GPU Acceleration of Finite Element Analysis and Its Application to Large-Scale Structural Topology Optimization

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

by

Subhajit Sanfui
(Roll No. 156103038)

to the

DEPARTMENT OF MECHANICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
February, 2022

CERTIFICATE

It is certified that the work contained in the thesis entitled "**GPU Acceleration of Finite Element Analysis and Its Application to Large-Scale Structural Topology Optimization**", by " Mr. Subhajit Sanfui", has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. D. Sharma
Department of Mechanical Engineering,
I.I.T. Guwahati.

February, 2022.

Dedicated to
My Parents

Acknowledgement

First, I would like to express my sincerest gratitude to my supervisor, Dr. Deepak Sharma, for his continuous support and guidance during my Ph.D. study and research work. His encouragement, motivation, knowledge, and patience have been invaluable to my work. It was a great honor and privilege to work under his guidance.

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

I would also like to thank my labmates, including my seniors, juniors, and batchmates, for our many stimulating discussions and friendships that made my stay at the Indian Institute of Technology, Guwahati, extremely memorable and fulfilling.

Finally, I am extremely grateful to my mother, without whom I would not have been able to complete this thesis, for her constant support, understanding, and care, which gave me great motivation and courage to undertake and successfully complete my doctoral journey.

Subhajit Sanfui
IIT Guwahati
February, 2022

Abstract

With the rapid advancement of GPU hardware, a trend for accelerating compute-intensive applications on massively parallel architectures has emerged. In keeping with this trend, several researchers in academia and industry are focusing their efforts on GPU-accelerated Finite Element Analysis (FEA) and its applications. Because FEA is a popular method for solving partial differential equations in a variety of engineering disciplines, any reduction in computational complexity through GPU acceleration would benefit a large portion of the scientific community. Furthermore, as problems of increasing complexity and size are introduced, the massive computational power that a modern GPU is capable of providing becomes more necessary. In a similar vein, this thesis seeks to address a number of issues concerning GPU acceleration of FEA and one of its most important applications, structural topology optimization. In the first part of this thesis, a novel three-stage GPU-based FEA matrix generation strategy is implemented with the key idea of decoupling the computation of global matrix indices and values by use of a novel data structure referred to as the neighbor matrix. The first stage computes the neighbor matrix on GPU based on the unstructured mesh. Using this neighbor matrix, the indices and values of the global matrix are computed separately in the second and third stages. The proposed method is compared to the state of the art GPU-based techniques for different element types, different benchmark problems and large-scale FEA meshes, demonstrating significant improvements in computation time and GPU performance. The second part of the thesis accelerates solid isotropic material with penalization (SIMP) method and bi-directional evolutionary structural optimization (BESO) method of topology optimization on GPU by incorporating algorithm-level and high performance computing-based modifications to the standard existing algorithms with a focus on the FEA stage. The key idea behind these implementations is to combine GPU acceleration of the entire application pipeline with a novel mesh reduction strategy that eliminates the need to compute empty voxels during the FEA stage of topology optimization. In the proposed strategy, the effective number of design variables is reduced by using the concept of *active* nodes and *active* elements in the finite element mesh. A novel mesh numbering scheme is also introduced to facilitate parallel identification of *active* nodes using the proposed GPU-based algorithm. The preconditioned conjugate gradient (PCG) solver is further developed using the proposed strategy and the numbering scheme. The proposed implementations of SIMP and

BESO are evaluated using different benchmark problems from the literature. When compared to the standard GPU-based implementation, the proposed GPU-adapted implementations provide three key benefits: shorter execution times, lower memory consumption, and improved FEA convergence, all of which mitigate the major computational issues associated with topology optimization.

Journal Publications

Published:

- Subhajit Sanfui and Deepak Sharma, 2021, "Symbolic and Numeric Kernel Division for GPU-based FEA Assembly of Regular Meshes with Modified Sparse Storage Formats", ASME Journal of Computing and Information Science in Engineering, 22 (1), 1-12. https://doi.org/10.1115/1.4051123
- Subhajit Sanfui and Deepak Sharma, 2020, "A Three-Stage GPU-based FEA Matrix Generation Strategy for Unstructured Meshes", International Journal of Numerical Methods in Engineering, 121 (17), 3824-3848. https://doi.org/10.1002/nme.6383

In review:

- Subhajit Sanfui and Deepak Sharma, "GPU-based mesh reduction strategy for accelerating structural topology optimization", Applied Soft Computing
- Subhajit Sanfui and Deepak Sharma, "Soft- and Hard-Kill Hybrid GPU-based Bi-Directional Evolutionary Structural Optimization", Structural and Multidisciplinary Optimization
- Subhajit Sanfui and Deepak Sharma, "A 250-line Fully Parallelized CUDA Code for Large-Scale Bi-Directional Evolutionary Structural Optimization using GPU", Structural and Multidisciplinary Optimization

In preparation:

- Subhajit Sanfui and Deepak Sharma, A Review of GPU accelerated FEA for Structural Analysis.

Conference Publications

- Subhajit Sanfui, Shashi Kant Ratnakar, Deepak Sharma, "A Parametric Study on the Convergence Behaviour of SIMP-Based Structural Topology Optimization using GPU", 4^{th} National Conference on Multidisciplinary Design, Analysis, and Optimization (NCMDAO 2021), 7-9 October 2021, IIT Madras, India.
- Subhajit Sanfui and Deepak Sharma, "Exploiting Symmetry in Elemental Computation and Assembly Stage of GPU-Accelerated FEA", Proceedings at the 10th International Conference on Computational Methods (ICCM2019), 9–13 July 2019, Singapore, Eds: G.R. Liu, Fangsen Cui, George Xu Xiangguo, ScienTech Publisher, pp. 641 – 651
- Subhajit Sanfui and Deepak Sharma, "GPU Acceleration of Local Matrix Generation in FEA by Utilizing Sparsity Pattern", In 1st International Conference on Mechanical Engineering (INCON 2018), 4–6 January 2018, Jadavpur University, India.

- 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, 3-5 February 2017, MNNIT Allahabad, India.

Other Publications

- Shashi Kant Ratnakar, Subhajit Sanfui and Deepak Sharma, "GPU-based Element-by-Element Strategies for Accelerating Topology Optimization of 3D Continuum Structures Using Unstructured Mesh", ASME Journal of Computing and Information Science in Engineering, 1-17. https://doi.org/10.1115/1.4052892

- Shashi Kant Ratnakar, Subhajit Sanfui and Deepak Sharma, "GPU – based Topology Optimization using Matrix-Free Conjugate Gradient Finite Element Solver with Customized Nodal Connectivity Storage", 2nd International Conference on Future Learning Aspects of Mechanical Engineering (FLAME - 2020), August 5 – 7, 2020, Amity University Uttar Pradesh, Noida, India

- Shashi Kant Ratnakar, Subhajit Sanfui and Deepak Sharma, "SIMP-based Structural Topology Optimization using Unstructured Mesh on GPU", 2nd International Conference on Future Learning Aspects of Mechanical Engineering (FLAME - 2020), August 5 – 7, 2020, Amity University Uttar Pradesh, Noida, India

- 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Finite element method (FEM) is a numerical method used for approximating solutions of boundary value problems for partial differential equations (PDEs). This method is utilized for accurate and precise modelling of thermal, mechanical, and other complex systems in several fields such as mechanical engineering, civil engineering, electrical engineering, medical applications and others. The primary mechanism behind FEM is to split a complex domain into smaller subdomains called elements using a network of nodes, finding local solutions that satisfy the governing PDE and, subsequently, combining the local solutions to obtain a global solution. Due to several natural advantages offered by FEM, such as flexibility, adaptability, and ease of implementation even for very complex and intricate geometries, it has become an integral part of a large variety of specializations. In industries such as aviation, automotive, and construction, FEM is typically an essential part of the design process (Zienkiewicz et al., 1977).

The process of applying the concepts of FEM for the analysis of physical systems is often referred to as "Finite Element Analysis (FEA)". Starting from a CAD model of the problem domain, a typical FEA consists of three basic steps.

1. *Pre-Processor Step* – In this step, the problem domain is defined by specifying the material properties, type of element, nodal coordinates and connectivity.

2. *Solver Step* – In this step, elemental matrices are computed and assembled into a global stiffness matrix. This matrix, after the application of proper boundary conditions, results in a set of algebraic equations. Upon solving, this system of equations yields the response of the physical system under the applied loading conditions.

3. *Post-processor Step* – In this final step, the values of the primary unknowns are used to extract several useful information about the problem domain; for example, information pertaining to

stresses, strains, etc. Plotting of relevant data is also performed in this stage.

Although FEA is a widely popular method utilized in many fields, it often suffers from high computational intensity, especially for real-world problems. This is primarily caused by the computations performed in the solver step (Cecka et al., 2011), which involves local stiffness matrix generation for all elements, their assembly, and solution of the resulting global system of equations. Among these, the solution of the linear system of equations is typically the most time-consuming (Georgescu et al., 2013). The choice of linear solver is a contributing factor in the final execution time, along with the required level of accuracy in the final solutions. The second-most time-consuming part is the matrix generation stage that includes numerical integration for all the elements and assembly into a global stiffness matrix (Cecka et al., 2011).

Typically, FEA is performed once for the analysis of a component. However, many real-world applications need it repetitively. For example, Nair et al. (2007) combined FEA with genetic algorithm (GA) for a biomechanical application, where FEA was performed for each GA individual. Despite performing tens of thousands of FEA, computation time was kept manageable by using a coarse mesh with a relatively low population size for GA. Another important application that requires repeated large-scale and high-fidelity solutions to FEA is structural topology optimization. For example, a structural topology optimization application may require anywhere from a couple of hundreds (Ratnakar et al., 2021a) to hundreds of thousands (Jansen et al., 2014) of FEA computations. In such applications, where the use of large-scale FEA is encouraged from the point of view of quality of final outcomes, the computational complexity of FEA often becomes the bottleneck in successful implementations as well as in further research and developments. In the last two decades, with advancements in computational hardware as well as through continuous research efforts of several research groups worldwide, structural topology optimization has emerged as one of the most important applications of FEA (Mukherjee et al., 2021). A large number of research studies on developing structural topology optimization methodology has been conducted, as well as on reducing its computational complexity resulting majorly from FEA. On a similar note, the aim of this thesis is to study and document these research efforts, to identify the research gaps, and to finally implement novel methodologies with the key objective of reducing the computational complexity in FEA with structural topology optimization as one of its primary applications.

Structural topology optimization is a mathematical method for optimizing material layout within a given design domain, with the aim to maximize the performance of a structure under a set of constraints and boundary conditions. Starting from the introduction of the homogenization method by Bendsøe and Kikuchi (1988), it has been used extensively in the fields of civil and mechanical engineering for reducing the material in a structure while maintaining its structural integrity (Bendsøe and Sigmund, 2003). Since its introduction, many methods for topology optimization have been developed, including level-

set method (Wang et al., 2003, Allaire et al., 2004), bidirectional evolutionary structural optimization (BESO) (Tang et al., 2015), solid isotropic material with penalization (SIMP) method (Bendsøe, 1989, Zhou and Rozvany, 1991, Mlejnek, 1992), etc. Martínez-Frutos et al. (2017) classified these methods into three broad categories: Eulerian, Lagrangian, and density-based methods. Methods such as the level-set method (Wang et al., 2003), where the structural boundaries of the domain are implicitly represented by a level-set model and those boundaries are manipulated according to the level sets, are categorized as Eulerian methods. On the other hand, in Lagrangian methods, the domain boundaries are explicitly represented (Misztal and Bærentzen, 2012, Christiansen et al., 2014). The third and most popular class of topology optimization methods is the density-based methods, which includes SIMP (Bendsøe, 1989) and homogenization methods (Bendsøe and Kikuchi, 1988). These methods employ density as the design variable in order to obtain black (material) and white (void) structures.

In addition to being widely used in solid mechanics (Ram and Sharma, 2017), topology optimization has found scope for application in fields such as biomedical engineering (Cucinotta et al., 2019), fluid mechanics (Duan et al., 2015), heat transfer (Bruns, 2007), acoustics (Dühring et al., 2008)), multi-physics (Munk et al., 2018), and others. Similar to FEA, structural topology optimization also suffers from a high computational cost, especially in many real-world applications. Since its inception more than 30 years ago, numerous advancements have been made for improving the performance of topology optimization methods in the last two decades. Despite these advancements, improving the computational efficiency of these methods is still the key challenge (Deaton and Grandhi, 2014). This challenge stems from the prohibitively high computational requirements of real-world topology optimization problems, which hinders its widespread use in industrial design. These problems are often large scale, involving a large number of design variables that could range from several millions to a few billions (Aage et al., 2017). Even a latest high-end computing hardware is not capable of effectively handling topology optimization problems of this scale. Furthermore, with an increase in the number of progressively complex real-world problems, there is a simultaneous increase in the need to mitigate the high computational requirements of topology optimization, which has been addressed by many studies in the literature (Mukherjee et al., 2021).

This thesis aims to address the problem of the computational complexity of FEA with topology optimization as its application. In general, for both FEA and structural topology optimization, the computational complexity stems primarily from three key factors.

- Large problems that require high computation time
- Large-scale meshes with high memory requirements
- Irregular patterns of data that affect efficient memory management

The research efforts to address these issues in the literature can be broadly classified into two

categories: algorithm-level modifications (Liao et al., 2019, De Troya and Tortorelli, 2018) and high performance computing (HPC)-based modifications (Schmidt and Schulz, 2011, Martínez-Frutos and Herrero-Pérez, 2016, Ram and Sharma, 2017). Algorithm-level modifications for FEA include the use of model simplification, sparse storage of data designed for FEA (Sanfui and Sharma, 2021), and utilization of assembly-free methods for handling the computational issues. Similarly, algorithm-level modifications for structural topology optimization include multilevel meshes, adaptive refinements, the use of efficient solution techniques for linear systems, and reduced basis methods, among others. On the other hand, HPC-based modifications for both FEA and structural topology optimization involve the use of parallel computing on different types of hardware to accelerate the application. Although HPC is an umbrella term encompassing different techniques for efficiently handling data and performing complex computations efficiently at high speeds, it has become almost synonymous with parallel computing where a large computational task is divided into smaller tasks and handled in parallel. Among the several different types of parallel platforms, massively parallel processors such as GPUs have attained widespread use owing to their many advantages, including low price-to-performance ratio, low running and maintenance cost, and an inherent suitability for handling large-scale data-parallel and throughput-intensive applications such as FEA and structural topology optimization. This has been corroborated by a large number of research works focusing on efficient GPU parallelization of FEA and topology optimization with great success (Georgescu et al., 2013, Mukherjee et al., 2021).

1.1 Motivation

Matrix generation and linear solver are the two important parts of FEA in terms of the fraction of total time consumed, as observed from the literature. For solving large linear system of equations, matrix-free methods are generally preferred on the GPU (Schmidt and Schulz, 2011). On the other hand, matrix generation for a system with a large number of degrees of freedom during FEA poses a challenge for many traditional methods in the literature. One of the primary reasons is the need to use sparse storage formats that create challenges in the assembly operation. These challenges become more prominent for assembly into 3D unstructured meshes for the irregular patterns of the non-zero (NZ) entries in the global stiffness matrix. For the assembly of an unstructured mesh directly into a specialized sparse storage format on GPUs, two important challenges can be observed. The first challenge is to calculate the memory required for the specific sparse storage format a priori (Dziekonski et al., 2013, Sanfui and Sharma, 2017a). The second challenge is the search operation needed to find the location for writing NZ entries in the specific sparse format (Sanfui and Sharma, 2017a). In general, these operations are considered undesirable on GPU, since they cause branching, which affects the performance of the application. It is also observed that the index computation of the entries of the global stiffness matrix depends only on the mesh of the domain and can be separated from the value computation for the

NZ entries. This thesis aims to focus on these challenges in assembly for 3D unstructured meshes by utilizing a novel divide-and-conquer implementation that divides the FEA matrix generation into three distinct stages.

In the literature of GPU-accelerated FEA, the computations of elemental matrices and their assembly into a global stiffness matrix have been done either by developing single kernel (Fu et al., 2014, Reguly and Giles, 2015, Kiran et al., 2018) or separate kernels (Cecka et al., 2011, Sanfui and Sharma, 2017a). The former approach obviates the need of storing matrices in the global memory, while the latter performs computing and assembly in separate kernels without overburdening individual compute threads of GPU. Since these approaches have not been compared in the prior studies, this thesis aims to compare them on both structured and unstructured finite element meshes.

From the extensive literature review, it has been observed that similar to FEA, several efforts have also been made to reduce the computational cost of topology optimization using algorithm-level and HPC-based modifications. The algorithm-level modifications require the researcher to focus on the *physics* or *mathematics* of the problem, whereas, for HPC-based modifications, the *efficient implementation* of the standard algorithm on the target architecture is the primary focus. To the best of our knowledge, no study in the literature has aimed at combining algorithm-level and HPC-based modifications for handling the computational cost of traditional topology optimization. In this thesis, we aim to fill this gap by combining these two approaches in order to significantly reduce the computational cost of topology optimization. As shown by Liao et al. (2019) and Wang et al. (2020), among the three algorithm-level modifications implemented, the reduction of design variables by a local update strategy has been proven to be the most effective approach. This thesis implements a similar mesh update strategy that is suitable for massively parallel architectures. It also discusses the challenges associated with the implementations, along with the key modifications to the algorithm for two different density-based topology optimization methods, that are, SIMP and BESO. It is further observed from the literature that, unlike the SIMP method, GPU acceleration of BESO is relatively an open research field with a limited number of research publications. Furthermore, no research study has focused on complete GPU acceleration of the BESO method pipeline to the best of the authors' knowledge. The need for complete parallelization on GPU arises primarily from the slow data transfer speed between a CPU and a GPU, which often hinders the performance of an application. Owing to this reason, applications utilizing GPUs for parallelization often tend to offload the entire computation on GPUs to avoid the expensive CPU-GPU transfer. In this thesis, a fully accelerated BESO is presented that includes parallelization of all major steps on GPU. Furthermore, this thesis also presents a complete GPU implementation code that includes all steps starting from meshing to the plotting of the final topology.

We also identified several key aspects and research gaps in the area of large-scale GPU-based acceleration of FEA and density-based structural topology optimization, which would benefit a wide variety of fields in the scientific community. In addition to filling some of those research gaps, we also focused

on building a versatile HPC tool for the entire linear FEA process and density-based structural topology optimization. This tool incorporates parallelization of entire algorithms with a particular focus on the individual building blocks of algorithms. The primary targets can be summarized into two important challenges observed in the literature: reduce execution time and reduce memory footprint of parallel applications (Mukherjee et al., 2021). These are addressed by incorporating a combination of several algorithm-level and HPC-based enhancements to the standard FEA and density-based topology optimization methods.

1.2 Objectives of the Thesis

Following are the key objectives of the thesis.

1. Generation of elemental matrices for different finite elements with their assembly and storage on GPUs
2. Implementation of topology optimization with mesh reduction on GPU using SIMP method
3. Complete GPU acceleration for large-scale bi-directional evolutionary structural optimization (BESO)
4. Development of GPU-based hybrid BESO method using the mesh reduction approach
5. Performance analysis of the developed tools for benchmark problems and comparison with the state of art

1.3 Organization of Thesis

The rest of the thesis has been organized as follows.

In **Chapter 2**, the preliminaries including the details of GPU computing, FEA, and topology optimization are discussed. This is followed by the literature review of GPU-accelerated FEA and density-based topology optimization.

Chapter 3 presents efficient GPU acceleration of the matrix generation stage in FEA. This includes parallel generation of elemental stiffness, the proposed kernel division strategy for structured and unstructured FEA assembly, and finally the efficient storage of the global stiffness matrix on GPU.

