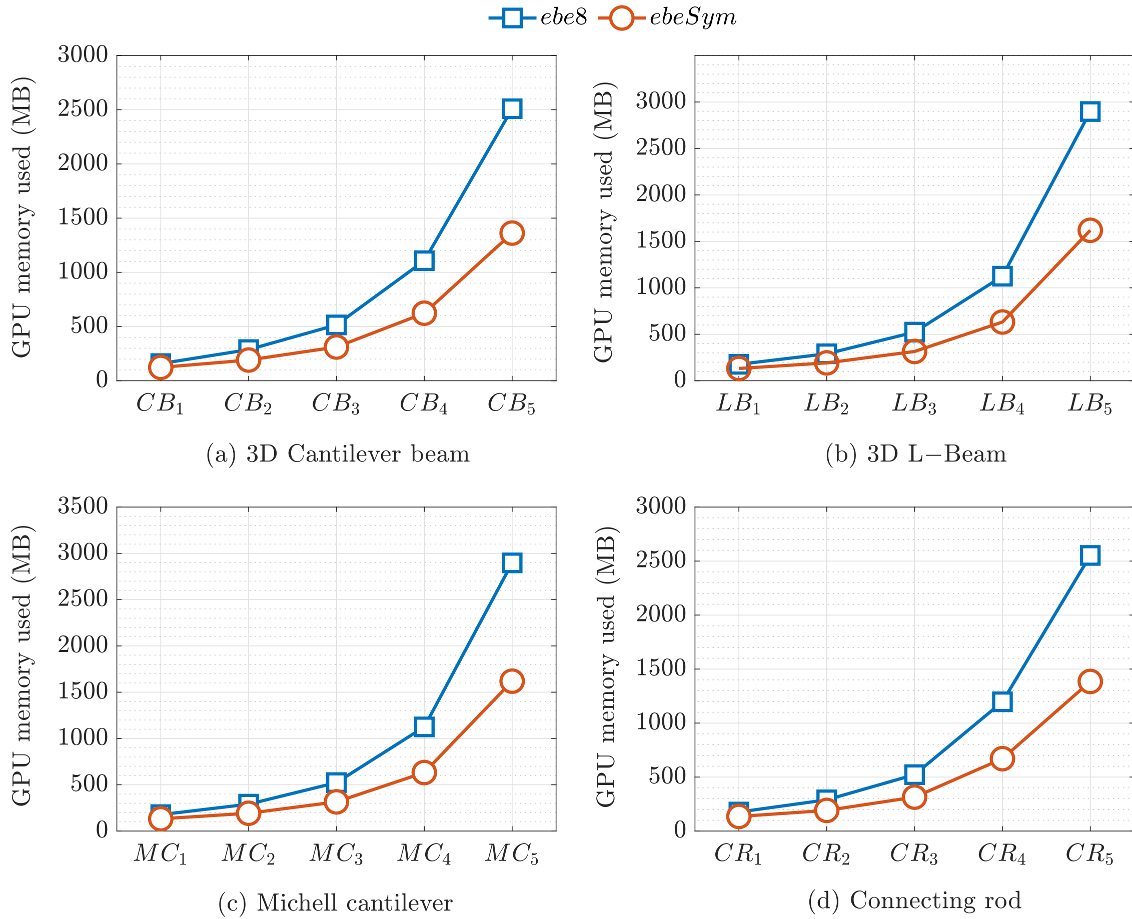


Figure 5.7: GPU memory required by the ***ebe8*** and ***ebeSym*** strategies for different mesh sizes.

5.5 Closure

The analysis of computational performance of the proposed ***ebeSym*** strategy shows that the idea of exploiting the symmetry of elemental stiffness matrices significantly reduces the amount of CPU-GPU data transfer. This is evident by comparing the amount of GPU memory requirement of ***ebeSym*** strategy and ***ebe8*** strategy, which reveals that the GPU memory requirement of the ***ebeSym*** strategy is close to half in comparison to the ***ebe8*** strategy. The proposed novel data storage formats enable threads to read the elemental data in coalesced manner. The combination of ***ebe8*** thread allocation

TH-3072_146103004

strategy with the novel data storage patterns to leverage symmetry results in a better computational performance of the proposed matrix-free PCG solver which can be observed in the analysis of PCG execution time and speedups. Using the symmetry-based SpMV strategy '***ebeSym***', the maximum speedups of 26.7×, 26.4×, 26×, and 22.2× were observed against the one-thread per element SpMV strategy '***ebe***' which uses full $\mathbf{K}_e$.

TH-3072_146103004

TH-3072_146103004

Chapter 6

Conclusions and Future Work

In this thesis, an efficient GPU-based matrix-free FEA solver is developed for topology optimization of large-scale 3D continuum structures, discretized using unstructured meshes. With the aim of accelerating the topology optimization, the entire FEA solver is executed on the GPU. Through a literature survey and by initial analysis, the computational bottleneck of the FEA solver is identified. Next, efficient thread allocation strategies are developed in order to optimize the computational load distribution among the GPU threads. Furthermore, novel data storage and data access patterns are developed in order to reduce the amount of CPU-GPU data transfer and to optimize the memory transactions on the GPU. The combination of these novel GPU computing strategies accelerated the performance of the proposed FEA solver by many-folds.

6.1 Conclusions

Following are the key conclusions of this thesis.

- A GPU-based FEA solver tailored to efficiently handle the unstructured meshes was developed. Two strategies were developed to perform matrix-free SpMV on GPU; node-by-node strategy (***nbn***), and element-by-element strategy (***ebe***). A customized nodal connectivity storage format was also developed for ***nbn***. Initial results show that both ***ebe*** and ***nbn*** strategies are 4× faster compared to their

TH-3072_146103004

CPU-based versions. However, overall computation time required by ***nbn*** is much higher compared to ***ebe***. From the results it can be concluded that ***ebe*** is a more suitable SpMV strategy for large unstructured meshes.

- The comparison of wall-clock time of the computational steps conclude that FEA solver is the computational bottleneck of topology optimization framework. When ***ebe*** is used, the solver consumes between 92% and 95% of the total wall-clock time. Whereas, for ***nbn***, the percentage share of FEA solver is from 95% to 99%.

- The ***ebe*** strategy has been further improved by distributing the computational load among more number of threads. Three fine-grained thread allocation strategies have been proposed in this thesis to efficiently perform SpMV on GPU.

  - ***ebe*8**: allocates 8-compute threads per finite element, 8.5× faster compared to the GPU-based ***ebe*** strategy.
  - ***ebe*24**: allocates 24-threads per element, shows a maximum speedup of 4.5× over ***ebe*** strategy.
  - ***ebe*64**: allocates 64-threads per element, shows a maximum speedup of 8.2× over ***ebe*** strategy.

  From the results it is concluded that for smaller meshes ***ebe*64** performs better, but for larger unstructured meshes the performance of ***ebe*8** is better. Furthermore, higher number of threads does not always guarantee a higher speedup. As observed with the ***ebe*24** strategy, inefficient data management on the GPU may affect the performance.

- Since results conclude that ***ebe*8** is the best performing thread allocation strategy, its performance is further improved by optimizing the CPU-GPU data transfer and data transactions on GPU. By leveraging the symmetry property of elemental stiffness matrices, the amount of data transfer between CPU and GPU is reduced by a large margin. Additionally, two novel data storage formats have been developed which enables the coalescence in GPU memory transactions. The proposed symmetry-based SpMV strategy is named '***ebeSym***'. The performance analysis shows that ***ebeSym*** is 3.6× faster than ***ebe*8** strategy, and 26.7× faster compared to the ***ebe*** strategy. Results also show that the GPU memory requirement

TH-3072_146103004

of ***ebeSym*** is 1.8× lesser than ***ebe8*** strategy. For the largest problem size taken in this thesis, ***ebe8*** strategy requires 2500 MB of GPU memory, whereas ***ebeSym*** requires 1400 MB.

- The GPU-based matrix-free FEA solver proposed in this thesis performed efficiently over the benchmark problems taken from the literature. The GPU framework developed using the proposed FEA solver was able to obtain the correct topologies, including for the problems represented by complex domain geometries and discretized using unstructured meshes. The proposed fine-grained thread allocation strategies explore different levels of granularity in SpMV operations. The symmetry-based SpMV strategy minimizes the amount of data movement, enabling us to solve a larger problem on the same GPU device. The novel data storage formats maintain the coalescence of data transactions, thereby reducing the latency. Overall, the work presented in this thesis makes a significant contribution to the research domain of GPU-based acceleration of structural topology optimization using large 3D unstructured meshes.

6.2 Future Work

- The GPU computing strategies proposed in this thesis are implemented on a single GPU. In the future, these strategies can be further extended to be implemented on multi-GPU systems. A multi-GPU implementation of the proposed strategies will allow the solution of significantly larger-scale problems.