Chapter 4 presents the mesh reduction strategy that is applied to the SIMP method on GPU with matrix-free FEA. Implementation-level details are also provided with a focus on the mesh update strategy, mesh renumbering scheme, and parallel computation of active nodes.

Chapter 5 presents the complete GPU acceleration of the BESO method pipeline. In this chapter, a compact 250-line code for GPU-accelerated FEA is provided that includes all steps of optimization

starting from meshing to the plotting of the optimal topology. This chapter also discusses the proposed hybrid BESO method that combines the soft-kill and hard-kill formulations. The strategies for on-the-fly computation of active DOFs, along with all the hybrid BESO implementation steps, have been discussed according to their respective acceleration strategies.

Chapter 6 presents the important conclusions of this thesis and the future work.

Chapter 2

Literature Review

This chapter starts with a brief introduction of FEA, structural topology optimization, and GPU computing in order to achieve a complete understanding the literature on GPU acceleration of FEA and density-based topology optimization discussed later in this chapter.

2.1 Preliminaries: FEA

A wide variety of physical phenomena, including the response of mechanical systems under specific loading conditions can be described by PDEs obtained from the basic physics of the problem. These PDEs, however, in most real-life applications, cannot be solved analytically due to factors such as non-linearity and complexity of the domain. In such cases, it is necessary to use numerical methods such as FEA. Using FEA, the solution to the governing PDE is obtained by breaking up the problem domain into smaller parts called finite elements as presented in Figure 2.1. In this process, specific functions are derived, known as shape functions to approximate the values of the primary unknown over the entire domain. Elemental contributions of each of the elements are computed and assembled to construct the global stiffness matrix.

2.1.1 Linear Elasticity

The study of stresses and elastic deformations in solid bodies subjected to prescribed loading conditions is known as linear elasticity. The governing PDEs of a linear elasticity problem are usually referred to as boundary value problems. The governing equations including (Zienkiewicz et al., 1977) the strain-displacement equations, equilibrium relations, and constitutive relations are given in equation (2.1).

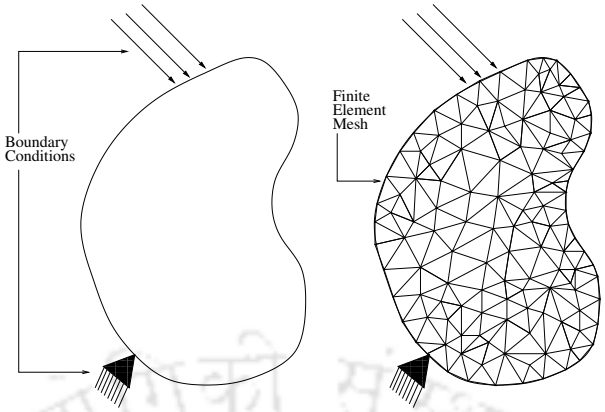


Figure 2.1: Typical finite element mesh

$$\begin{aligned} \sigma_{ij,j} + b_i = &\quad \rho\ddot{u}_i, \qquad\qquad i,j = 1,2,3, \\ \epsilon_{ij} = &\quad \tfrac{1}{2}(u_{j,i} + u_{i,j}), \\ \sigma_{ij} = &\quad C_{ijkl}\epsilon_{kl}. \end{aligned} \tag{2.1}$$

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

Equation (2.1) is subjected to the Dirichlet and Neumann boundary conditions, which are given by,

$$u_i = \bar{u}_i \text{ on } \Gamma_u, \qquad\qquad t_i = \bar{t}_i \text{ on } \Gamma_t, \tag{2.2}$$

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

2.1.2 Finite Element Formulation

In order to avoid the difficulties of directly handling the strong form of the PDE, the continuity requirements are reduced to obtain its weak form, which can then be solved over a discretized domain (Zienkiewicz et al., 1977).

For constructing the weak form, equation (2.1) is multiplied by an arbitrary vector and integrated over the domain Ω. We obtain the virtual work as weak form where the arbitrary function is the virtual displacement δu_i. By applying Gauss divergence theorem and traction boundary conditions with Cauchy's rule, we get,

$$\int_{\Omega} \delta u_i \rho \ddot{u}_i d\Omega + \int_{\Omega} \delta \varepsilon_{ij}(\underline{u}) \sigma_{ij} d\Omega - \int_{\Omega} \delta u_i b_i d\Omega - \int_{\Gamma_t} \delta u_i \bar{t}_i d\Gamma = 0 \ , \tag{2.3}$$

where the four terms on the left-hand side of the equation denote the virtual work of the inertial forces, internal forces, body forces and traction forces, respectively. Equation (2.3) calculated over all the elements of the discretized domain, written in matrix form gives,

$$\sum_{e=1}^{n_e} [\int_{\Omega} \delta\{u\}^T \rho\{\ddot{u}\} d\Omega + \int_{\Omega} \delta([S]\{u\})^T \{\sigma\} d\Omega - \int_{\Omega} \delta\{u\}^T \{b\} \delta\Omega - \int_{\Gamma_t} \delta\{u\}^T \{\bar{t}\} d\Gamma] = 0 \ . \tag{2.4}$$

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

$$\{u\}^e \approx [N]^e \{\tilde{u}\}^e, \tag{2.5}$$

where $[N]^e$ are the shape functions used for approximation. Substituting (2.5) into (2.4) and performing assembly over all the elements, we get,

$$\sum_{e=1}^{n_e} [[M]^e \{\ddot{u}\}^e + [K]^e \{\tilde{u}\}^e - \{f\}^e_{ext}] = 0, \tag{2.6}$$

where $[M]^e, [K]^e$ and $\{f\}^e_{ext}$ are the elemental mass matrix, stiffness matrix and external force vector, respectively. The inertia term in (2.6) becomes zero for a static problem.

2.1.3 Assembly

This thesis focuses on FEA of two-dimensional and three-dimensional elastic structures, which are discretized using different types of finite elements. As an example, for 20-noded hexahedral (HEX20) and 4-noded tetrahedral (TET4) element types, equation (2.7) can be used to compute the elemental stiffness matrices (Zienkiewicz et al., 1977). In equation (2.7), partial derivatives of the shape functions are stored in B with respect to x, y and z directions. ξ, η and ζ constitute the local coordinate system for each individual element. The Jacobian and constitutive matrices are represented by J and D, respectively. The Gauss Quadrature (GQ) weights for integration are denoted by w_{ξ_i}, w_{η_j} and w_{ζ_k}. n_g is the number of Gauss points required. The assembly of all the elemental $[K]^e$ results in the global stiffness $[K]$ matrix, which is solved to obtain the values of the primary unknown over the entire mesh, following the application of boundary conditions.

$$
\begin{aligned}
[K]^e &= \int_V [B]^T[D][B]\ dxdydz \\
&= \int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} B(\xi,\eta,\zeta)^T DB(\xi,\eta,\zeta) \mid J(\xi,\eta,\zeta) \mid d\xi d\eta d\zeta \\
&= \sum_{i=1}^{n_g}\sum_{j=1}^{n_g}\sum_{k=1}^{n_g} w_{\xi_i} w_{\eta_j} w_{\zeta_k} B(\xi_i,\eta_j,\zeta_k)^T DB(\xi_i,\eta_j,\zeta_k) \mid J(\xi_i,\eta_j,\zeta_k) \mid
\end{aligned}
\tag{2.7}
$$

The generation of elemental stiffness and their assembly together is referred to as the matrix generation stage throughout this thesis. In the next section, the preliminaries about topology optimization is discussed.

2.2 Preliminaries: Topology Optimization

Topology optimization is essentially a material-distribution problem that aims to find the optimum topology under a defined design domain and a set of boundary conditions and constraints. Among the different topology optimization methods, SIMP method is the most widely used in the literature (Deaton and Grandhi, 2014).

2.2.1 Solid Isotropic Material with Penalization

In this method, a density variable is assigned to each element in the finite element discretization. These elemental densities dictate the presence of material inside the design domain and, hence, the final resulting topology. The topology optimization problem can be formulated as,

$$
\begin{aligned}
\underset{d}{\text{Minimize}} \quad & c(u(d),d) = \{u\}^T[K]\{u\}, \\
\text{s.t.} \quad & V(d)/V_0 = v_f, \\
& [K(d)]\{u(d)\} = \{F\}, \\
& d_i \in [0,1],
\end{aligned}
\tag{2.8}
$$

where compliance c is the objective function, u is the displacement, K is the global stiffness matrix, and F is the global load vector. Here, density d is the vector of design variables, $V(d)$ is the total volume of the structure, and V_0 is the maximum volume of the design domain. v_f is the required volume fraction. In reality, the optimization problem in equation (2.8) results in structures with intermediate densities, which are impossible to manufacture. To counter this problem, SIMP method employs a penalization

method that discourages the appearance of intermediate densities in the final resulting structures. In this method, the elemental stiffness matrices are calculated as,

$$K_i = K_{min} + d_i^P (K_{max} - K_{min}), d_i \in [0, 1]. \tag{2.9}$$

K_{max} and K_{min} are the elemental stiffness matrices for a completely solid and completely void element, respectively. As standard practice, a small nonzero stiffness is used for the void elements (Sigmund, 2001). P is the penalization parameter. Typically, the value of P is taken as more than three. The problem can be solved using different optimization techniques such as optimality criteria (OC), sequential linear programming (SLP), and the method of moving asymptotes (MMA), among others. In this thesis, the OC-based method has been used, and a heuristic-based updating scheme is used for the design variables. The scheme (Bendsøe and Sigmund, 2003) is formulated as,

$$d_i^{new} = \begin{cases} max(d_{min}, d_i - m) & \text{, if } d_i B_i^\eta \leq max(d_{min}, d_i - m), \\ min(1, d_i + m) & \text{, if } d_i B_i^\eta \geq min(1, d_i + m), \\ d_i B_i^\eta & \text{, otherwise,} \end{cases} \tag{2.10}$$

where m is a positive move limit, η is a numerical damping coefficient, and B_i is defined as,

$$B_i = (-\frac{\partial c}{\partial d_i})/(\lambda \frac{\partial V}{\partial d_i}). \tag{2.11}$$

A bisection search needs to be performed for finding the value of the Lagrangian multiplier, λ. The sensitivity of objective function c is found as,

$$\frac{\partial c}{\partial d_i} = -p d_i^{p-1} u_i^T K_i u_i. \tag{2.12}$$

u_i are the nodal displacements. In order to handle the issues of mesh dependency and checkerboard patterns, a mesh sensitivity filter is used. This filter has been used in several studies in the literature and has been proven to be effective in solving these issues. Details about the mesh independency filter can be found in the work by Sigmund (2001). Figure 2.2 illustrates the flowchart of topology optimization with an approximate percentage of time consumed at each stage. It is observed that approximately 95% of the total time is consumed by FEA in each iteration. In the next section, details about BESO method is presented.

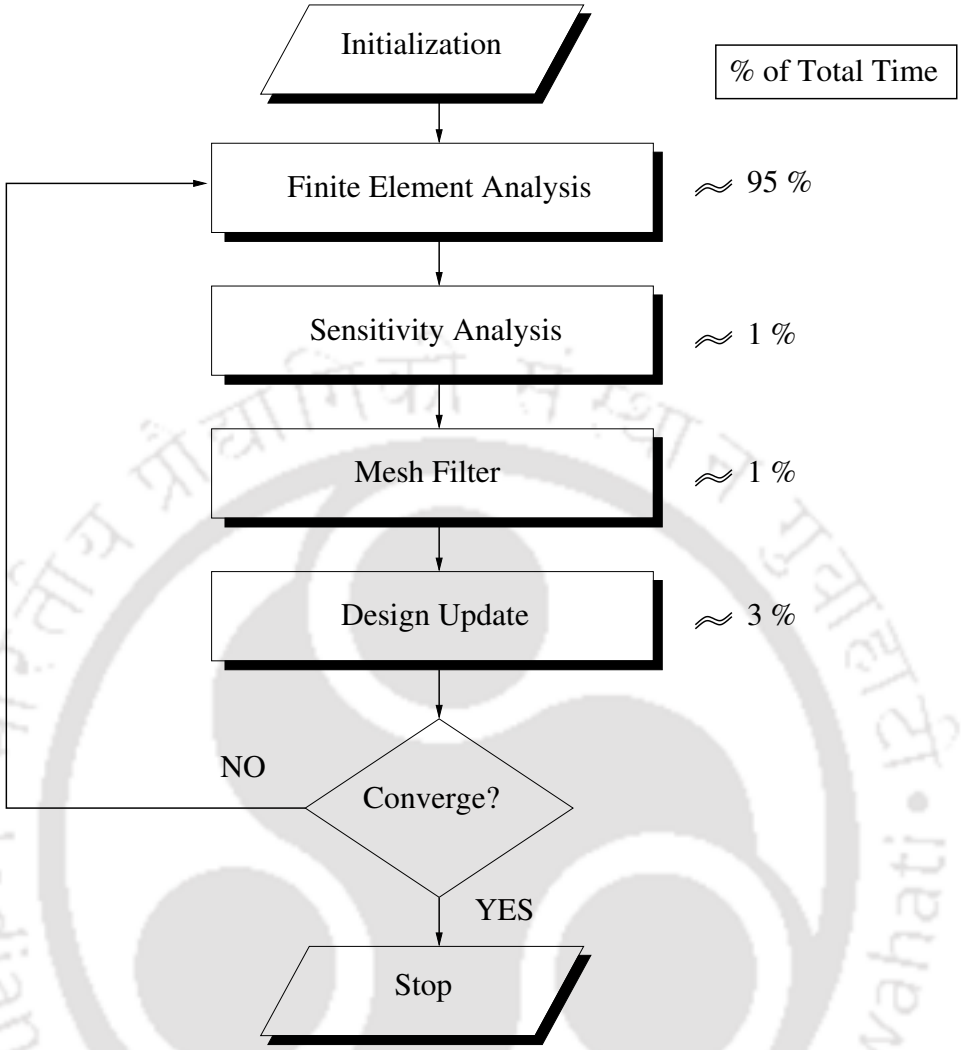


Figure 2.2: Flowchart of the main steps in SIMP-based topology optimization with the percentage of computation time of each step. FEA is performed on the GPU. The respective times are shown for SIMP-based topology optimization of a cantilever beam with 100 iterations.

2.2.2 Bi-Directional Evolutionary Structural Optimization

The basic idea behind BESO is simple and intuitive. A completely void or completely solid structure is taken at the start of the optimization. FEA is performed at every iteration of BESO and depending on the results of FEA, inefficient elements of the mesh are removed and efficient elements are added simultaneously. This removal and addition of elements to the mesh are done by calculating the sensitivities of the finite element mesh and setting a threshold value at every iteration following the FEA step till convergence is reached. At the start of the algorithm, the BESO parameters and the initial structure are set with the proper boundary condition and material properties. Then, FEA is performed to obtain the response of the system to the prescribed loading conditions, which is followed by the calculation of the sensitivities for each element in the mesh. The structural compliance is also computed as

the objective function. Subsequently, a mesh-independency filter is used to address several issues such as checkerboard patterns and mesh-dependent solutions similar to the SIMP method. A stabilization technique is then applied by averaging the individual sensitivity values. Next, the structure is updated using the filtered sensitivities, and the termination conditions are checked. If the conditions are not met, FEA is run again with the updated structure, and this loop continues. The BESO method can be divided into two categories, namely the soft-kill and the hard-kill, details of which are presented in the following subsections.

2.2.2.1 Hard-kill BESO

The optimization problem of a hard-kill BESO for compliance minimization can be stated as,

$$\begin{aligned} \text{Minimize} \quad & C = \frac{1}{2}\{u\}^T[K]\{u\}, \\ \text{s.t.} \quad & V_f - \sum_{i=1}^{n} v_i \rho_i = 0, \\ & [K]\{u\} = \{F\}, \\ & \rho_i = 0 \quad \text{or} \quad 1, \end{aligned} \tag{2.13}$$

where mean compliance C is the objective function, $\{u\}$ is the vector of displacements, $[K]$ is the global stiffness matrix, and $\{F\}$ is the load vector. v_i are the elemental volumes and V_f is the prescribed final volume of the structure. ρ_i are the binary design variables signifying either a solid or void element. It should be noted here that the values and sizes of $\{u\}$,$[K]$, $\{F\}$, v_i and ρ_i are based only on the solid portion of the rectangular grid only, and not on the entire grid. The elemental sensitivity α_i^e for the i^{th} element is calculated as,

$$\alpha_i^e = \frac{1}{2} u_i^T [K]^i u_i, \tag{2.14}$$

where u_i and $[K]^i$ are the elemental displacement vector and elemental stiffness matrix for the i^{th} element, respectively. This number physically represents the change in mean compliance as a result of removal or addition of the i^{th} element. The addition and removal of new elements are performed based on a threshold value α_{th}, set at every iteration, based on the target volume V_{it} of that iteration. The value of α_{th} is determined from the sorted list of individual elemental sensitivities. For example, there are 100 elements in the mesh and $\alpha_1 > \alpha_2 > \alpha_3 > ... > \alpha_{100}$. If V_{it} corresponds to a structure with 70 elements, then $\alpha_{th} = \alpha_{70}$.

Typically, a mesh-independency filter is applied to the sensitivities before constructing a new structure. A value of the filter radius is set that identifies the set of neighboring elements for each element in the mesh as shown in Figure 2.3. It is worth noting here that, apart from solving mesh dependency

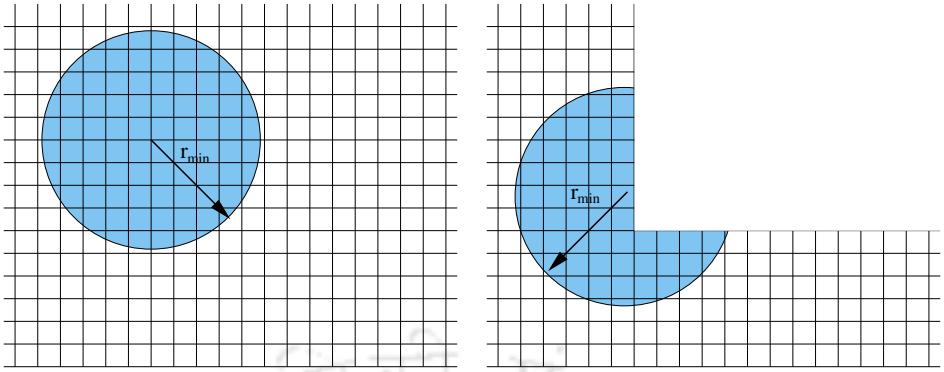


Figure 2.3: Sub-domain for mesh filter using filter radius.

and checkerboard patterns, the filtering scheme is essential for hard-kill BESO, since it works as a way of extrapolating sensitivities of void elements after their removal from the FE model. In this thesis, the filter scheme presented by Huang and Xie (2010a) is used. A vector of weights $[w]$ is computed for each element that contains the weights for all of its neighbors. If there are total ne number of neighbors, the weights are calculated as $w_k = r_{min} - r_k, k = 1, 2, ..., ne$. If the sensitivities of the neighboring elements are stored in vector $[\alpha^e]$, the filtered sensitivities are calculated as,

$$\alpha_i = [w]^T[\alpha^e]/W, \tag{2.15}$$

where W is the sum of all the weights in $[w]$. Finally, a stabilization scheme is used on the filtered sensitivities by averaging them with their historical information in order to improve the convergence.

2.2.2.2 Soft-kill BESO with material interpolation

The optimization problem of a soft-kill BESO with material interpolation for compliance minimization can be stated as,

$$\begin{aligned} \text{Minimize} \quad & C = \frac{1}{2}\{u\}^T[K]\{u\}, \\ \text{S.t.} \quad & V_f - \sum_{i=1}^{n} v_i\rho_i = 0, \\ & [K]\{u\} = \{F\}, \\ & \rho_i = \rho_{min} \quad \text{or} \quad 1, \end{aligned} \tag{2.16}$$

where ρ_{min} is the minimum set density for the structure. The only apparent difference between equation (2.16) and equation (2.13) is the inclusion of ρ_{min}. However, it is important to note that, unlike the hard-kill BESO, the values and sizes of vectors $\{u\}$,$[K]$, $\{F\}$, v_i, and ρ_i are based on the entire structure,

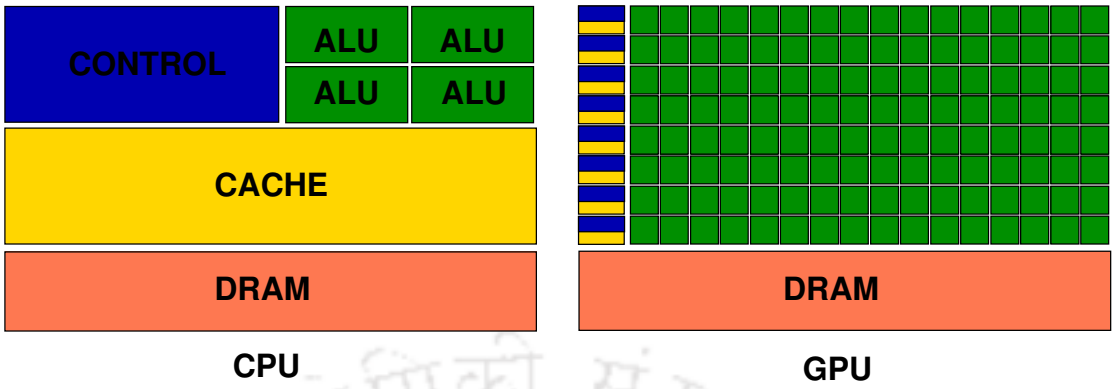


Figure 2.4: Simplified model of CPU and CPU architectures.

including the void elements. The soft elements are penalized using penalty parameter p. This is done by modifying the Young's modulus of all elements in the mesh as,

$$E(\rho_i) = E_0 \rho_i^p, \tag{2.17}$$

where E_0 represents the Young's modulus of the solid element. Further, instead of directly assembling the global stiffness matrix, as in the hard-kill approach, elemental stiffness matrices are multiplied with the penalized densities. The global stiffness matrix for soft-kill BESO is calculated as,

$$[K] = \sum_{i=1}^{n} \rho_i^p [K]_i. \tag{2.18}$$

Here, the summation symbol represents FEA assembly of elemental stiffness matrices, and not simple addition. Finally, the elemental sensitivities are calculated as,

$$\alpha_i^e = \begin{cases} \frac{1}{2} u_i^T [K]_i u_i, & \text{when} \quad \rho_i = 1 \\ \frac{\rho_{min}^{p-1}}{2} u_i^T [K]_i u_i, & \text{when} \quad \rho_i = \rho_{min}. \end{cases} \tag{2.19}$$

Following the sensitivity calculation, the same filter is used as in equation (2.15). In the next section, the details about GPU computing is presented.

2.3 Preliminaries: GPU Computing

In this thesis, a hybrid computing environment is used, which comprises of a CPU and a GPU. From an architectural point of view, CPUs and GPUs are vastly different from each other; thus, they are suitable for particular sets of tasks. A comparison of the simplified architectures of a CPU and a GPU is shown

in Figure 2.4 (Kirk and Hwu, 2010). It can be seen that a GPU utilizes its chip area to accommodate a high number of individual processors. On the other hand, a CPU uses a significantly large cache unit and multiple processors, which, although only a few in number, are significantly faster (in terms of processor clock speed) and smarter (in terms of available instruction sets) than GPU cores. Therefore, a CPU is more capable of quickly handling a variety of tasks; however, it is limited in terms of the ability to run tasks concurrently. As shown in Figure 2.4, the high number of cores available in a GPU enables it to run a large number of tasks simultaneously, making it highly suitable for carrying out tasks that are data parallel and throughput intensive. In a heterogeneous computing environment, such as the one used in this thesis, a CPU typically deals with the more logical and sequential parts of the computation, whereas a GPU is responsible for handling the high-intensity arithmetic. Through proper use of the complex memory and thread hierarchy of GPUs, highly efficient applications can be written that can harness the parallel computing power of a GPU.

2.3.1 Memory Hierarchy in GPU

GPUs have a memory hierarchy that is significantly more complex than that of a CPU. A GPU contains different types of on-chip and off-chip memories, each with special purpose characteristics. The three main types of GPU memories are registers, shared memory and global memory. Registers are fast on-chip memories that are exclusive to a particular thread. Shared memory is also an on-chip memory that is exclusive to a thread block and, thus, visible to all the threads in that particular block. Lastly, global memory is the largest in size and visible to all the threads at all times. However, it is off-chip with a high latency, making it significantly slower than the shared memory and registers. Besides, the other forms of memories available on the GPU are local memory, constant memory, and texture memory.

2.3.2 Thread Hierarchy in GPU

A kernel is a special function that is executed on the GPU and contains instructions to be carried out by a single thread in the grid. The grid dimensions are passed as the kernel-launch parameters. A grid typically contains several thousands to a few million threads. These threads are then divided into several thread blocks that run concurrently on a streaming multiprocessor. These thread blocks are further divided into single instruction, multiple threads (SIMT)-groups called warps, containing 32 threads. In this complex hierarchy of threads, a programmer has control only over the grid and block size. The rest of the layout, including the creation of warps, is implicitly controlled by the GPU. The GPU threads can be configured in one, two, or three dimensions, as required by the computation.

2.3.3 CUDA Programs

The most commonly used API for programming GPUs is CUDA, which was developed by NVIDIA. A typical CUDA program consists of the host code, which is to be executed on the CPU, and the device code, which is to be executed on the GPU. The host code is written using standard C/C++/Fortran language, whereas, the device code is written using special syntax that gives the programmer explicit control over the GPU hardware. In the next section, the relevant literature related to the GPU acceleration of FEA is discussed.

2.4 Literature: GPU-Accelerated FEA

The FEA of a structure typically starts with the preparation of a CAD model of the structure, created using specific pieces of software. This CAD model is then converted into a form suitable for the FEA in the pre-processing step. Before meshing, processes such as *Model-Simplification* may be performed, where the features of the model that are not important for the analysis are removed. Subsequently, a polygonal or polyhedral mesh of the model is generated to approximate the geometric domain of the structure. The process then enters the solver stage, where the elemental stiffness matrices are computed for all the elements and assembled to form a global system of equations in the form of a matrix and a vector. Next, the system of equations is solved by a direct or iterative method after the application of the proper boundary conditions. The literature has been grouped and discussed according to the different steps in the following subsections.

2.4.1 Pre-Processor

The acceleration of the pre-processor stage of FEA has been done in two steps in the literature: *Meshing* and *Model Simplification*. Meshing is the method of approximating domain of the structure by discretizing it into several small parts or elements. Figure 2.1 shows a mesh of triangular elements. A comprehensive study and categorization of several mesh generation techniques are provided by Ho-Le (1988). Model or shape simplification, on the other hand, is a method for the removal or suppression of features of the model that are not likely to influence the analysis result much but may add to the computation performed by a considerable amount. For example, Venkataraman et al. (2001) implemented an algorithm for automatic detection and removal of blends or fillets from the FE mesh based on the criterion that the radius of the blend should be sufficiently low according to the size of the finite element. Similar algorithms have been implemented on the CPU for small holes and bosses.

2.4.1.1 Meshing on GPU

Meshing can take a considerable portion of the total time taken by FEA and, hence, is suitable for implementation on GPU. All the parallel implementations of meshing on GPU in the literature makes use of the Delauney Triangulation (DT) method. The key idea behind parallel mesh generation in existing implementations is to divide the domain into several smaller sub-domains and apply standard sequential meshing techniques such as *Advancing Front Method* and *Delauney Triangulation* to each of the sub-domain in parallel. Efficient parallelization of the meshing process, even on the CPU, is still considered a challenging task (Ito et al., 2007). This task becomes even more challenging for GPU implementations, making it an open area of research with only few publications. Rong et al. (2008) first implemented parallel DT on GPU in two-dimensions. The authors achieved a speedup of 1.53× on an NVIDIA GeForce GTX8800 compared to the CPU implementation provided by the software *Triangle* on an Intel Core2 Duo processor at 1.86 GHz. Qi et al. (2013) implemented a constrained DT on GPU to achieve a speedup ranging from 14× to 28× tested on an NVIDIA GeForce GTX580 compared to the implementation on an Intel Core i7 CPU at 3.4GHz. Cao et al. (2014) first implemented DT on GPU in three-dimensions. Their implementation *gDel3D* on an NVIDIA GTX580 achieved a speedup of 6× to 10× approximately over CGAL 4.2, one of the fastest DT libraries on the CPU, running on an Intel i7 at 3.4 GHz. It should be noted that the implementation by Cao et al. (2014) computes a triangulation that is close, but not exact, to the actual DT. Another limitation of this work is the weak scalability of the implementation on different GPUs. The authors tested the implementation on four GeForce GPUs and one Tesla GPU. Results, however, indicated that by simply using a better hardware, no significant improvement in performance can be expected. An implementation scalable for the wide range of GPUs available, is still considered a challenge for the developers.

2.4.1.2 Shape Simplification on GPU

Model or shape simplification, although often performed manually, is another suitable candidate for parallel implementation. If properly applied, shape simplification can drastically reduce the run time of the FEA, with only negligible loss of accuracy in the final results. There are two primary ways of implementation. First, the simplification is performed directly at the CAD level before the process of meshing, whereas, in the second way, the simplification is performed after mesh generation. It is intuitive that the first process is more efficient than the second since a considerable amount of effort is saved while meshing complex geometries. In the literature, there are several works that have targeted this issue on the CPU (Thakur et al., 2009). However, no research efforts have been made to accelerate mesh simplification at the CAD level using GPUs. This is primarily because of the extra effort required to handle and manipulate the CAD model as compared to handling the FE mesh on GPU. The only work concerning simplification on GPU in the context of structural analysis is that by Hjelmervik and

Léon (2007), which uses the OpenGL paradigm. The authors used a parallel vertex removal method, where vertices are processed and removed in parallel, subject to the satisfaction of certain geometric or mechanical conditions. A speedup of approximately 8× was achieved on an NVIDIA 7800GT over the implementation on an AMD Athlon 4400+ CPU at 2.2 GHz. Later, Hjelmervik and Léon (2010) developed a flexible algorithm for shape simplification on IBM's *Cell Broadband Engine Architecture*, which is suitable for multi-core heterogeneous architectures in general. In this work, a maximum speedup of around 8× was achieved.

Additionally, mesh simplification has been studied extensively by the graphics community (Botsch et al., 2004, DeCoro and Tatarchuk, 2007, Boubekeur and Alexa, 2009). In the next section, we will discuss the progress in the acceleration of the solver step.

2.4.2 Solver

The literature (Georgescu et al., 2013, Cecka et al., 2011, Maciol et al., 2010) shows that there are several steps of FEA solver, such as the elemental matrix generation, assembly into global stiffness matrix, and solution of the linear system of equations, for large and complex domains that are time-consuming in nature. Although a single FEA, in itself can often be perfectly handled by traditional CPU implementations, challenges do arise in cases involving complex geometries and ultra-fine meshes or in cases where the FEA is coupled with another computational algorithm such as topology optimization (Ram and Sharma, 2017). One common strategy for reducing time is to perform FEA computations in parallel by exploiting the data-parallel nature (Georgescu et al., 2013). Due to the single-instruction-multiple-data (SIMD) structure of FEA, it is typically suited for shared-memory parallel architectures such as GPUs. As a result, a lot of research work has focused on GPU-based parallel implementations of different stages of FEA in the last decade. The solver stage of FEA can be divided into two major steps: the matrix generation step and the linear solver step. The next section discusses the literature on GPU-based matrix generation that includes elemental stiffness generation and assembly on GPU.

2.4.2.1 Matrix Generation

The first step in FEA matrix generation is computing elemental stiffness matrices, which can often computationally expensive for unstructured meshes due to repeated calculations for a high number of elements or high order of elements (Komatitsch et al., 2009). Thus, several studies have focused explicitly on the numerical integration aspect (Banaś et al., 2016, Knepley and Terrel, 2013) of FEA. This generation of elemental data can be done independently for all the elements, which makes this stage *embarassingly parallel* as reported by several researchers (Fu et al., 2014, Reguly and Giles, 2015). The first work dedicated to numerical integration on GPU is by Maciol et al. (2010) using the Gauss-Legendre Quadrature Method. The authors demonstrated the complete scalability of the numerical integration

process on GPU. This, in the context of GPU computing, implies that for a high enough number of finite elements, the execution time for a GPU with n multiprocessors will be n times smaller than for a GPU with one multiprocessor. One limitation of this study was that it was conducted entirely in single-precision, which can pose serious problems for convergence in the solver stage. This is specifically true for iterative solvers such as conjugate gradient (CG) solver, which are known to be notoriously sensitive to round-off errors. A maximum speedup of 20× was achieved for third-order approximation on an NVIDIA GeForce GTX8800 as compared to an AMD X2 at 2.4 GHz. The authors also concluded that the massive amount of parallelism could not be fully realized due to the insufficient memory resources in individual threads. This finding was later supported by Dziekonski et al. (2012a), where several strategies on efficient generation and assembly of large finite-element matrices were presented while maintaining the desired level of accuracy in numerical integration. The authors used higher-order curvilinear elements, which made the integration step more computation intensive. Again, the method of GQ was used for analysis. The authors also presented a strategy to exploit the symmetry of local matrices in the integration step. A speedup of 2.5× was achieved on an NVIDIA Tesla C2075 over two 12-core AMD Opteron 6174 at 2.2 GHz. Later, Banaś et al. (2014, 2016) addressed the problem of implementing numerical integration that is portable across several GPU architectures and different orders of approximation. This is in general difficult to achieve due to the vastly varying memory size, memory hierarchies, and computational resources available to the programmer across different GPU vendors such as NVIDIA, AMD, and Intel. The OpenCL implementation by Banaś et al. (2014) achieved a maximum speedup of 4× tested on four GPUs with different architectures (NVIDIA GeForce GTX 580, Nvidia Tesla M2075, AMD HD5870 and AMD HD7950). The performance analysis shows 302-372 GFlops for NVIDIA and 111-172 GFlops for AMD. These values are, however, only 19%-23.5% and 4%-6% of the peak Flops for the particular GPUs from NVIDIA and AMD, respectively. This clearly implies a large scope for improvement in performance, specifically for AMD GPUs. The portable OpenCL implementation by Banaś et al. (2016), targeting numerical integration for only first-order approximation, achieved a maximum speedup of approximately 9× on a Tesla K20m, compared to an Intel Xeon E5 2620 CPU. The authors presented details on several optimization aspects for implementations on different target hardware, while also indicating the factors limiting the performance for different problem types on different architectures.

In addition, many works concentrating on applications of FEA have also implemented numerical integration on GPU (Reguly and Giles, 2015, Komatitsch et al., 2009, Schmidt and Schulz, 2011). However, in these cases, no relevant details on the implementation of numerical integration were provided. The known implementations of numerical integration on GPU, along with the methods, orders of approximation used, and the associated speedup values are summarized in Table 2.1.

Following elemental stiffness generation, another important challenge is the assembly of the elemental stiffness matrices into a global stiffness matrix, which can become computationally expensive (Georgescu

Table 2.1: Summary of numerical integration on GPUs. C stands for CUDA and O, for OpenCL implementations. S and D stand for single precision and double precision implementations.

Publication	Method	Order of Approx.	Hardware	Speedup
Maciol et al. (2010)	Gauss-Legendre Quadrature(C)	1 – 7	Tesla C2075	3.5 × −19.8×(S)
Dziekonski et al. (2012a)	Gauss Quadrature(C)	5 – 14	Tesla C2075	1.2 × −2.5×(D)
Banaś et al. (2014)	Gauss Quadrature(O)	2 – 7	GeForce GTX580 Radeon HD5870	4×(max.)(S)
Banaś et al. (2016)	Gauss Quadrature(O)	1	Tesla K20m	9× (max.)(D)

et al., 2013) for a large mesh or in case of non-linear FEA where assembly needs to be performed repeatedly. After the assembly is performed, the implementation of boundary conditions and the solution of linear equations have also been implemented on GPU (Altinkaynak, 2017, Fu et al., 2014). Some studies in the literature have investigated matrix-free methods for GPU-based FEA (Challis et al., 2014, Schmidt and Schulz, 2011). These methods work on the key principle of reducing memory access and storage by introducing redundant computation. Unlike the elemental stiffness matrix generation stage, the other stages of FEA cannot be broken down into small independent chunks of computation due to the dependencies among the nodes and elements. New and innovative approaches that are often radically different are required (Markall et al., 2013, Altinkaynak, 2017, Kiss et al., 2012) in order to extract a good performance.

Among the very early studies on GPU-based implementation of FEA matrix generation, Bolz et al. (2003), Rodríguez-Navarro and Susín Sánchez (2006) used simplified parallel implementations, where expressions were derived for each NZ entry in the global stiffness matrix separately and the computations were performed in parallel on GPU. Although this implementation provided some gain in performance, it was not possible to derive such expressions for many real-world applications. After the launch of NVIDIA's CUDA in 2007, more sophisticated and generalized implementations were being reported with significant performance gains. Komatitsch et al. (2009) implemented numerical simulations of seismic wave propagation caused by earthquakes on an NVIDIA GPU. In this implementation, each thread block was assigned to one 125-noded element of the finite element mesh. The inherent data race was countered by coloring the elements and computing the colors sequentially. A similar coloring strategy, targeting the assembly stage on GPU, was also reported by Cecka et al. (2011), where a detailed study of different ways to perform an assembly for unstructured 2D mesh was presented. An elemental subroutine was used for calculating the elemental matrices, where each thread was designated different tasks for different implementation strategies. The data race associated with assembly was handled by allocating each NZ entry to one compute thread or by using a coloring approach identical to the implementation by Komatitsch et al. (2009). The coloring approach was found to be under-performing due to excessive accesses to the global memory of GPU. For lower-order elements, the authors recommended the use of shared memory to store the elemental data and assigning each thread to compute an NZ entry of the global stiffness matrix. For higher orders, a single-thread-per-element strategy was suggested. Fu

et al. (2014) discussed the acceleration of FEM on GPU for higher-order elements. Again, the thread allocation was done by assigning a single thread to one element of the FE mesh. This implementation, however, involved redundant computation as the elemental stiffness matrices were calculated on the fly, and were not stored on GPU memory. Markall et al. (2013) discussed assembly by segmentation into two separate approaches, namely the `addto` method and the local matrix approach (LMA). The standard assembly approach followed on the CPU was termed as the `addto` method of assembly. The LMA, on the other hand, is a matrix-free method that aims to alleviate GPU-related implementational difficulties. The idea of these implementations originated from the methods followed in higher-order FEM (Cantwell et al., 2011), where similar *scatter and multiply* methods were implemented for performing multiplication on the global stiffness matrix. It was concluded that the LMA is suitable for GPU implementations, whereas `addto` method is more suitable for CPU implementations. Kiss et al. (2012) took this method of assembly another step further, in that the elemental matrices were computed whenever required instead of being stored on GPU. This low-storage computation-intensive implementation was shown to be better suited to many-core systems for large data sets. However, Fu et al. (2014) later demonstrated, that the results of comparative studies that used this approach were worse than those for the traditional assembly into complete global stiffness matrix for 3D grids. The authors presented a novel algorithm for assembly using a patch-based division of the entire mesh with the aim to alleviate several difficulties of GPU-based assembly. Dziekonski et al. (2012b, 2013) presented an efficient method for generating the global stiffness matrices in FEA in the field of computational electromagnetic using both single and multi-GPU setups. The key idea behind this strategy was to assemble the sparse global stiffness matrix into the coordinate format (COO) in the first step and to transform it into the compressed sparse row (CSR) format by removing the duplicate entries in the second step. Among more recent works, Dinh and Marechal (2016) studied a real-time FEM implementation on GPU, in which they performed a sorting-based implementation of parallel global assembly. In this implementation, after executing the elemental routine, the data was accumulated into an unsorted COO format with duplicate entries. Next, the entries were sorted using parallel radix sort and reduced to form the global stiffness matrix. An implementation based on the principle of dividing GPU assembly with standard sparse formats was presented by Sanfui and Sharma (2017a). The authors used structured meshes with brick elements to demonstrate the advantage of workload division at the assembly stage. The implementation divided the assembly operation into a separate symbolic kernel and a numeric kernel. Later, Zayer et al. (2017) accelerated the assembly of sparse matrices by modifying the assembly stage as a matrix-matrix multiplication with the aim to remove any CPU or GPU-based preprocessing. This approach enabled them to reduce the storage and movement of data on GPU. Among more recent works, Kiran et al. (2018) presented a warp-based assembly approach for hexahedral elements in single precision where the numerical integration and assembly were performed in the same kernel. An implicit finite element model with cohesive zones and collision response was accelerated using CUDA by Gribanov et al. (2018). For

handling the race condition in assembly, instead of coloring the elements, the `atomicAdd` function of CUDA toolkit was used. Later, Sanfui and Sharma (2019) implemented FEA on GPU by utilizing the symmetry of the elemental and global stiffness matrix. Numerical integration and assembly operations were implemented on GPU to achieve a speedup of approximately 2× over the standard implementation that computed the entire matrix on GPU. It should be noted here that in addition to traditional FEM implementations, GPU implementations of matrix generation for isogeometric analysis (IGA) have been reported in the literature (Woźniak et al., 2014, Woźniak, 2015).