- The GPU computing strategies proposed in this thesis are developed for 8-noded hexahedral elements. In the future, these strategies can be modified for other types of finite elements.

- The GPU-based PCG solver proposed in this thesis uses diagonal preconditioner. In the future, a more advanced preconditioner such as multigrid or domain decomposition can be used to further improve convergence of the solver.

TH-3072_146103004

TH-3072_146103004

References

[1] Alok Sutradhar, Glaucio H Paulino, Michael J Miller, and Tam H Nguyen. Topological optimization for designing patient-specific large craniofacial segmental bone replacements. *Proceedings of the National Academy of Sciences*, 107(30):13222–13227, 2010.

[2] David J Munk and Jonathan D Miller. Topology optimization of aircraft components for increased sustainability. *AIAA Journal*, 60(1):1–16, 2021.

[3] Matthew Tomlin and Jonathan Meyer. Topology optimization of an additive layer manufactured (ALM) aerospace part. In *Proceeding of the 7th Altair CAE technology conference*, pages 1–9, 2011.

[4] Ji-Hong Zhu, Wei-Hong Zhang, and Liang Xia. Topology optimization in aircraft and aerospace structures design. *Archives of Computational Methods in Engineering*, 23(4):595–622, 2016.

[5] Guan Zhou, Guangyao Li, Aiguo Cheng, Guochun Wang, Hongmin Zhang, and Yi Liao. The lightweight of auto body based on topology optimization and sensitivity analysis. Technical report, SAE Technical Paper, 2015.

[6] Suh In Kim, Seok Won Kang, Yong-Sub Yi, Joonhong Park, and Yoon Young Kim. Topology optimization of vehicle rear suspension mechanisms. *International Journal for Numerical Methods in Engineering*, 113(8):1412–1433, 2018.

[7] Pedro G Coelho, João B Cardoso, Paulo R Fernandes, and Hélder C Rodrigues. Parallel computing techniques applied to the simultaneous design of structure and material. *Advances in Engineering Software*, 42(5):219–227, 2011.

TH-3072_146103004

[8] Yuriy Elesin, Boyan Stefanov Lazarov, Jakob Søndergaard Jensen, and Ole Sigmund. Time domain topology optimization of 3D nanophotonic devices. *Photonics and Nanostructures-Fundamentals and Applications*, 12(1):23–33, 2014.

[9] Francisco Javier Ramírez-Gil, Emilío Carlos Nelli Silva, and Wilfredo Montealegre-Rubio. Topology optimization design of 3D electrothermomechanical actuators by using GPU as a co-processor. *Computer Methods in Applied Mechanics and Engineering*, 302:44–69, 2016.

[10] Deepak Sharma, Kalyanmoy Deb, and NN Kishore. Domain-specific initial population strategy for compliant mechanisms using customized genetic algorithm. *Structural and Multidisciplinary Optimization*, 43(4):541–554, 2011.

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

[12] Deepak Sharma, Kalyanmoy Deb, and NN Kishore. Customized evolutionary optimization procedure for generating minimum weight compliant mechanisms. *Engineering Optimization*, 46(1):39–60, 2014.

[13] Tomás Zegard and Glaucio H Paulino. Bridging topology optimization and additive manufacturing. *Structural and Multidisciplinary Optimization*, 53:175–192, 2016.

[14] ZHU Jihong, ZHOU Han, WANG Chuang, ZHOU Lu, YUAN Shangqin, and Weihong Zhang. A review of topology optimization for additive manufacturing: Status and challenges. *Chinese Journal of Aeronautics*, 34(1):91–110, 2021.

[15] Martin Philip Bendsøe and Noboru Kikuchi. Generating optimal topologies in structural design using a homogenization method. *Computer methods in applied mechanics and engineering*, 71(2):197–224, 1988.

[16] Martin P Bendsøe. Optimal shape design as a material distribution problem. *Structural optimization*, 1(4):193–202, 1989.

[17] M Zhou and GIN Rozvany. The COC algorithm, part ii: Topological, geometrical and generalized shape optimization. *Computer methods in applied mechanics and engineering*, 89(1-3):309–336, 1991.

TH-3072_146103004

[18] Laura Berrocal, Rosario Fernández, Sergio González, Antonio Periñán, Santos Tudela, Jorge Vilanova, Luis Rubio, Jose Manuel Martín Márquez, Javier Guerrero, and Fernando Lasagni. Topology optimization and additive manufacturing for aerospace components. *Progress in Additive Manufacturing*, 4:83–95, 2019.

[19] ALR Prathyusha and G Raghu Babu. A review on additive manufacturing and topology optimization process for weight reduction studies in various industrial applications. *Materials Today: Proceedings*, 62:109–117, 2022.

[20] Blaise Bourdin and Antonin Chambolle. Design-dependent loads in topology optimization. *ESAIM: Control, Optimisation and Calculus of Variations*, 9:19–48, 2003.

[21] Blaise Bourdin and Antonin Chambolle. The phase-field method in optimal design. In Martin Philip Bendsøe, Niels Olhoff, and Ole Sigmund, editors, *IUTAM Symposium on Topological Design Optimization of Structures, Machines and Materials*, pages 207–215, Dordrecht, 2006. Springer Netherlands.

[22] Michael Yu Wang, Xiaoming Wang, and Dongming Guo. A level set method for structural topology optimization. *Computer methods in applied mechanics and engineering*, 192(1-2):227–246, 2003.

[23] Grégoire Allaire, François Jouve, and Anca-Maria Toader. A level-set method for shape optimization. *Comptes Rendus Mathematique*, 334(12):1125–1130, 2002.

[24] Grégoire Allaire, François Jouve, and Anca-Maria Toader. Structural optimization using sensitivity analysis and a level-set method. *Journal of computational physics*, 194(1):363–393, 2004.

[25] Takayuki Yamada, Kazuhiro Izui, Shinji Nishiwaki, and Akihiro Takezawa. A topology optimization method based on the level set method incorporating a fictitious interface energy. *Computer Methods in Applied Mechanics and Engineering*, 199(45-48):2876–2891, 2010.

[26] Hao Li, Zhen Luo, Liang Gao, and Qinghua Qin. Topology optimization for concurrent design of structures with multi-patch microstructures by level sets. *Computer Methods in Applied Mechanics and Engineering*, 331:536–561, 2018.

TH-3072_146103004

[27] Hao Li, Zhen Luo, Nong Zhang, Liang Gao, and Terry Brown. Integrated design of cellular composites using a level-set topology optimization method. *Computer Methods in Applied Mechanics and Engineering*, 309:453–475, 2016.

[28] Hans A Eschenauer, Vladimir V Kobelev, and Axel Schumacher. Bubble method for topology and shape optimization of structures. *Structural optimization*, 8(1):42–51, 1994.

[29] Yi Min Xie and Grant P Steven. A simple evolutionary procedure for structural optimization. *Computers & structures*, 49(5):885–896, 1993.

[30] Xiao Y Yang, Yi M Xie, Grant P Steven, and Osvaldo M Querin. Bi-directional evolutionary structural optimization. In *Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization (St. Louis)*, pages 1449–1457, 1998.

[31] Osvaldo M Querin, Grant P Steven, and Yi Min Xie. Evolutionary structural optimisation (ESO) using a bi-directional algorithm. *Engineering computations*, 15(8):1031–1048, 1998.

[32] Rubén Ansola, Estrella Veguería, Javier Canales, and José A Tárrago. A simple evolutionary topology optimization procedure for compliant mechanism design. *Finite Elements in Analysis and Design*, 44(1-2):53–62, 2007.

[33] Ole Sigmund. A 99-line topology optimization code written in matlab. *Structural and multidisciplinary optimization*, 21(2):120–127, 2001.

[34] Krister Svanberg. The method of moving asymptotes—a new method for structural optimization. *International journal for numerical methods in engineering*, 24(2):359–373, 1987.

[35] Joshua D Deaton and Ramana V Grandhi. A survey of structural and multidisciplinary continuum topology optimization: post 2000. *Structural and Multidisciplinary Optimization*, 49(1):1–38, 2014.

[36] Cris Cecka, Adrian J Lew, and Eric Darve. Assembly of finite element methods on graphics processors. *International journal for numerical methods in engineering*, 85(5):640–669, 2011.

TH-3072_146103004

[37] Serban Georgescu, Peter Chow, and Hiroshi Okuda. GPU acceleration for FEM-based structural analysis. *Archives of Computational Methods in Engineering*, 20(2):111–121, 2013.

[38] James K Guest, Jean H Prévost, and Ted Belytschko. Achieving minimum length scale in topology optimization using nodal design variables and projection functions. *International journal for numerical methods in engineering*, 61(2):238–254, 2004.