An important issue related to the implementation of FEA on GPU is the handling of race condition. A race condition or data race is a situation where the output of one or more operations becomes dependent on the order or timing of a number of uncontrollable events. This can occur in the assembly step in which different thread blocks running in parallel are assigned to different elements of the FE mesh and certain threads in a number of thread blocks try to perform addition on the same entry of the global stiffness matrix at the same time. The final output of the stiffness entry will depend on the order of read, addition, and write by all the parallel threads causing a race condition as illustrated in Figure 2.5. The CUDA API provides a remedy called the atomic operations, which is also depicted in Figure 2.5. The primary use of an atomic operation is to keep a particular memory location locked until a particular operation is complete (Kirk and Wen-mei, 2012). This is generally undesirable, as the serialization of the execution of parallel threads results in reduced performance. A number of strategies have been implemented in the literature to avoid race condition. Arguably, the most efficient one is the mesh coloring method (Komatitsch et al., 2009, Cecka et al., 2011). In this method all the elements of the FE mesh are colored in such a way that no two elements with the same color share the same node. The assembly operation is then performed for all the colors in a sequential manner. Such a coloring scheme is shown in Figure 2.6, where a mesh of cubic hexahedron elements is colored using eight different colors in a deterministic manner. If, however, the mesh becomes un-structured, a stochastic coloring algorithm needs to be employed. Other strategies to handle race include assigning each NZ to one thread (Cecka et al., 2011), atomic operations (Dziekonski et al., 2012b) and assembly-free methods (Markall et al., 2013).

The coloring method for FEM was first introduced several decades ago on vector computers (Hughes et al., 1987, Farhat and Crivelli, 1989, Adams and Ortega, 1982). On GPU architecture, Komatitsch et al. (2009) made use of such a scheme to efficiently handle assembly operations over several nodes in parallel. Finding the minimum number of colors for an unstructured mesh is also a complex problem in itself. Several heuristics exist in the literature that can help determine a number based on the implementation. Cecka et al. (2011), however, analyzed a number of such heuristics and concluded that finding the optimum coloring of a mesh is not of the primary concern and that almost any standard heuristics available in the literature can be sufficient. Other two methods of handling the data race are the atomic operations provided by the CUDA API, and a matrix-free or LMA-based approach. Another

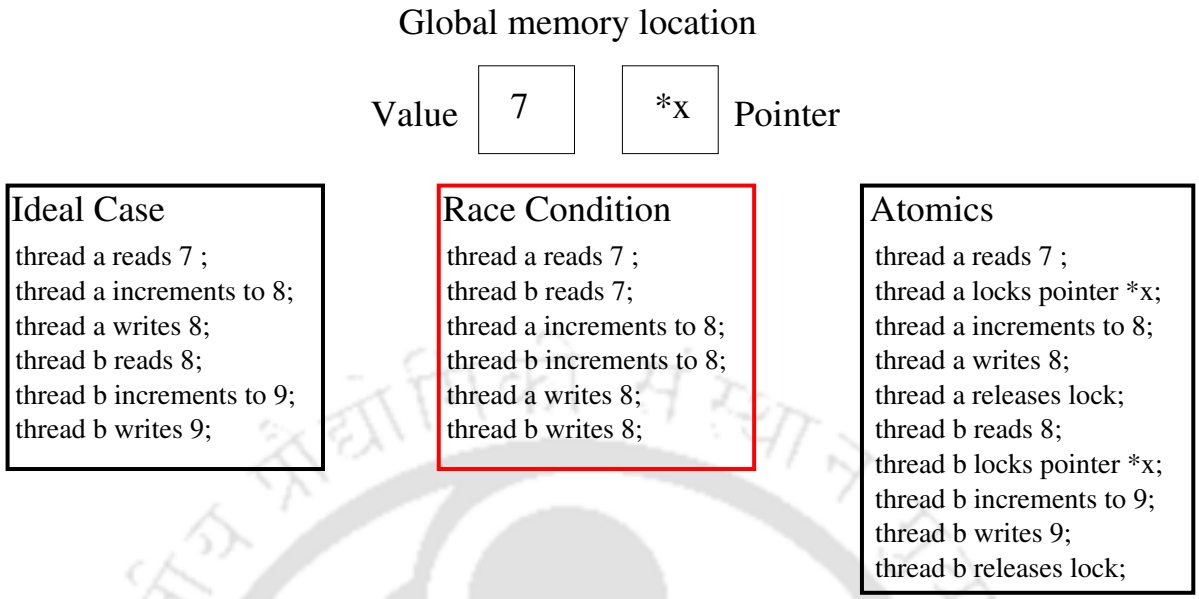


Figure 2.5: Race Condition and Atomics

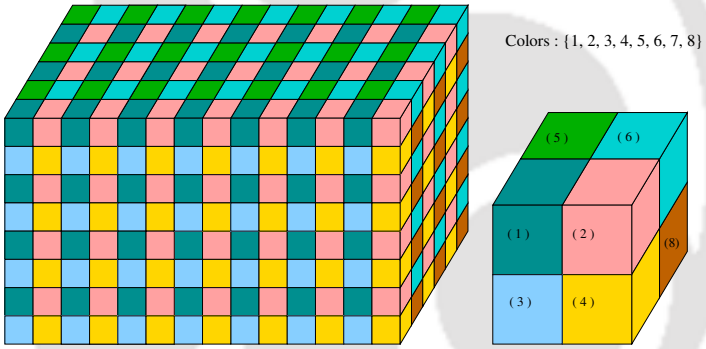


Figure 2.6: Mesh Coloring

strategy with a high degree of redundant computations is allocating one thread to compute one NZ of the global stiffness matrix. This automatically solves the issue of the race condition. The general consensus in the GPU-computing community is that atomics should, in general, be avoided and be only used as the last resort. However, several works have targeted assembly with the help of atomic operations. The details of these implementations has been presented in Table 2.2.

Table 2.2: Summary of FE matrices assembly on GPUs. Here, C and O stand for CUDA and OpenCL implementations, respectively. CO stands for both CUDA and OpenCL. HW stands for hardware used. S and D stand for single and double precision, respectively.

Publication	Method	Size	Application	HW	Speedup	Race
Filipovic et al. (2009)	addto (C)	(60K)	FEA (3D)	GTX 280	15× (S)	Atomics
Cecka et al. (2011)	NZ by NZ (Shared) (C) NZ by NZ (Global) (C)	(4.8M)	FEA (2D)	GTX8800 Tesla C1060	35× (S) 10 − 20× (S)	Coloring Atomics
Dziekonski et al. (2012b)	addto(CO)	(30K)	FEA (3D)	Tesla C2075	81× (D)	-
Markall et al. (2013)	addto(CO) LMA (CO)	(29K)	FEA	GTX480	8× (D) 10× (D)	- -
Challis et al. (2014)	LMA (C)	(4M)	Top. Op. (3D)	C2050 C2070 M2070 S2050	-	-
Schmidt and Schulz (2011)	Matrix-free (C)		Top. Op. (3D)	M2050	22 − 58× (Combined) (D)	-
Herrero et al. (2013)	addto (C)	(13K)	Top. Op. (2D)	C2070	3.3× (Combined) (D)	-
Krishnan (2013)	Matrix-Free (C)		Top. Op. (3D)	GTX 480	5 − 10× (Combined) (D)	-
Wadbro and Berggren (2009)	Matrix-Free (C)	(5K)	Top. Op. (2D)	GTX 8800	-	-
Kiss et al. (2012)	Matrix-Free EbE (C)	(30M)	FEA	GTX 590	6 − 99× (Combined) (D)	-
Reguly and Giles (2015)	addto (C) Matrix-free (C)	(16M)	FEA (2D)	Tesla C2070	5× (D) 10× (D)	Coloring -
Komatitsch et al. (2009)	addto (C)	(73K)	E. S. (3D)	GTX 280 GTX 8800	15× (Combined) (S)	Coloring
Martínez-Frutos et al. (2015)	Matrix-free EbE (C)	(2M DOF)	FEA (3D)	Tesla K40m	14× (D)	Coloring
Martínez-Frutos and Herrero-Pérez (2015)	Matrix-Free (C)	(2.5M DOF)	FGFEA (3D)	Tesla C2070 Quadro 4000	20× (D) 12× (D)	
Cai et al. (2013)	Matrix-free NbN (C)	(4.1M Elements)	FEA	GTX 580	170× (max.) (D)	

2.4.2.2 Data Structures for FEA on GPU

Another important factor for consideration in GPU-acceleration of FEA is the use of sparse format for storing the global stiffness matrix. A specialized sparse format only records the NZ entries of the sparse matrix in a particular manner, thereby reducing the storage requirements by a substantial amount. In such cases, the trade-off is that accessing individual elements become more complex. The problem is more prominent on GPU since access patterns become highly irregular on GPU, resulting in the loss of data coalescence (Kirk and Hwu, 2010), if not handled carefully. It is worth mentioning here that dense linear algebra maps readily into GPU architecture and is devoid of many complications arising in sparse linear algebra. Three of the most common sparse storage formats in the literature are COO, CSR, and ELL. Bell and Garland (2008) presented an exhaustive study on a variety of sparse data formats with different storage requirements, computational characteristics, and methods of accessing and manipulating entries of the matrix for use on GPU. The authors also discussed their applications and associated trade-offs for storage and sparse matrix-vector multiplication (SpMV), which is one of the most important parts of sparse linear algebra on GPU.

Based on utilization on GPU, these sparse formats can be broadly categorized into three types. The first type supports efficient modifications (LIL, COO, etc.), and the second type is the one that supports efficient data access and matrix operations (CSR, ELL) on GPU. The third type of sparse format is one that is application-specific and utilizes certain properties of the matrices in the particular application (in this case, FEA) to further reduce the storage space or attain some advantages for the computation. Markall et al. (2013) have demonstrated that data structures that contain redundant data but enable coalesced access are suitable for many-core architectures, whereas data structures that are redundancy-free, work better on multi-core architectures. It must be noted here that using a dense format for storage on GPU for even low to medium scale problems becomes infeasible, except when using an assembly-free method such as the LMA (explained in section 2.4.2.3). To illustrate, let's take the example of a cubic FE mesh with 20 nodes each in $x-$, $y-$, and $z-$ directions with three degrees of freedom per node. Storing the global stiffness matrix for this system in dense format requires in excess of four gigabytes of data if we use double precision variables. Such a massive requirement of global memory automatically renders the format infeasible for GPUs. On the other hand, a sparse format such as CSR can store the same matrix in approximately 20 megabytes. This reduced storage, however, comes at the cost of several implementation difficulties on GPU, as discussed before. Bell and Garland (2008) provided an excellent study that can help GPU programmers to choose the correct format from a plethora of options available. The COO format, although the simplest and probably the easiest to implement, is generally avoided on GPU due to complications in data accesses during matrix operations that can lead to random accesses and write conflicts, thereby seriously hampering the performance (Reguly and Giles, 2015). The most popular choice of format for FEA on GPU seems to be the CSR format along with

Table 2.3: Summary of data-structures on GPUs

Sparse Format	Publications
CSR	Dziekonski et al. (2012b), Li and Saad (2013), Reguly and Giles (2015), Sanfui and Sharma (2017b) Cevahir et al. (2009, 2010)
ELL	Reguly and Giles (2015), Sanfui and Sharma (2017b)
BCRS	Buatois et al. (2009)
JAD	Li and Saad (2013)
DIA	Li and Saad (2013)
MCTO	Tian et al. (2013)
JDS	Cevahir et al. (2009, 2010)
HYB	Cevahir et al. (2010)

its many variants, as seen in Table 2.3. Buatois et al. (2009) implemented a Block Compressed Sparse Row (BCSR) storage format, which enables register blocking in the CSR format in order to reduce the required memory bandwidth. Georgescu and Okuda (2010) also implemented the BCSR format with blocking techniques. Tian et al. (2013) implemented the third type of sparse format called Modified Compile Time Optimization (MCTO) format. However, instead of reducing the storage requirements, this format helped in reducing execution times of iterative solution of FEM equations, especially for matrices with uneven row lengths. A summary of the sparse formats used in the literature can be found in Table 2.3.

Single and double precision computations on GPU Implementations that involve sparse or dense linear algebra benefit from the use of double precision (FP64) arithmetic, instead of single precision (FP32). The use of single precision in the linear solver stage can cause several complications ranging from degrading the accuracy of the solution and increasing number of iterations for convergence. Needless to say, the benefits of using double precision come at the expense of increased computation cost. On modern multi-core architectures, the single precision performance is close to twice as fast compared to double precision performance. This, however, is not necessarily true in the case of many-core architectures such as GPUs and the performance usually depends on the sub-class of GPU. Before GPUs that can handle double precision calculations were available, Göddeke et al. (2005) implemented a *mixed-precision* approach, where computations were performed on GPU in single precision with double-precision error corrections from the CPU at regular intervals. Using this algorithm, authors achieved a speedup of approximately 2× on an NVIDIA Geforce GTX6800 compared to a CPU implementation on an Opteron 250 in double precision while maintaining the same accuracy in the final results. A few years later, with the introduction of double precision support in NVIDIA compute capability 3.0, Göddeke et al. (2007) demonstrated that a mixed-precision solver still provided higher-level performance than the newly introduced double precision solver. Komatitsch et al. (2009), by comparing both single and double precision implementations on the CPU with a single precision implementation on GPU, demonstrated that the use of FP64 is not required for the particular problem. The choice of precision for major works in the literature are summarized in Table 2.4.

Table 2.4: Summary of precision of computations on GPU

Precision	Publications
Single	Cecka et al. (2011), Buatois et al. (2009), Wadbro and Berggren (2009), Komatitsch et al. (2009), Markall et al. (2013), Maciol et al. (2010), Dziekonski et al. (2012b), Filipovic et al. (2009), Wu and Heng (2004)
Double	Schmidt and Schulz (2011), Krishnan (2013), Li and Saad (2013), Banaś et al. (2014), Reguly and Giles (2015), Georgescu and Okuda (2010), Krawezik and Poole (2009)
Mixed	Göddeke et al. (2005), Göddeke et al. (2007), Cevahir et al. (2009)

2.4.2.3 Solution of linear system of equations on GPU

It has been reported that solving the linear system of equations is the most time-consuming step not only in the solver step but also in the entire FEA (Cecka et al., 2011). The global stiffness matrices of most structural analysis problems have certain properties such as symmetry, positive-definiteness and sparsity. Naturally, a solution technique that exploits all these properties is the preferred choice for the linear solver. It is worth mentioning here that there exist certain scenarios where the symmetry and/or positive-definiteness of the global matrix can be destroyed in due course, and thus, special care needs to be taken. However, this thesis only focuses on linear elasticity or similar problems where these properties are kept intact.

Sparse direct solvers are often the choice in the industry, the scale of the problems in most cases being small to medium in size (Georgescu et al., 2013). These solvers have several advantages; for example, they can handle extremely ill-conditioned matrices and usually do not pose convergence problems. In addition, factorization-based solvers have the added advantage of reusing the same factorization in the case of multiple load vectors. Sparse direct solvers, however, also have a few drawbacks, such as the inability to provide any useful result until the process is complete, greater storage requirements due to the coefficients of the matrices changing between steps, etc. (Lucas et al., 2010, George et al., 2011).

An iterative solver, on the other hand, is less reliable than a direct method, in general, and is often largely dependent on the choice of the initial guess and other parameters. However, one useful advantage of using this solver is that it can provide partial results within only a few iterations, which can be useful in many practical scenarios or for debugging purposes. Moreover, a known approximation of the results may be exploited to achieve faster convergence. They are the choice of solution methods in the case of large scale problems (Cecka et al., 2011) or in problems, where accuracy is of a lesser concern. Further, in case of iterative solvers, as seen in the subsequent sections, the key difficulty lies in preconditioning the stiffness matrix (Li and Saad, 2013) on GPU.

Considering the amount of research on linear solvers for GPUs, several resources are available such that a developer can rely on libraries without having to go through much trouble to tackle this step. Most GPU-accelerated software packages for performing FEA target only the step of matrix solver. Examples include ABAQUS, ANSYS Mechanical, MSC Nastran, among others. The most time-consuming parts of an iterative solver are the SpMV and the preconditioner. The SpMV process, in general, is easy to

parallelize due to the data parallel structure. There are a number of standard libraries that provide optimized SpMV for a large range of data formats, such as CUBLAS, CUSPARSE, and CUSP. However, as mentioned before, in more real-world applications, preconditioning becomes a stringent requirement in the analysis. Unfortunately, a number of efficient preconditioners have a more serial structure, making them difficult to implement on GPU architecture.

Among the different types of iterative solvers, CG methods are preferred due to their ability to handle positive definite symmetric matrices. The first implementation of CG on GPUs dates back more than a decade (Bolz et al., 2003, Krüger and Westermann, 2003), at a time when modern programming paradigms such as CUDA and OpenCL were not available. The authors used the Open Graphics Library (OpenGL) for acceleration using shaders on GPU. A couple of years later, Göddeke et al. (2005) implemented a mixed-precision approach for hybrid CPU-GPU environment to achieve a speedup of approximately 2× on an NVIDIA GeForce GTX6800 over an AMD Opteron 250 at 2.4 GHz. This is the first instance of successful implementation of CG in double precision on a GPU. The first use of CG solver for unstructured meshes was done by Buatois et al. (2009). Techniques such as Block Compressed Row Storage and register-blocking were implemented to attain a speedup of more than 12× over the SpMV using Intel MKL library. Subsequently, several works implemented solvers on multiple GPUs and GPU clusters using unstructured meshes (Cevahir et al., 2009, Georgescu and Okuda, 2010). Cevahir et al. (2010) implemented an un-preconditioned CG solver for an un-structured mesh on a cluster of GPUs to achieve high scalability using hypergraph-partitioning. A speedup of 17× was achieved on an NVIDIA Tesla S1070 GPU over an AMD Opetron CPU at 2.4 GHz. Georgescu and Okuda (2010) achieved a similar speedup of approximately 18× on NVIDIA GeForce GTX 280 compared to an Intel Core i7 975 at 3.2 GHz.