[39] Ole Sigmund and Kurt Maute. Topology optimization approaches: A comparative review. *Structural and Multidisciplinary Optimization*, 48(6):1031–1055, 2013.

[40] Boyan Stefanov Lazarov and Ole Sigmund. Filters in topology optimization based on Helmholtz-type differential equations. *International Journal for Numerical Methods in Engineering*, 86(6):765–781, 2011.

[41] Ole Sigmund and Joakim Petersson. Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. *Structural optimization*, 16(1):68–75, 1998.

[42] Thomas Borrvall and Joakim Petersson. Large-scale topology optimization in 3D using parallel computing. *Computer methods in applied mechanics and engineering*, 190(46-47):6201–6229, 2001.

[43] Kumar Vemaganti and W Eric Lawrence. Parallel methods for optimality criteria-based topology optimization. *Computer Methods in Applied Mechanics and Engineering*, 194(34-35):3637–3667, 2005.

[44] A Mahdavi, R Balaji, M Frecker, and EM Mockensturm. Topology optimization of 2D continua for minimum compliance using parallel computing. *Structural and Multidisciplinary Optimization*, 32(2):121–132, 2006.

[45] Jesús Martínez-Frutos, Pedro J Martínez-Castejón, and David Herrero-Pérez. Efficient topology optimization using GPU computing with multilevel granularity. *Advances in Engineering Software*, 106:47–62, 2017.

[46] Jesús Martínez-Frutos and David Herrero-Pérez. Large-scale robust topology optimization using multi-GPU systems. *Computer Methods in Applied Mechanics and Engineering*, 311:393–414, 2016.

TH-3072_146103004

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

[48] Olek C Zienkiewicz, Robert Leroy Taylor, and Jian Z Zhu. *The finite element method: its basis and fundamentals*. Elsevier, 2005.

[49] Eddie Wadbro and Martin Berggren. Megapixel topology optimization on a graphics processing unit. *SIAM review*, 51(4):707–721, 2009.

[50] Stephan Schmidt and Volker Schulz. A 2589 line topology optimization code written for the graphics card. *Computing and Visualization in Science*, 14(6):249–256, 2011.

[51] Tomás Zegard and Glaucio H Paulino. Toward GPU accelerated topology optimization on unstructured meshes. *Structural and multidisciplinary optimization*, 48(3):473–485, 2013.

[52] Leonardo S Duarte, Waldemar Celes, Anderson Pereira, Ivan FM Menezes, and Glaucio H Paulino. Polytop++: an efficient alternative for serial and parallel topology optimization on CPUs & GPUs. *Structural and Multidisciplinary Optimization*, 52(5):845–859, 2015.

[53] Subhajit Sanfui and Deepak Sharma. 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*, pages 1–9. MNNIT Allahabad, India., 2017.

[54] Utpal Kiran, Deepak Sharma, and Sachin Singh Gautam. GPU-warp based finite element matrices generation and assembly using coloring method. *Journal of Computational Design and Engineering*, 6(4):705–718, 2019.

[55] Utpal Kiran, Sachin Singh Gautam, and Deepak Sharma. GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices. *Computing*, 102(9):1941–1965, 2020.

[56] Joakim Petersson and Ole Sigmund. Slope constrained topology optimization. *International Journal for Numerical Methods in Engineering*, 41(8):1417–1434, 1998.

TH-3072_146103004

[57] Martin Philip Bendsøe and Ole Sigmund. *Topology optimization: theory, methods, and applications*. Springer Science & Business Media, Berlin,Germany, 2003.

[58] George IN Rozvany, Ming Zhou, and Torben Birker. Generalized shape optimization without homogenization. *Structural optimization*, 4:250–252, 1992.

[59] Ole Sigmund. On the design of compliant mechanisms using topology optimization. *Journal of Structural Mechanics*, 25(4):493–524, 1997.

[60] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[61] Henk A Van der Vorst. *Iterative Krylov methods for large linear systems*. Number 13. Cambridge University Press, 2003.

[62] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[63] Lloyd N Trefethen and David Bau. *Numerical linear algebra*, volume 181. Siam, 2022.

[64] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[65] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, 182(2):418–477, 2002.

[66] Grégoire Allaire, Sidi Mahmoud Kaber, Karim Trabelsi, and Grégoire Allaire. *Numerical linear algebra*, volume 55. Springer, 2008.

[67] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[68] Stephen F McCormick. *Multilevel adaptive methods for partial differential equations*. SIAM, 1989.

[69] Klaus Stüben. Algebraic multigrid (amg). an introduction with applications. 1999.

[70] Barry F Smith. *Domain decomposition methods for partial differential equations*. Springer, 1997.

TH-3072_146103004

[71] Andrew J Wathen. Preconditioning. *Acta Numerica*, 24:329–376, 2015.

[72] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[73] Anshul Gupta, Vipin Kumar, and Ahmed Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):455–469, 1995.

[74] Tae Soo Kim, Jae Eun Kim, and Yoon Young Kim. Parallelized structural topology optimization for eigenvalue problems. *International Journal of Solids and Structures*, 41(9-10):2623–2641, 2004.

[75] Niels Aage and Boyan S Lazarov. Parallel framework for topology optimization using the method of moving asymptotes. *Structural and multidisciplinary optimization*, 47(4):493–505, 2013.

[76] J París, I Colominas, F Navarrina, and M Casteleiro. Parallel computing in topology optimization of structures with stress constraints. *Computers & Structures*, 125:62–73, 2013.

[77] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[78] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[79] Sparsh Mittal and Jeffrey S Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[80] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[81] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

TH-3072_146103004

[82] CUDA NVIDIA. Basic linear algebra subroutines (cuBLAS) library, 2013.

[83] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[84] David Herrero-Pérez and Pedro J Martínez Castejón. Multi-GPU acceleration of large-scale density-based topology optimization. *Advances in Engineering Software*, 157:103006, 2021.

[85] Vivien J Challis, Anthony P Roberts, and Joseph F Grotowski. High resolution topology optimization using graphics processing units (GPUs). *Structural and Multidisciplinary Optimization*, 49(2):315–325, 2014.

[86] Hui Liu, Ye Tian, Hongming Zong, Qingping Ma, Michael Yu Wang, and Liang Zhang. Fully parallel level set method for large-scale structural topology optimization. *Computers & Structures*, 221:13–27, 2019.

[87] Yixin Li, Bangjian Zhou, and Xianliang Hu. A two-grid method for level-set based topology optimization with GPU-acceleration. *Journal of Computational and Applied Mathematics*, 389:113336, 2021.

[88] Jesús Martínez-Frutos and David Herrero-Pérez. GPU acceleration for evolutionary topology optimization of continuum structures using isosurfaces. *Computers & Structures*, 182:119–136, 2017.

[89] Laxman Ram and Deepak Sharma. Evolutionary and GPU computing for topology optimization of structures. *Swarm and evolutionary computation*, 35:1–13, 2017.

[90] David J Munk, Timoleon Kipouros, and Gareth A Vio. Multi-physics bi-directional evolutionary topology optimization on GPU-architecture. *Engineering with Computers*, 35(3):1059–1079, 2019.

[91] Niels Aage, Erik Andreassen, and Boyan Stefanov Lazarov. Topology optimization using PETSc: An easy-to-use, fully parallel, open source topology optimization framework. *Structural and Multidisciplinary Optimization*, 51:565–572, 2015.

[92] Thijs Smit, Niels Aage, Stephen J Ferguson, and Benedikt Helgason. Topology optimization using PETSc: a python wrapper and extended functionality. *Structural and Multidisciplinary Optimization*, 64:4343–4353, 2021.

TH-3072_146103004

[93] Abhinav Gupta, Rajib Chowdhury, Anupam Chakrabarti, and Timon Rabczuk. A 55-line code for large-scale parallel topology optimization in 2D and 3D. *arXiv preprint arXiv:2012.08208*, 2020.

[94] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[95] Erik A Träff, Anton Rydahl, Sven Karlsson, Ole Sigmund, and Niels Aage. Simple and efficient gpu accelerated topology optimisation: Codes and applications. *Computer Methods in Applied Mechanics and Engineering*, 410:116043, 2023.

[96] Sergei A Goreinov, EE Tyrtyshnikov, and A Yu Yeremin. Matrix-free iterative solution strategies for large dense linear systems. *Numerical linear algebra with applications*, 4(4):273–294, 1997.

[97] Emily D Sanders, Anderson Pereira, Miguel A Aguiló, and Glaucio H Paulino. Polymat: an efficient matlab code for multi-material topology optimization. *Structural and Multidisciplinary Optimization*, 58(6):2727–2759, 2018.

[98] Alexandre Nana, Jean-Christophe Cuillière, and Vincent Francois. Towards adaptive topology optimization. *Advances in Engineering Software*, 100:290–307, 2016.

TH-3072_146103004