Remacle et al. (2016) presented a finite element scheme using spectral elements for solving elliptic problems on unstructured meshes using OCCA, an open-source unified library for programming multi-core and many-core architectures. The authors drew an interesting conclusion that, unlike majority of finite element implementations, it is more efficient to recompute several quantities when required rather than to store them on the global memory due to the memory-bound nature of GPU computation. Abdelfattah et al. (2016) presented an optimized approach for performing SpMV on block-sparse matrices for multi-component PDE based applications. In this work, the authors have shown the design ideas of KBLAS, an open-source library for dense matrix-vector multiplication, on smaller blocks of a sparse matrix. This work also introduces a new load-balancing technique based on dividing longer rows of the matrix into smaller parts. The implementation was tested on several matrices from real-world applications, which outperformed state-of-the-art libraries such as CuSparse and Magma.

Preconditioning the CG solver can drastically improve its performance. Factorization-based preconditioners such as Incomplete LU (ILU), Incomplete Cholesky (IC), Successive Over-Relaxation (SOR) and Systematic Successive Over-Relaxation (SSOR) are usually the choice of preconditioners for FEA

on CPU (Georgescu et al., 2013). These preconditioners, however, being mostly serial in nature, are difficult to implement on GPU. Wang et al. (2009a) and Li and Saad (2013) implemented these types of preconditioners by dropping all elements from the factorization, except the ones residing at the blocks along the main diagonal. This type of preconditioning is termed as *Block-Jacobi Preconditioning*, which, although more efficient than a diagonal preconditioner, performs poorly in comparison to a full IC or ILU preconditioner. Another way of extracting performance from factorization-based preconditioners on GPUs is to use domain decomposition techniques, along with a graph coloring algorithm. Since there are no dependencies between the partitions of same color, they can be run in parallel. A multi-colored SSOR-based preconditioner was implemented by Li and Saad (2013). A fully optimized implementation by the authors achieved a speedup of only $2 \times -3\times$ on an NVIDIA Tesla C1060 over an Intel Xeon E5504 at 2 GHz. In the white-paper (Naumov, 2011), NVIDIA demonstrated the use of CUSPARSE and CUBLAS libraries in order to achieve a $2\times$ speedup using ILU and IC preconditioned iterative methods when tested on an NVIDIA Tesla C2050 over an Intel Core i7 950 at 3.07 GHz. Another powerful type of preconditioners is the Multigrid type. The Geometric Multigrid Preconditioner, although used primarily for the field of Computational Fluid Dynamics (CFD), has also been implemented for FEA on GPU. Geveler et al. (2011) presented such an implementation of Geometric Multigrid for FEA to obtain a speedup of $5 \times -10\times$ on an NVIDIA GTX 285 over an Intel Core i7 920 at 2.66 GHz. Algebraic Multigrid preconditioners can, on the other hand, be more easily applied to FEA and are not limited to any specific implementations (Georgescu et al., 2013). Another useful type of preconditioner is the Sparse Approximate Inverse type (SPAI), which has been implemented on GPU by Helfenstein and Koko (2012) to achieve a maximum speedup of $10\times$ on an NVIDIA Tesla T10 over an Intel Xeon Quad-Core at 2.66 GHz.

Most of the research on the implementation of sparse direct formats on GPU rely on decomposing the matrix into dense blocks, which can be processed in parallel. These methods are called multi-frontal methods. These are known to provide good performance for GPU implementations. Multi-frontal Cholesky was implemented successfully by both Lucas et al. (2010) and George et al. (2011). The first GPU acceleration of the ANSYS sparse direct solver was presented by Krawezik and Poole (2009), who used a multi-frontal Cholesky based implementation. Speedups of $4\times$ for mixed precision and $2.5\times$ for double precision were achieved on an NVIDIA Tesla C1060 over two Intel Xeon 5335 at 2.00 GHz each. Sao et al. (2014) implemented the first hybrid MPI + OpenMP + CUDA implementation of a distributed memory-based sparse LU factorization. Summary of the matrix solution methods used in major existing works can be found in Table 2.5.

Table 2.5: Summary of matrix solver on GPUs. Here, C and O represent CUDA and OpenCL implementations. D, S and M stand for double, single and mixed precision implementation.

Publication	Method	Size	Preconditioners	HW	Speedup
Schmidt and Schulz (2011)	Matrix-Free CG(C)		-	M2050	$22-58\times$(Combined)(D)
Buatois et al. (2009)	PCG(C)		Jacobi	ATI X1900XTX	$12\times$(S)
Krishnan (2013)	Matrix-Free CG(C)		-	GTX 480	$5-10\times$(Combined)(D)
Wadbro and Berggren (2009)	Matrix-Free CG(C)	(5K)	-	GTX 8800	-
Markall et al. (2013)	CG (CO)	(29K)	-	GTX480 (D)	$8-10\times$(S)
Kiss et al. (2012)	PCG (C)	(30M)	Jacobi	GTX 590 (D)	$6-99\times$(Combined)(D)
Sao et al. (2014)	Sparse LU(C)		-	M2090, C2050	$2\times$(D)
Li and Saad (2013)	PCG(C) GMRES(C)		L-S polynomial ILUT	TESLA C1060	$7\times$(D) $4\times$(D)
Wang et al. (2009b)	GMRES(C)		Block ILU	GTX280	$20\times$(D)
Wu and Heng (2004)	CG		-		$1.5\times$(S)
Reguly and Giles (2015)	PCG(C)	(16M)	Jacobi, SSOR	C2070	$3-4\times$(D)
Göddeke et al. (2005)	CG	(1M)	-	6800	- (M)
Krüger and Westermann (2003)	CG	(4M)	-	ATI 9800	$12-15\times$(S)
Cevahir et al. (2009)	PCG(C)	(1.5M)	-	8800 GTS	$5\times$(M)
Georgescu and Okuda (2010)	CG (C)		-	280GTX, 295 GTX	$22\times$(D)
Cevahir et al. (2010)	CG(C)	(1.5M)	-	Tesla	$14\times$(D)
Naumov (2011)	PCG, BiCGStab(C)	(1.5M)	ILU/IC	C2050	$2\times$(D)
Geveler et al. (2011)	PCG(C)	(4M)	GMG	GTX 285	$2-15\times$(D)
Helfenstein and Koko (2012)	PCG(C)	(2M)	SSOR	Tesla T10	$1.4-10\times$(D)
Krawezik and Poole (2009)	multi-frontal(C)	(2.5M DOF)	-	Tesla C1060	$3-4\times$(D)
Lucas et al. (2010)	multi-frontal(C)		-	C1060	$6\times$(S)
George et al. (2011)	multi-frontal(C)	(1.5M)	-	Tesla T10	$10-25\times$(M)
Remacle et al. (2016)	PCG(CO)	(50M DOF)	Additive Schwartz	GTX 980	$10\times$(S)
Martínez-Frutos et al. (2015)	Matrix-Free CG(C)	(2M DOF)	-	Tesla K40m	$14\times$(D)
Martínez-Frutos and Herrero-Pérez (2015)	Matrix-Free PCG(C)	(2.5M DOF)	-	Tesla C2070	$20\times$(D)
Cai et al. (2013)	Matrix-free NbN(C)	(4.1M El)		GTX 580	$170\times$(max.)(D)
Abdelfattah et al. (2016)	SpMV(c)	(2M mat)		Tesla K40c	$30\times$(max.)(D)
Tao et al. (2015)	SpMV(C)	(5.1M mat)		GTX Titan	$5\times$(max.)(D)

2.5 Literature: GPU-Accelerated Topology Optimization

2.5.1 GPU-based SIMP

The SIMP method is one of the important techniques of topology optimization in the literature. Starting from the landmark 99 line code by Sigmund (2001), several implementations of the SIMP method have been made available in the literature for reducing its computational complexity. One of the most successful techniques in accelerating SIMP method on GPU is the use of adaptive mesh refinement technique (Maute and Ramm, 1995, De Troya and Tortorelli, 2018). In this technique, instead of adhering to a regular fixed grid throughout the topology optimization, a series of refinements and coarsening is applied on the meshes adaptively with the purpose of achieving smoother structures while reducing the number of design variables. In this context, there are implementations where meshes of different granularity are used for topology optimization and FEA (Nguyen et al., 2010). Another common technique is using reduced-order modeling, which involves mapping a high-dimensional space to a lower-dimensional space by using a reduced basis. This technique has been successfully implemented in different studies to reduce the computational complexity of the conventional SIMP-based topology optimization (Xiao et al., 2020, Gogu, 2015). Kim et al. (2012) implemented topology optimization using an approach where the variables that get quickly converged are reduced from the subsequent optimization iterations. This significantly reduced the computational complexity of the application. Another implementation based on the reduction of variables was performed by Yoo and Lee (2017). The authors implemented a variable-grouping method to reduce the number of design variables and, subsequently, the computational complexity. The researchers recognized that the variables can be grouped according to the histogram of the multiplication of sensitivity and density variable. Moreover, a dual-layer element approach was used to further reduce the execution time of the application. For reducing computational complexity of SIMP method, the multigrid methods have been found to be successful in the literature. Both geometric multigrid (GMG) and algebraic multigrid (AMG) methods work by accelerating the convergence of the linear system by solving the problem on a coarse grid. GMG methods require the use of physical meshes, whereas AMG methods perform the coarse grid computations at the algebraic level. Both GMG (Amir et al., 2014, Aage et al., 2015) and AMG (Aage et al., 2017) methods have been used in the literature for solving topology optimization problems. Among the more recent works, Liao et al. (2019) implemented a three-step acceleration method for reducing the computation time of SIMP method. All these three steps involved modifications at the algorithm level. By combining multilevel meshes, the initial value-based preconditioned conjugate gradient (PCG) method, and a local update strategy, the computational cost was significantly reduced (by 35-80%). More recently, Wang et al. (2020) combined multilevel meshes, the multigrid conjugate gradient method, and a local update strategy to reduce the computational complexity of isogeometric topology optimization.

Parallel computing has been implemented in the literature for reducing the computation time of large-scale SIMP-based topology optimization since very early times (Borrvall and Petersson, 2001, Mahdavi et al., 2006). Large-scale SIMP-based topology optimization is computationally expensive, and at the same time, suitable for massively parallel implementation (Deaton and Grandhi, 2014). Thus, as expected, a large number of research has focused on the parallel-computing aspect of SIMP. Different parallel setups have been used in the last two decades to accelerate different topology optimization methods including SIMP that range from cray parallel computers (Borrvall and Petersson, 2001) using message passing interface (MPI) (Sharma et al., 2011, 2014, Sharma and Deb, 2014) in the early days to multi-core Intel processors using OpenMP (París et al., 2013).

Several other studies have investigated FEA, specifically in the context of GPU-based topology optimization. These works can be broadly classified into two categories: assembly methods and matrix-free methods. Both of these categories imply an iterative method, specifically conjugate gradient (CG), or one of its variations. The assembly method is the same as the conventional FEA assembly (Zegard and Paulino, 2013), whereas, the matrix-free method is an assembly-free approach (Martínez-Frutos et al., 2017), in which the entire global stiffness is never formed explicitly and all the computations are performed at the element level. Apart from these categories of iterative solvers, a third category of direct solvers is also used to accelerate topology optimization on GPU (Duarte et al., 2015). Both assembly-based iterative solvers and direct solvers face the same issue of high memory requirements, which is exacerbated for direct solvers. For large-scale problems, despite sparse representation, the high memory requirements make matrix-free iterative methods a viable option. These methods reduce the amount of memory required at the cost of increasing the computational cost. This is the reason the matrix-free CG is the most widely used method (Deaton and Grandhi, 2014) in the literature for large-scale problems. Wadbro and Berggren (2009) implemented an element-by-element approach of matrix-free preconditioned conjugate gradient (PCG) for solving a heat transfer-based topology optimization problem with two different materials on GPU. A comparison of the execution times of sequential and parallel CPU implementations revealed speedups of up to 20×. Later, Schmidt and Schulz (2011) implemented topology optimization for 3D structures on GPU using matrix-free CG. For thread assignment, a node-by-node approach was adopted, where each GPU thread was assigned to one node of the FEA mesh. By effectively using shared memory, special attention was given to the procedural generation of matrix-vector product on GPU. Taking advantage of the regular grid, the authors calculated the elemental stiffness matrix only once and stored it on the constant memory to make it available for all the threads to access. Challis et al. (2014) implemented the level-set method on GPU for three-dimensional topology optimization problems with millions of elements. The finite element computations in this work were accelerated using an assembly-free approach with a combination of Thrust library operations and custom GPU kernels.

A similar strategy was followed by Martínez-Frutos and Herrero-Pérez (2016), who implemented a

large-scale and robust implementation of topology optimization on a GPU. A matrix-free PCG solver was used to solve the system of equations. In order to find robust optimal designs, which would be less sensitive to variations in the design variables, a probabilistic approach was taken. Compared to a multi-CPU implementation, significant speedups were observed in this approach. Later, Martínez-Frutos et al. (2017) implemented topology optimization on a GPU using multilevel granularity. In this study, different granularities of kernel configuration were implemented for different parts of the topology optimization pipeline to achieve the best performance. Matrix-free PCG was implemented at a DOF-by-DOF level, whereas, for mesh filtering, density update, and the calculation of sensitivities, an element-by-element approach was chosen. Two different preconditioners were tested for the PCG. Later, Ram and Sharma (2017) addressed the challenges of generating geometrically feasible structures and computational complexity by implementing a triangular representation of 2D continuum structures using evolutionary algorithm on GPU. A speedup of 5× was achieved on GPU compared to the CPU implementation. Ratnakar et al. (2021b) implemented GPU-based topology optimization using a customized data structure for nodal connectivity. A node-by-node implementation was adopted for matrix-free CG to achieve a speedup of 3× over CPU implementation. Later, Ratnakar et al. (2020) implemented SIMP-based structural topology optimization using 3D unstructured meshes on GPU.

2.5.2 GPU-based BESO

Since its inception, evolutionary structural optimization (ESO) (Xie and Steven, 1993) has been extensively studied in the context of structural topology optimization owing to its intuitive nature and ease of implementation. The fundamental idea behind ESO is to start from a completely solid structure and gradually remove the inefficient portions until the specified weight constraints for the structure are met. Although the original ESO could produce topologies, the procedure ignored several key aspects of topology optimization, such as mesh-independency, existence of solutions, checkerboard patterns, and convergence to local optima. Over the years, several modifications have been made to its original methodology to address these key aspects, which eventually led to the development of the modern bi-directional ESO (BESO) method (Huang and Xie, 2009). The term "bi-directional" in BESO means that the structural material can be added and removed at the same time in a particular iteration, unlike the original ESO where only unidirectional removal was possible. The earlier version of ESO/BESO in which completely void elements were used is called hard-kill BESO. Later, Huang and Xie (2009) proposed the idea of adding a soft material instead of void elements utilizing a material-interpolation scheme with a penalization parameter. This method was referred to as soft-kill BESO.

A large number of studies have been conducted to accelerate topology optimization methods such as SIMP (Schmidt and Schulz, 2011, Martínez-Frutos and Herrero-Pérez, 2016, Martínez-Frutos et al., 2017, Ratnakar et al., 2021b, 2020) and level-set (Challis et al., 2014, Li et al., 2021, Liu et al., 2019)

methods on GPU. However, to the best of the authors' knowledge, there has been only two studies in the literature for GPU-acceleration of ESO/BESO-type methods (Martínez-Frutos and Herrero-Pérez, 2017, Munk et al., 2019). Martínez-Frutos and Herrero-Pérez (2017) implemented a parallel ESO method using isosurfaces with GPU-accelerated fixed-grid FEA for continuum structures. A hard-kill approach was implemented by using marching cubes algorithm to extract isosurfaces from scalar volumetric data sets. Speedups of $7 \times -19\times$ were achieved for GPU implementations over CPU implementations with good scalability. Further, the authors demonstrated the computational benefits of removing void elements from the solution of the system of equations in terms of wall-clock time and device memory requirements. The other GPU-based implementation of BESO is by Munk et al. (2019), who implemented a GPU-accelerated lattice Boltzmann method for multi-physics topology optimization. The GPU-based results were compared with CPU-based results of the implementation (Munk et al., 2017) to reveal $18 \times -67\times$ speedups. Furthermore, the obtained results indicated some discrepancies between the CPU and GPU-based implementations that are imperative to the acceleration of multi-physics topology optimization.

2.6 Literature: Closure

From the literature review, it can be seen that although almost every part of FEA has been ported on GPUs, a majority of the research of effort in accelerating FEA is concentrated on the matrix generation and solver stage. Several works have worked on accelerating assembly and numerical integration on GPU. It is clear from the literature as pointed by several researchers, that in general, assembly-free methods are more effective than addto methods for use on GPUs. This is due to the requirement of complex data movement during traditional addto assembly. In the field of numerical integration on GPUs, although significant amount of research has been done exploiting the embarassingly parallel nature of the computation. In the question of choosing the precision, it can be seen that most of the works in single precision date back to a time when double precision computations were not supported by GPUs. However, even after incorporating the double precision support, a number of researchers has opted for either single or mixed precision approach. Although the high end HPC oriented GPUs such as Tesla from NVIDIA and FirePro from AMD provide significant double precision performance, a majority of GPUs from varying vendors suffer from mediocre to poor double precision performance. Furthermore, it is observed that most of the research work in the field of matrix solver are oriented toward iterative methods such as CG with only a handful of works targeting sparse direct solvers.

Among the applications of large-scale computation-intensive FEA, one of the most important applications is topology optimization using density-based methods. Two of the most popular methods in the literature for topology optimization are SIMP and BESO. There has been several algorithm-level and HPC-based improvements to these methods over the last two decades. While there are different types of algorithm-level modifications in the literature, most of the HPC-based modifications come in the

form of efficient GPU acceleration. A majority of GPU-accelerated topology optimization implementations makes use of matrix-free FEA instead of traditional addto assembly. It is observed that most of the research effort on accelerating topology optimization on GPUs have focused on the SIMP method, whereas, there is only two research work that have targeted GPU acceleration of BESO. Furthermore, to the best of the authors knowledge, there has been no work implementing a complete acceleration of the BESO method pipeline, making it an open area of research.

Lastly, from the survey, it is clear that the choice of GPU vendor in both academia and industries is NVIDIA. The reason is primarily CUDA, which provides the developer with a very efficient API to control the hardware while also having a learning curve far less steep than that of other programming paradigms.

Chapter 3

Three-stage FEA Matrix Generation

The first objective of the thesis is to accelerate the matrix generation stage of FEA using GPU computing. This objective includes generation of elemental stiffness matrices for three dimensional structured and unstructured meshes and for lower and higher order elements on GPU, along with their assembly into a global stiffness matrix. The primary challenge with this objective is the effective distribution of workload along with an efficient storage scheme for the global stiffness matrix on GPU. For assembly, a three-stage GPU-based FEA matrix generation strategy is presented with the key idea of decoupling the computation of global stiffness matrix indices and values by the use of a novel data structure referred to as the neighbor matrix.

In the first stage, a new data structure called the neighbor matrix is introduced. This neighbor matrix is computed from the connectivity information of the unstructured mesh in the form of two one-dimensional arrays. In the second stage, the indices of the NZ entries in the sparse storage format are computed and stored in parallel. This computation of indices can be distinguished from the value computation of NZ entries due to the fact that the former depends only on the connectivity information of the mesh. Finally, in the third stage, the NZ values are computed and stored based on the indices computed in the second stage. Thus, the index and value computation for NZ entries are decoupled for unstructured meshes with the help of the mesh preprocessor. This, although straightforward in the case of structured meshes, becomes challenging for unstructured meshes where the data is scattered irregularly over the mesh.

Two sparse storage formats based on the proposed strategy are also developed by modifying the existing sparse storage formats with the intention of removing the degrees of freedom-based redundancies in the global stiffness matrix. The inherent problem of race condition is resolved through the implementation of coloring and atomic operations. Furthermore, in this chapter, the effect of computing the elemental stiffness matrix for each element inside and out of the assembly kernel on GPU in connection

to the kernel division strategy is investigated.

3.1 Mesh Preprocessing: Neighbor Matrix Computing

Due to restrictions imposed by the memory hierarchy on GPU hardware, the entire data cannot be made available to all the compute threads at the same time. A common solution to this problem is to process the data in a manner such that it can be segregated easily for each thread and only the part of the data required by a thread can be made available exclusively. Furthermore, since the FEA matrices are large and sparse in nature, specialized sparse storage formats such as COO, CSR, and ELL are required to store the values. This creates more challenges for writing and accessing the global stiffness matrix due to the compact storage of the NZ entries into the sparse format. For example, in Figure 3.1, a one-dimensional domain is taken with four elements marked as '1', '2', '3', and '4'. The four elemental matrices considering two DOFs per node are shown in the figure. If assembly is to be performed in the standard dense format, as shown by K^g in the figure, it can be done directly from the connectivity mapping of the mesh. However, if the global stiffness matrix is to be assembled directly into the ELL format, as shown by K^g_{ELL} in the same figure, the target locations of NZ entries (marked as red) get distorted, and consequently, the target indices for the elemental matrices cannot be found directly from the mesh connectivity information alone. This can be handled on GPU by precomputing the column indices and performing a bisection search on the column array when a value is to be added to a particular row. However, this search operation is undesired on GPU and usually results in performance loss. In the proposed implementation, the search is instead performed at the node level with the pre-computed neighbor matrix to reduce the effect of the search.

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

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

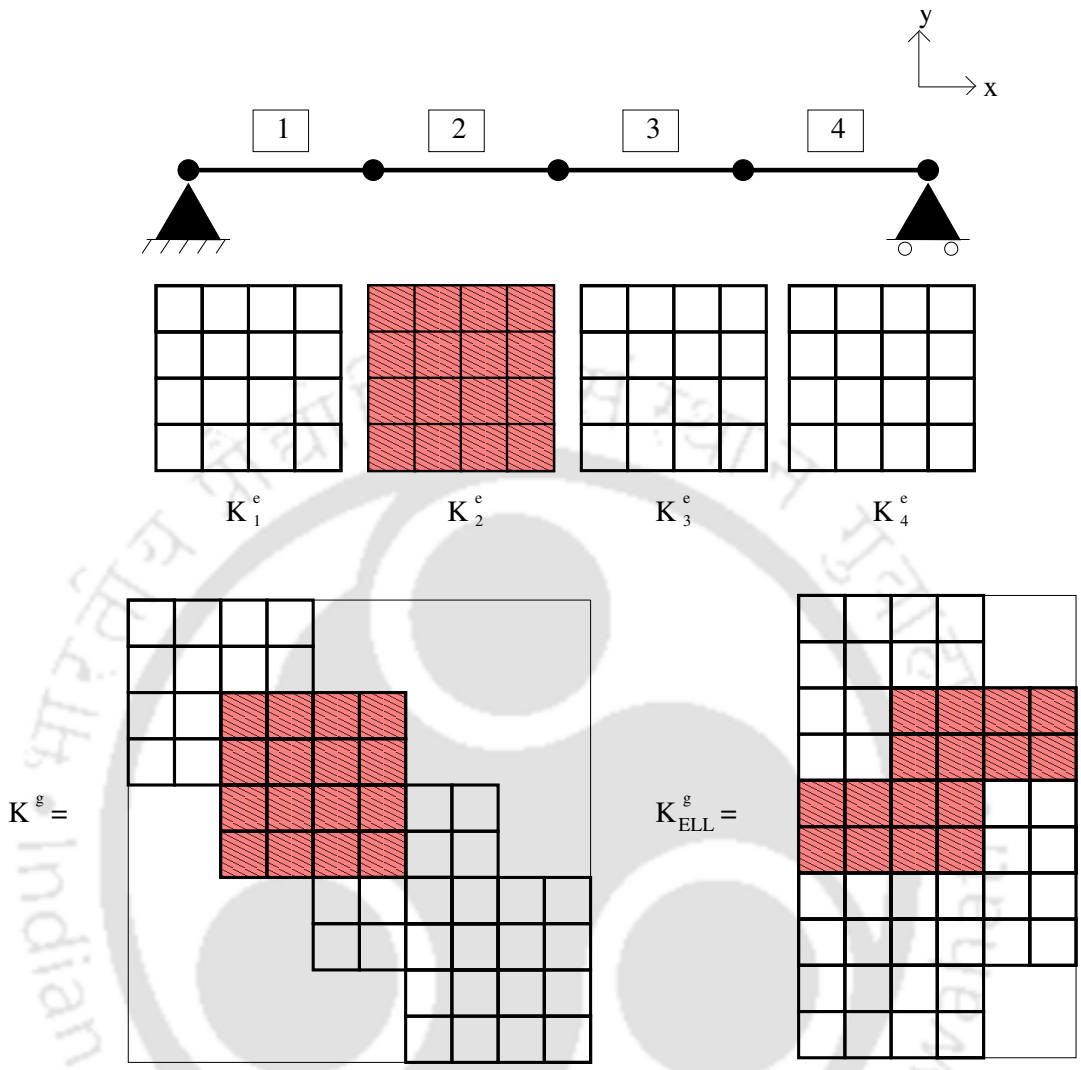


Figure 3.1: Assembly into the standard format versus the sparse storage formats

3.1.1 Neighbor Matrix Computation

The implementation details of the neighbor matrix computation are outlined in this section. It is computed using a combination of custom kernels and functions available in the *thrust* library of the NVIDIA CUDA Toolkit. Thrust is a software library for CUDA based on the C++ Standard Template Library (STL) that allows the user to implement high-performance parallel applications through a high-level interface. The neighbor matrix is designed to store a sorted list of nodes connected to a particular node of the unstructured finite element mesh. This matrix will later be required to allocate memory and to store the NZ entries of the global stiffness matrix in a sparse storage format. The neighbor matrix is demonstrated in Figure 3.2, where a mesh of triangular elements has been taken. The example mesh has seven nodes marked as '1', '2', ..., '7' and six elements marked as '*a*', '*b*', ..., '*f*'. The neighbor matrix consists of two one-dimensional arrays called *nNum[]* and *nInd[]*, which are also illustrated in the figure.

It is to be noted that *nNum[]* has been shown as two-dimensional in the figure for clarity purposes. However, it is flattened and stored as a one-dimensional array on GPU. In the two-dimensional format, it contains the same number of rows as the number of nodes in the mesh. Each row contains a sorted list of nodes connected to the corresponding node including itself. For example, in the figure, node '5' is connected to nodes '4', '6', and '7'. Hence, the 5^{th} row of *nNum[]* array lists '4', '5', '6', and '7'. To identify the entries in the flattened array for each node, the *nInd[]* array is introduced in the figure. The $(i-1)^{th}$ and i^{th} entries of the *nInd[]* array give the start and end indices of *nNum[]* array for the i^{th} node. For example, the start and end indices for node 5 in the figure is given by the 4th and 5th entry of *nInd[]* (16 and 20), respectively. Furthermore, the *nInd[]* array will be used to compute the memory allocation size for sparse storage formats on GPU. It is to be noted here that since the neighbor matrix computation is dependent only on the connectivity information of the mesh, it can be performed for any type and order of elements with small modifications in the implementation.

Algorithm 1 lists the functions required for the neighbor matrix computation. These functions are described in details with the help of Figure 3.3. In the figure, the neighbor matrix is computed step-by-step for the triangular mesh shown in Figure 3.2. The steps in the neighbor matrix computation are demonstrated using arrays [I], [II], ..., [XI] in the same figure. The flattened arrays shown in the figure point to their respective indices ([V′] and [IX′]).

Algorithm 1: Neighbor matrix computing

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

Output: $nNum[]$: Array [IX′]; $nInd[]$: Array [XI];

1 $countElemNum <<< ((nElem-1)/256)+1, 256 >>> (C[])$;

2 $thrust :: inclusive\_scan()$;

3 $fillElemNum <<< ((nElem-1)/256)+1, 256 >>> (C[])$;

4 $thrust :: inclusive\_scan()$;

5 $fillNodeNum <<< ((nNode-1)/256)+1, 256 >>> (C[])$;

6 $thrustDynamicSort <<< ((nNode-1)/128)+1, 128 >>> (C[])$;

7 $thrust :: inclusive\_scan()$;

3.1.1.1 countElemNum Kernel

The purpose of this kernel is to compute the number of elements connected to each node of the mesh, as shown in Figure 3.3. Array [I] shows the node numbers of the given triangular mesh in Figure 3.2. It can be seen that node 1 is only connected to element 'a'. Therefore, one is stored at the first place of array [II]. Similarly, node 2 is connected to three elements 'a', 'b', and 'c', therefore, three is stored at the second place of array [II]. By following the same procedure, the other places of array [II] are filled

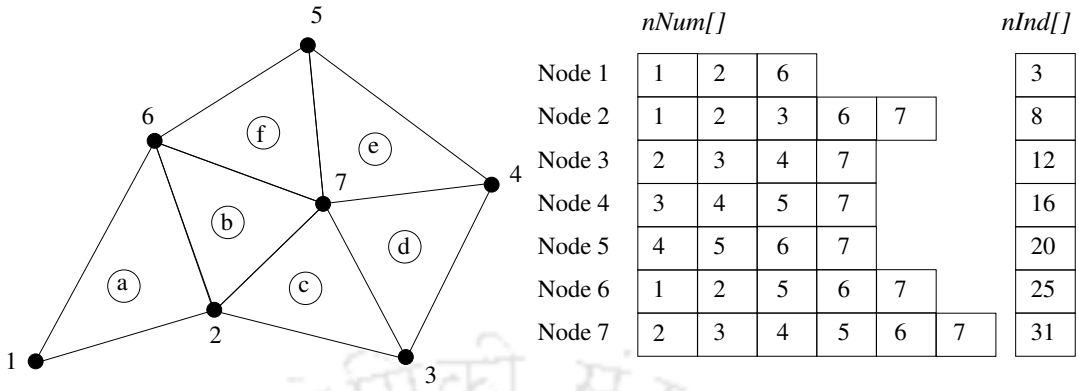


Figure 3.2: Neighbor matrix for a triangular mesh

with the numbers of elements connected to the nodes. The above procedure for *CountElemNum kernel* is presented in Algo. 2, which is launched with threads equal to the total number of elements in the mesh. It can be understood from the algorithm that *atomicAdd* function is used and the values of array [II] are stored in *neighborCount[]* array.

Algorithm 2: countElemNum Kernel

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

```
countElemNum <<< ((nElem − 1)/256) + 1, 256 >>> ();
__global__ void countElemNum(int *C, int *neighborCount, int nElem);
int tx = blockIdx.x × blockDim.x + threadIdx.x;
if tx < nElem then
    for i ← 1 : n_e do
        atomicAdd(&neighborCount[C[n_e × tx + i]], 1);
    end
end
```

3.1.1.2 Inclusive Scan

An inclusive scan is performed to find the size of memory allocation to GPU using *thrust* library. A cumulative sum of the numbers shown in array [II] is stored in array [III]. The number on the last position of array [III] (15 in the example figure) shows the memory size required on GPU for the given mesh. For clarity, array [IV] is presented in Figure 4, which shows the number of empty cells equivalent to the number shown in array [II] for every node. Array [III] is also utilized to identify entries for each node in array [V].

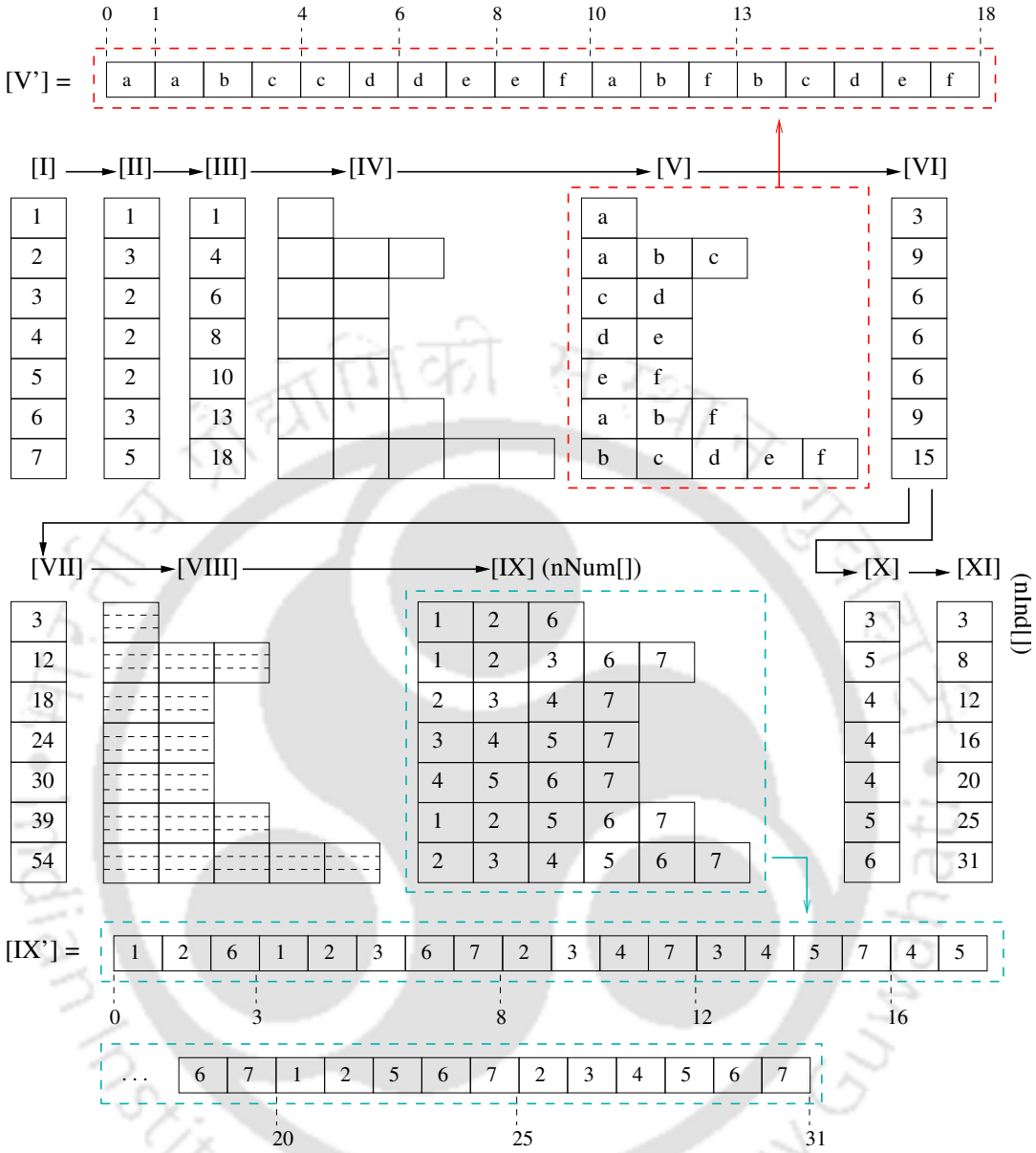


Figure 3.3: Pre-processing the mesh data for assembly

3.1.1.3 fillElemNum Kernel

The memory allocated on GPU after the inclusive scan is now used for storing the element numbers connected to each node. Array [V] in figure 3.3 represents those connected elements. It is noted that array [V] is stored as a one-dimensional array [V′] on GPU, which can be seen in the inset of the same figure. It can be seen that element '*a*' is stored at the first place of array [V′], followed by the elements of node 2, '*a*', '*b*', and '*c*'. Similarly, the elements connected to other nodes are stored. The indices of array [V′] are the same as shown in array [III] in order to locate the elements connected with the nodes. The procedure of this kernel is shown in Algo. 3, which is launched with threads equal to the

total number of elements in the mesh. It is to be noted that the one-dimensional array [V′] is stored in *elemNum[]* array after executing this kernel.

Algorithm 3: fillElemNum Kernel

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

```
1  fillElemNum <<< ((nElem − 1)/256) + 1, 256 >>> ();
2  __global__ void fillElemNum(int *elemNum, int *C, int *neighborCount, int nElem);
3  int tx = blockIdx.x × blockDim.x + threadIdx.x;
4  if tx < nElem then
5  |  for i ← 1 : n_e do
6  |  |  int temp;
7  |  |  temp = atomicSub(&neighborCount[C[n_e × tx + i]], 1);
8  |  |  elemNum[temp - 1] = tx;
9  |  end
10 end
```

3.1.1.4 Inclusive Scan

After storing the number of elements in array [II], the total numbers of nodes connected to those elements are stored in array [VI]. In the given mesh, every element is connected to $n_e = 3$ nodes. Therefore, the number '3' is stored at the first position of array [VI]. At the second position of array [VI], three elements 'a', 'b', and 'c' are stored; therefore, $3 \times n_e =$ '9' is stored. Similarly, the numbers are stored for other positions of array [VI]. It must be noted that the number stored in array [VI] can be directly found from array [III] after multiplying every number with n_e. The numbers of array [VI] are stored in *nodeCount[]* array. Another inclusive scan is performed on array [VI], and the cumulative sum of the numbers in array [VI] is stored in array [VII]. The number on the last position of array [VII] indicates the size of memory allocation on GPU. Similar to the approach described in Section 3.1.1.2, array [VIII] is shown in Figure 4 for clarity. In this figure, the number of empty cells is equal to the numbers shown in array [VII]. After the inclusive scan, the required memory is allocated to *nodeNum[]* array on GPU.

3.1.1.5 fillNodeNum Kernel

The memory allocated in the inclusive scan in Section 3.1.1.4 is filled with the node numbers in array [VIII]. The first position of array [VIII] stores the node numbers '1', '2', and '6' because these nodes are connected to the element 'a', which was stored at the first position of array [V]. At the second position of array [VIII], the elements stored at the second position of array [V] 'a', 'b', and 'c' are used. The nodes connected to element 'a' are '1', '2', '3', to element 'b' are '2', '6', '7', and to element 'c' are '2', '3', and '7'. These node numbers are stored at the second place of array [VIII]. Similarly, the remaining positions of array [VIII] (*nodeNum[]* array in Algo. 4) store the nodes. This procedure is shown in Algo.

4, which is launched with threads equal to the number of nodes in the mesh.

3.1.1.6 thrustDynamicSort Kernel

This kernel is designed to sort the nodes at every row of array [VIII] and to remove duplicate nodes. At the first position of array [VIII], the nodes '1', '2', and '6' are already sorted with no duplicates. These nodes are copied to array [XI]. At the second place of array [VIII], the nodes are unsorted and the duplicate nodes are also present. At this stage, the kernel sorts them and the duplicates are removed. Thereafter, the nodes '1', '2', '3', '6', and '7' are stored at the second position of array [IX]. By following the same procedure, the other positions of array [IX] are filled. It must be noted that the two-dimensional array [IX] has been presented as a graphical representation for clarity. However, the data is stored in a one-dimensional array [IX′] in the same figure with the required indices. This kernel uses the *Dynamic Parallelism* feature of the Kepler microarchitecture. With this feature, grids can be launched from inside a GPU kernel. Two *thrust* calls are made inside the kernel for each node, as shown in line 7 and line 8 of Algo. 5, for first sorting the nodes and then removing the duplicates. The kernel is launched with threads that are equal to the number of nodes in the mesh, and array [IX] is stored in *nNum[]* array. At the end of this kernel, the *nodeCount[]* array (array [VI]) is updated in line 9 by subtracting the number of repeated entries from each corresponding row. The updated values are shown in array [X] in Figure 3.3. Essentially, this array simply lists the number of neighboring nodes of each node.

3.1.1.7 Inclusive Scan

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

Algorithm 4: fillNodeNum Kernel

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

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

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

Algorithm 5: thrustDynamicSort Kernel

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

Output: $nNum[]$: Array [IX′];

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

After performing all the steps of Algo. 1, the neighbor matrix is generated in the form of two

one-dimensional arrays *nNum[]* and *nInd[]* on GPU. This matrix is used to compute GPU memory allocation size for the global stiffness matrix in the following ways:

- **ELL Sparse Format:** ELL sparse format consists of two matrices for storing the column indices and the values of the sparse matrix. Both of these matrices have rows equal to the number of DOFs of the system. The number of columns is computed by calculating the maximum number of nodes surrounding any particular node in the mesh and multiplying it by the number of DOF per node. For a general mesh, after calling `thrustDynamicSort` kernel of Algo. 5, `thrust::max_element` is called with the final *nodeCount[]* array (array [X] in Figure 3.3) to give the maximum number of neighboring nodes for any node of the mesh. This number multiplied with the DOF per node gives the column number of the ELL format.
- **COO/CSR Sparse Format:** COO format consists of three one-dimensional arrays for storing the row indices, column indices, and the values of the NZ entries. For this format, the last entry of *nInd[]* array is used. This value multiplied by the DOF per node gives the length of the three arrays of the COO format. For storing in the CSR format, the sizes of the value array and the column array remain the same. The size of the row offsets array is calculated as the number of rows in the global stiffness matrix plus one.

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

3.2 Index Computation for the NZ Entries

In this stage, the row and column indices of the sparse storage format for storing the NZ entries in the global stiffness matrix are calculated. The idea behind this stage is that the number of NZ entries in a particular row or column of the global stiffness matrix can be calculated by multiplying the number of neighboring nodes to a node with its DOF. For a node i with θ_i number of neighboring nodes, there can be $(\theta_i + 1)ndof$ total NZ entries in the associated rows and columns of the global stiffness matrix, where $ndof$ is the DOF per node. The corresponding indices can be calculated from the neighboring node numbers. In this kernel, each thread is assigned to one node of the finite element mesh, as shown in Figure 3.4.

A generalized algorithm for the index computation is presented in Algo. 6, in which the required data is preprocessed for determining the row and column indices for every NZ entry. At step 4, neighboring nodes (S^i) are derived from the neighbor matrix that is computed in the mesh preprocessor stage using Algo 1. At step 6 of Algo. 6, one thread is assigned to every node i. Each thread first initializes two null sets (Φ^i, Γ^i) for storing the row and column indices. It can be seen that step 10 stores d_k^i DOF of

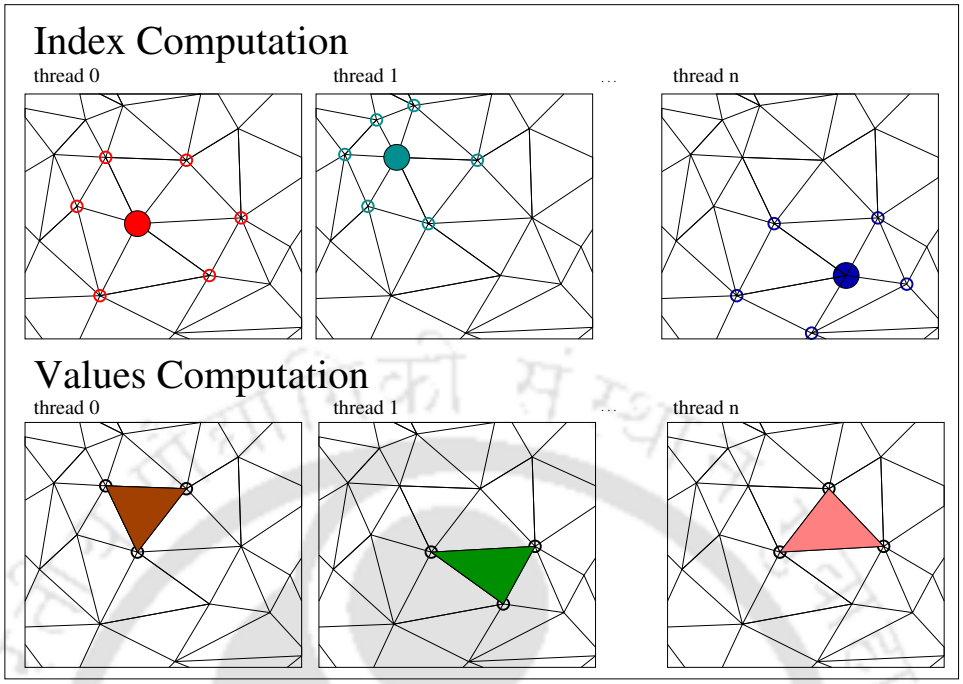


Figure 3.4: Thread allocation scheme for the index computation and values computation kernels.

node i in Φ^i, which is copied $(\theta_i \times ndof)$ times. As can be seen from Figure 3.5, all DOFs of node i are copied one by one into Φ^i. Therefore, Φ^i stores $(\theta_i \times ndof^2)$ indices for node i. Every Φ^i is then copied to Φ set of row indices. Regarding the column indices for every node i, d_k^j DOF of node $j \in S^i$ is copied into Γ^i (refer to step 16). Once all the DOFs of neighboring nodes to node i are copied into Γ^i, the same set is copied $(ndof)$ times to Γ. The set Γ also stores $(\theta_i \times ndof^2)$ indices for node i. Figure 3.5 shows D^i, S^i, Φ^i, Γ^i, Φ, and Γ for node i in the two-dimensional domain having two DOFs per node.

3.3 Values Computation for NZ Entries

In this stage, the values of the global stiffness matrix entries are computed and stored in a sparse storage format. An element-by-element assembly strategy is adopted for the implementation, as shown in Figure 3.4. Since each node is shared by a number of elements in the mesh, a situation may occur where a number of threads try to access the same memory location simultaneously, resulting in race condition or data race. This can cause inaccuracies in the final result. This is handled at the hardware level by using atomic operations.

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

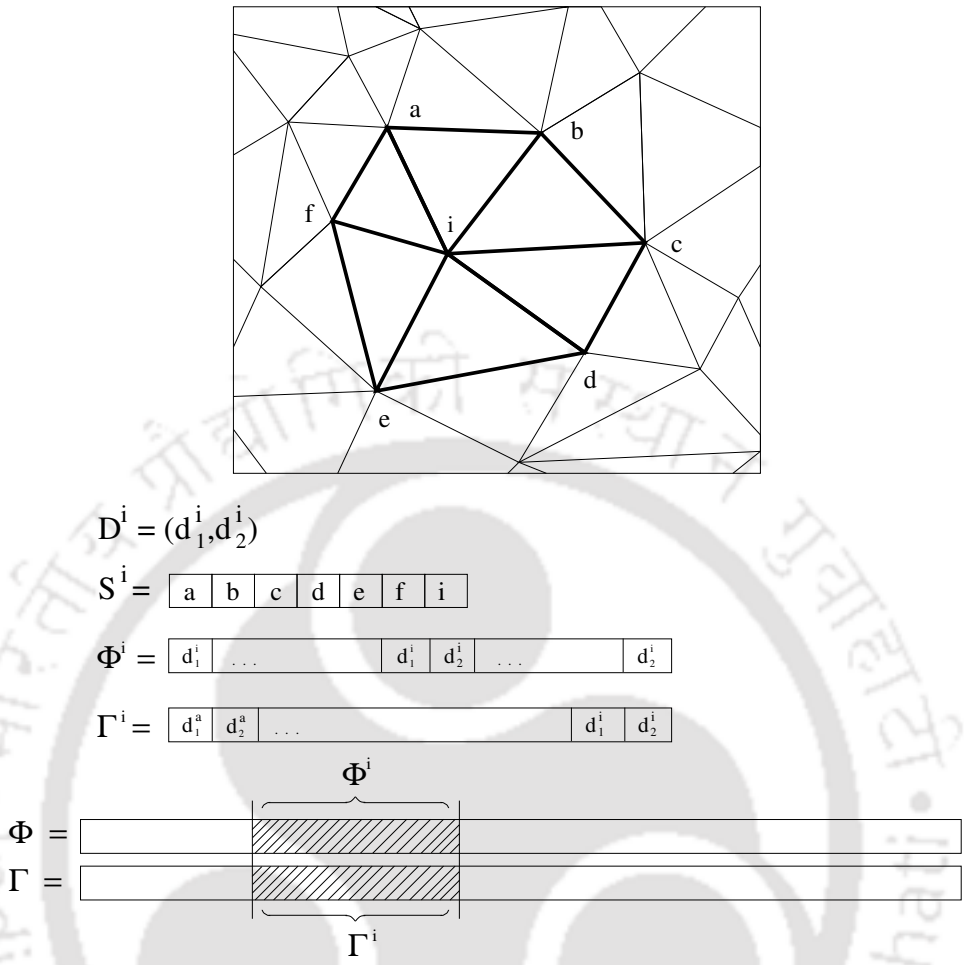


Figure 3.5: Different sets of data for node i are presented for a two-dimensional domain having two DOF per node. D^i represents the DOFs of node i. S^i represents the set of neighboring nodes to node i. Φ^i and Γ^i represent the row and column indices respectively of the NZ entries related to node i.

A generalized algorithm for the value computation of NZ entries is presented in Algo. 7 in which NZ entries are stored based on the row and column indices (Φ, Γ), which are determined using the computed indices in the previous stage. In this kernel, one thread is assigned to one element in step 1. Using the connectivity matrix for element e ($C[e, nnodes]$), the target index (p) for the first DOF (d_1^j) of node $j \in C[e, nnodes]$ is searched in $\Gamma^i \in \Gamma$ for every *node-to-node* connection (refer step 4). We need to search only the first DOF (d_1^j) of node $j \in C[e, nnodes]$, as other DOFs can be determined from the pattern used for storing indices using Algo. 6. Finally, the entries of the elemental stiffness matrix are stored in the sparse storage format, as shown in step 9.

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

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

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

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

3.4 Assembly into COO, CSR and ELL

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

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

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

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

Input : $ndof$: DOF per node; E: total elements; $nnodes$: nodes per element; $C[E, nnodes]$: Connectivity Matrix; Φ, Γ;

Output: K: Global stiffness matrix in the sparse storage format;

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

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

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

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

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

3.5 Race Condition

A race condition or data race in computation occurs when the output of a process becomes dependent on the sequence in which two simultaneous threads access a memory location. The GPU-based implementation of FEA assembly is susceptible to this issue when more than one thread write in the same memory location. In the symbolic kernel for computing the NZ indices, with each node being assigned to a single unique thread, the possibility of sharing of nodes among any thread is alleviated. This nodal independence removes any chance of a race condition. This, however, is not true for the numeric kernel responsible for computing the NZ values, where each element is assigned exactly to one thread. Due to the fact that all the elements in the mesh have shared nodes, the possibility exists for two or more threads writing to the same memory location at the same time. Therefore, we implement atomic operations and element-coloring methods to counter the race condition.

When an atomic operation such as `atomicAdd` or `atomicSub` is invoked, it locks the concerned memory location and waits until the requested operation is completed (Kirk and Wen-mei, 2012). This is usually avoided due to the serialization of atomic threads hampering the overall performance. This thesis keeps the threads waiting for an atomic lock to a bare minimum. For a dense-enough mesh, the amount of serialization caused by an atomic operation becomes smaller and smaller and the performance degradation becomes negligible. In the second implementation, we color the mesh elements with different colors such that an element is only allowed to share nodes with elements of different colors. The element-coloring method is illustrated in Figure 3.6 for a structured mesh. After the coloring is done, separate kernels for each of the colors are invoked in a serial manner.

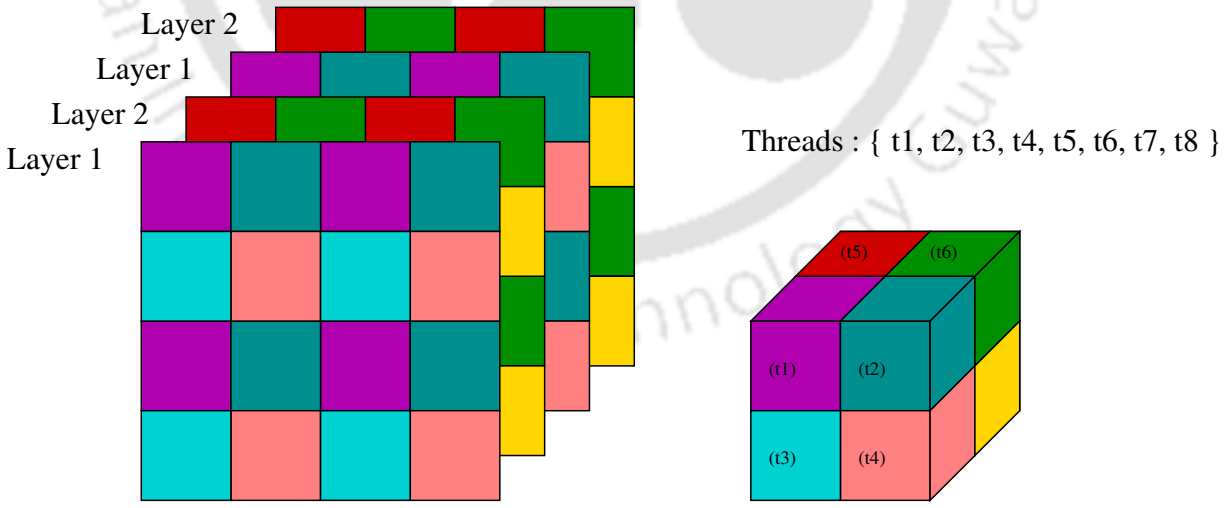


Figure 3.6: Coloring Scheme for race condition

3.6 Same and Separate Kernel Approaches

The same kernel and separate kernel approaches are used for assembling and storing the elemental stiffness matrices in the value array of the sparse storage format. As shown in Figure 3.7, the neighbor matrix is computed first followed by index computation in the same kernel approach. In this approach, the generation of elemental stiffness matrix and value computation of the global stiffness matrix are performed in the same kernel. This obviates the need to store the elemental matrices on GPU. In the separate kernel approach, as shown in Figure 3.7, elemental matrix generation is performed separately at the beginning, and all the elemental stiffness matrices are stored in the global memory of GPU. After computing the neighbor matrix, the indices of the global stiffness matrix are computed. Subsequently, the value computation kernel is launched that takes the previously computed elemental stiffness matrices from global memory and assembles them into the value array.

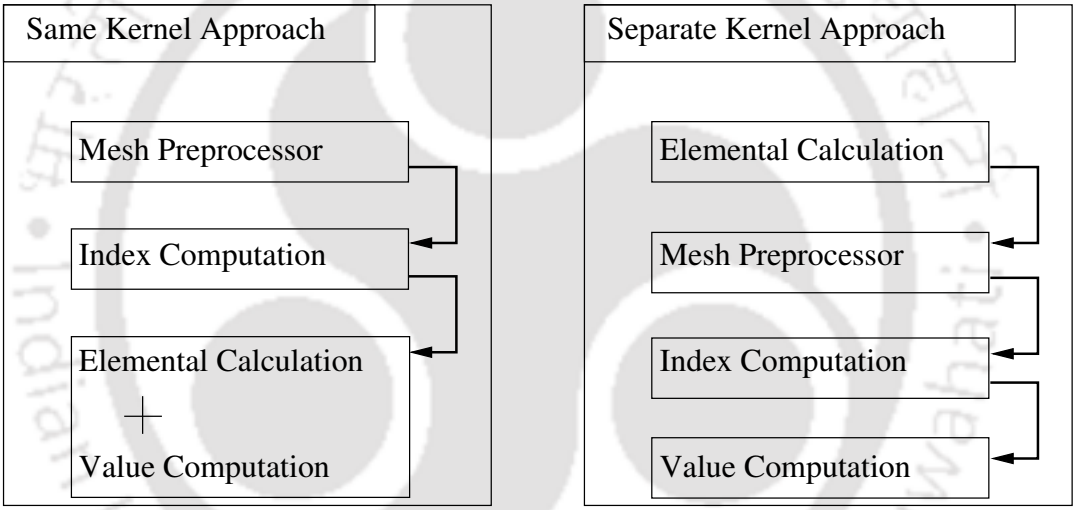


Figure 3.7: Flow charts for the same kernel and separate kernel approaches.

3.7 Results and Discussion

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

Table 3.1: The mesh preprocessing time for different element types for hollow cylinder example

Element type	Elements	Nodes	Time taken (s)
TET4	7,32,007	1,30,884	0.201
TET10	7,32,129	10,12,919	0.813
HEX20	1,75,550	7,42,677	0.110

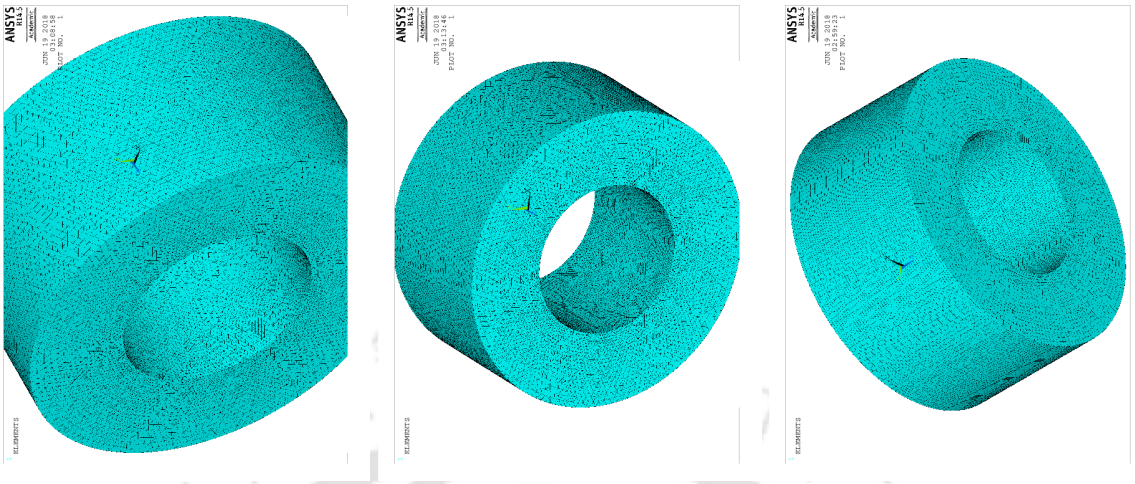
Table 3.2: Time taken by different functions in the pre-computation stage in seconds (hollow cylinder)

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

3.7.1 Example 1: Hollow Cylinder

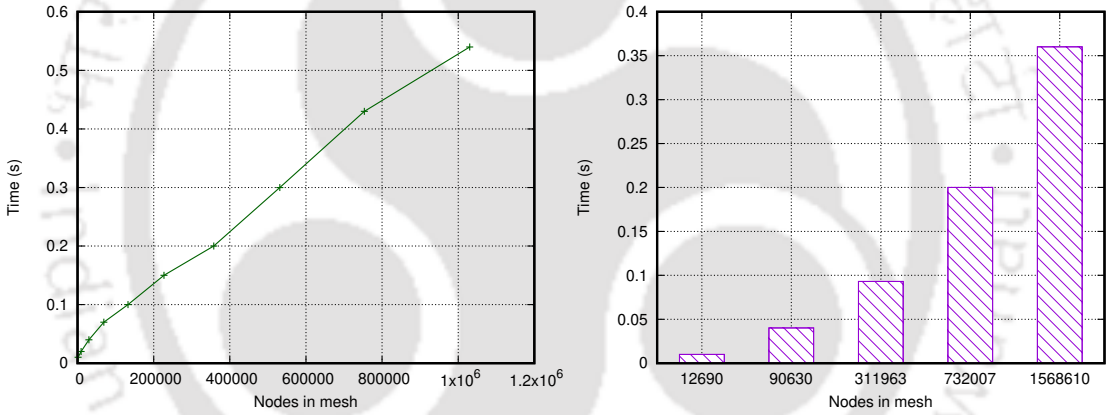
A hollow cylinder is considered, which is meshed with three types of meshes using ANSYS® software. Figure 3.8 shows the hollow cylinder with 4-noded tetrahedral (TET4), 10-noded tetrahedral (TET10) and 20-noded hexahedral (HEX20) meshes. The mesh preprocessing (neighbor matrix generation) time for different types of meshes is presented in table 3.1. This time is important because it is part of the total assembly time. Table 3.4 lists the times required by different steps of algorithm 1 for three different mesh sizes. It can be seen that among all the steps, *thrustDynamicSort* kernel takes the most amount of time for all the mesh sizes. Figure 3.9(a) shows the mesh preprocessing time for HEX20 for different numbers of nodes with the hollow cylinder example. It can be observed that the preprocessing mesh time is quite less for even a large size of mesh. With TET4 mesh type, the same observation can be seen from figure 3.9(b). The execution times of the mesh-preprocessor are found to have an almost linear relationship with the increasing number of nodes in the mesh.

The assembly time using HEX20 and TET4 mesh is shown in figure 3.10. This assembly time has been further divided into mesh preprocessing time, index computation time and values computation time. Figure 3.10(a) shows the variation among those computation times for different number of nodes using HEX20 element type using ELL sparse storage format. It is noted that the same number of nodes has been taken in all three-directions for meshing the hollow cylinder. It can be seen from the figure that the mesh preprocessing time consumes the least time as compared to other kernels for assembly. Most of the assembly time is consumed in the value computation kernel. Similar analysis can be seen in figure 3.10(b) by meshing the hollow cylinder with TET4 mesh type. The figure shows computation time of mesh preprocessing, index computation and values computation using ELL sparse storage format. Most of the time is consumed during the third stage in value computation followed by mesh preprocessing



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

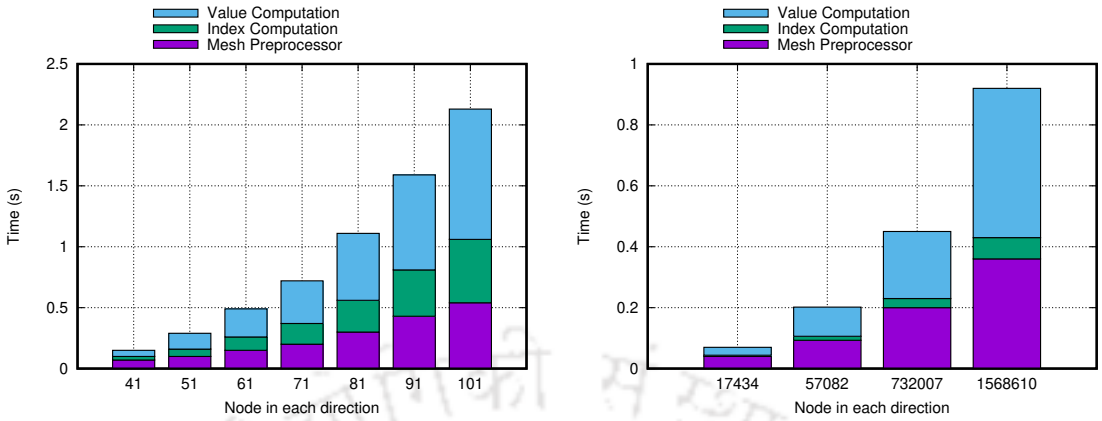
Figure 3.8: Hollow cylinder mesh using (a) TET4, (b) TET10 and (c) HEX20 element type



(a) HEX20 (b) TET4

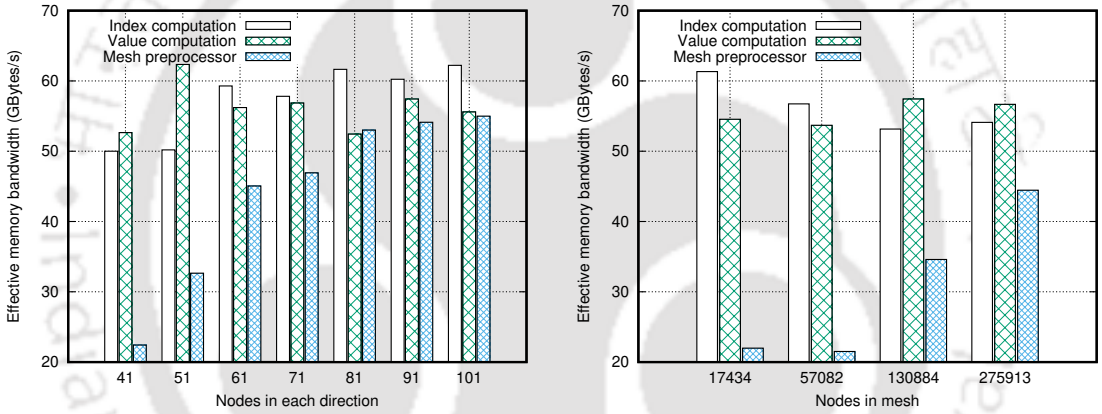
Figure 3.9: Wall clock time comparison for mesh preprocessing using (a) HEX20 and (b) TET4 with increasing number of nodes in the mesh for hollow cylinder example

and index computation. The comparison of effective memory bandwidth using HEX20 and TET4 mesh is shown in figure 3.11. The achieved bandwidth for index computation, value computation and mesh preprocessor are shown. In figure 3.11(a), the variation of achieved memory bandwidth is shown for HEX20 element type using ELL sparse storage format for all those stages. Similar to figure 3.10(a), cylindrical mesh with same number of nodes in all direction is taken. It is observed that the achieved bandwidth for the mesh preprocessor is the least among the three stages. A similar observation can be done from the figure 3.11(b), where the achieved bandwidth of a hollow cylinder meshed with TET4 elements is compared for increasing node numbers. Little variation is seen in the bandwidth values for the index computation and values computation stages with increasing node numbers. The mesh



(a) HEX20 (b) TET4

Figure 3.10: Wall clock time comparison for (a) HEX20 and (b) TET4 element type for increasing number of nodes in the mesh for hollow cylinder example



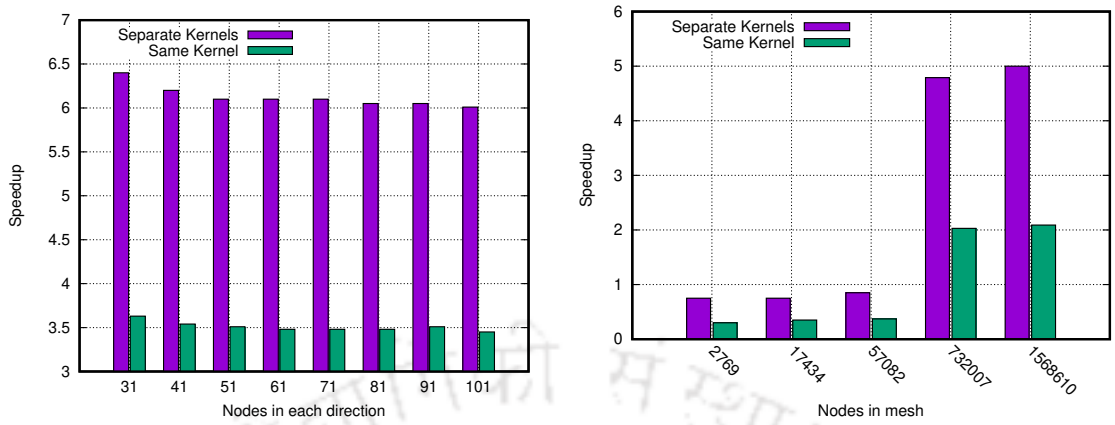
(a) HEX20 (b) TET4

Figure 3.11: Effective bandwidth comparison for (a) HEX20 and (b) TET4 element type for increasing number of nodes in the mesh for hollow cylinder example

preprocessor is observed to achieve the least memory bandwidth among the three stages.

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

The speedup for the kernel division strategy using ELL storage format over SharedNZ algorithm is shown in figure 3.12 for HEX20 and TET4 mesh types respectively. The same and separate kernel approaches for numerical integration and assembly are shown in these figures. It can be seen that both



(a) HEX20 (b) TET4

Figure 3.12: Speedup for (a) HEX20 and (b) TET4 type elements using same and separate kernel approach with respect to SharedNZ (Cecka et al., 2011) implementation for hollow cylinder example.

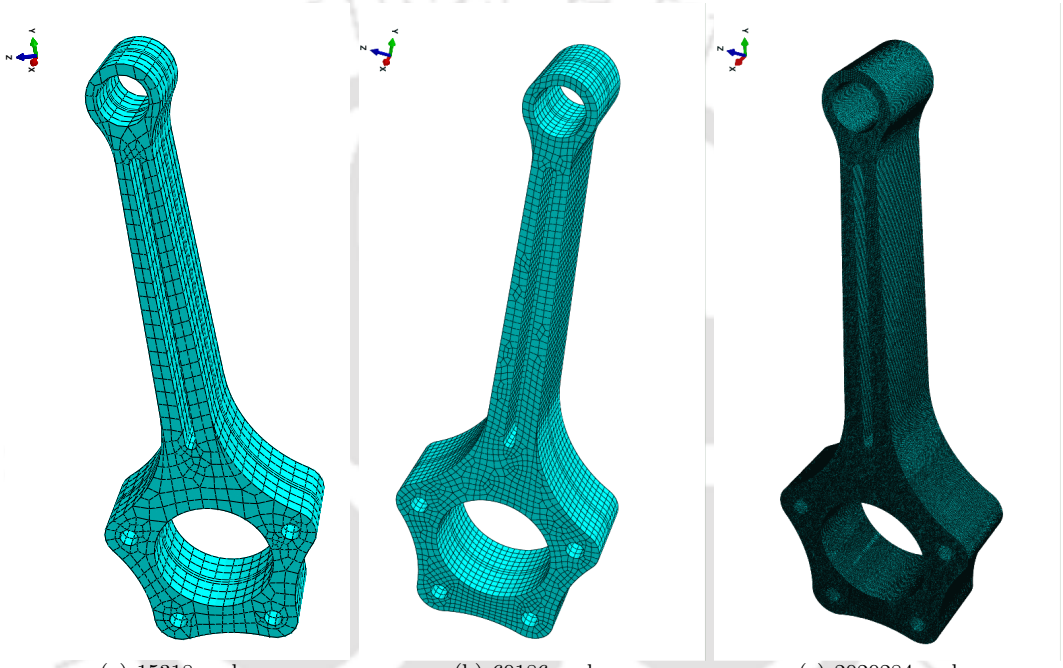
Table 3.3: Element type, number of nodes, number of elements, runtime and memory used for storing the global stiffness matrix on GPU global memory are presented for example 1. S1, S2 and S3 under the runtime column denote mesh preprocessor, index computation and value computation stages respectively.

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

approaches show significant speedup over the SharedNZ implementation in which the separate kernel approach shows speedup close to six for different node numbers for HEX20 mesh type. However, a decent speedup of four can be seen for a large number of nodes with TET4 mesh type. In terms of speedup, HEX20 mesh seems to outperform TET4 mesh for both same kernel and separate kernel approaches for the present example. In table 3.3, details of the element type, number of nodes and elements, runtime of different stages and global memory usage are listed for the current example.

Table 3.4: Time taken by different functions in the pre-computation stage in seconds (connecting rod)

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



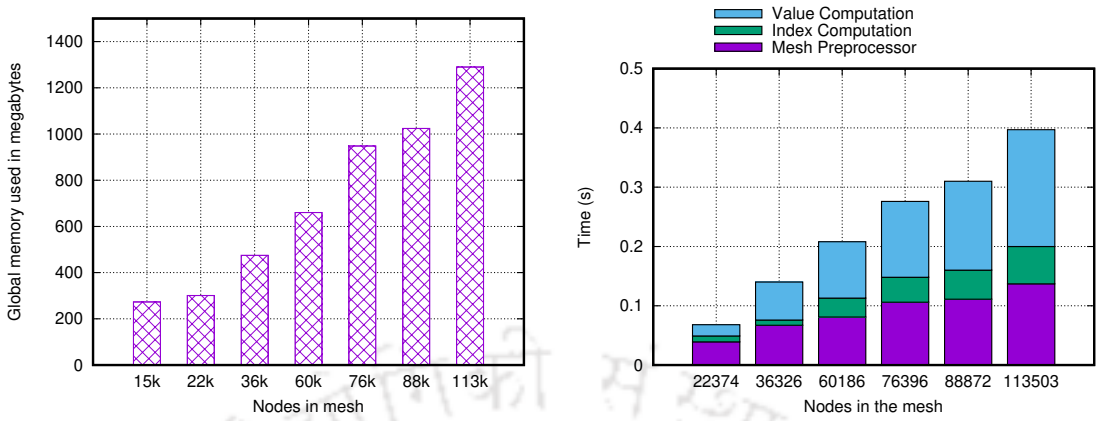
(a) 15318 nodes (b) 60186 nodes (c) 2020284 nodes

Figure 3.13: Connecting rod mesh using (a) 15318 nodes, (b) 60186 nodes and (c) 2020284 nodes

3.7.2 Example 2: Connecting Rod

A connecting rod example is considered as the second example for which unstructured mesh is required. Figure 3.13 shows three different meshes of a connecting rod with different number of nodes. The execution times for different functions in the mesh-preprocessor stage are listed in table 3.4. Similar to the previous example, the *thrustDynamicSort* kernel is found to have the largest execution time, followed by *fillNodeNum* kernel in all three meshes.

Since the unstructured mesh is used for this example, the amount of global memory used for assembly with increasing number of nodes is shown in figure 3.14(a). The assembly time is shown figure 3.14(b) with increasing mesh sizes. Similar to the previous example, the assembly time is further divided into times required by mesh-preprocessor, index computation and values computation. It can be observed



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

Figure 3.14: (a) Global memory usage and (b) Wall clock time comparison for assembly using HEX20 mesh of connecting rod with increasing number of nodes in the mesh

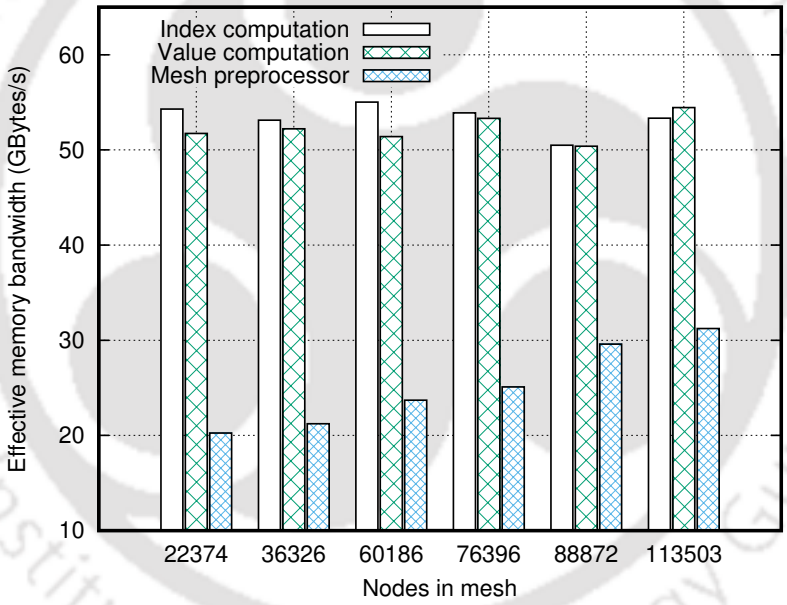
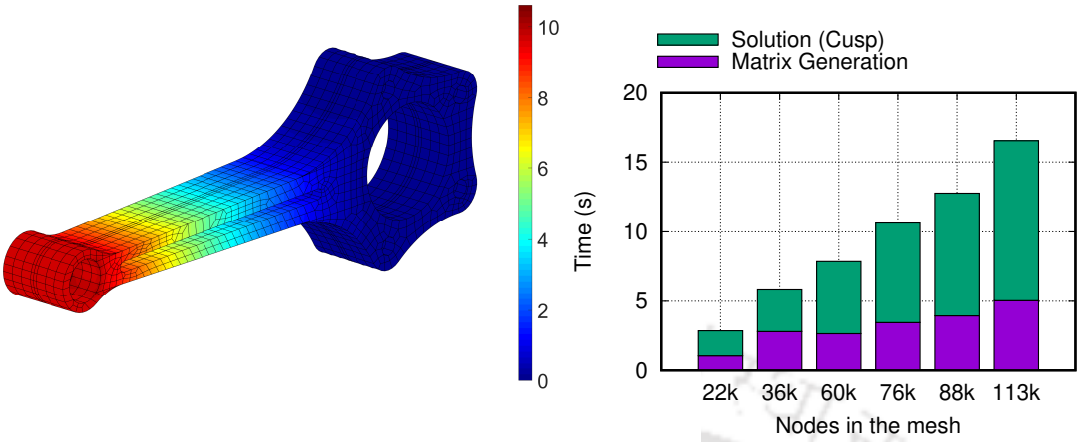


Figure 3.15: Effective bandwidth comparison for assembly using HEX20 mesh of connecting rod with increasing number of nodes in the mesh.

again that the index computation stage consumes the least amount of time, followed by the mesh preprocessor for all the mesh sizes. The achieved memory bandwidth for the different mesh sizes of the connecting rod is shown in figure 3.15. The values for all three individual stages are shown in the plot. The achieved bandwidth values for the index computation and value computation stages are observed to have little variation, whereas it is seen to increase with the mesh sizes for the mesh preprocessor stage.

Cusp library is used for solution of the resulting global stiffness matrix after applying proper bound-



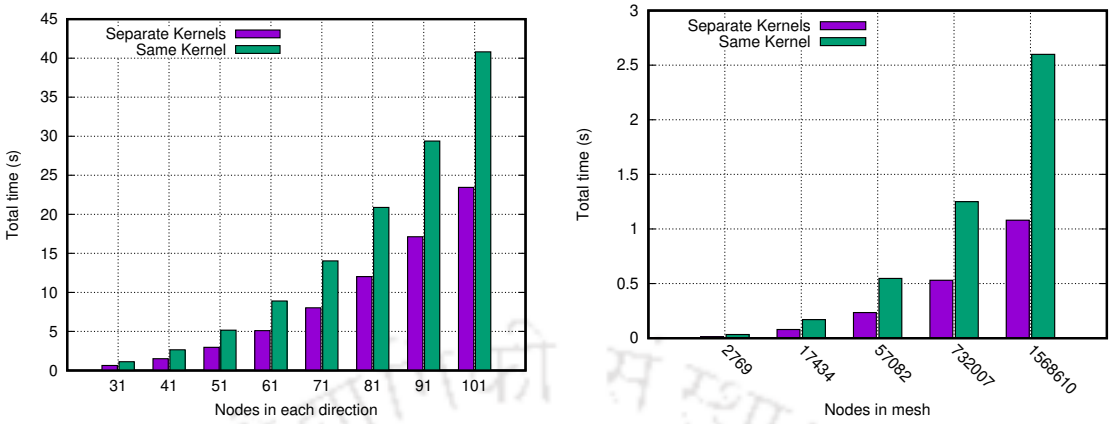
(a) Displacement field

(b) Total execution time

Figure 3.16: (a)Plot of the displacement field and (b) Total execution time of the application, divided into matrix generation and solution using Cusp library for the connecting rod domain

ary conditions. Cusp is a GPU-based library for sparse linear algebra. Conjugate gradient method is used for solution of the resulting matrix on GPU. The resulting displacement field of the connecting rod domain is plotted in figure 3.16(a). Figure 3.16(b) shows the total time taken by the FE analysis with increasing mesh sizes. This time is further divided into time required by the matrix generation and solution stage. It is important to mention here that the time consumed for data transfer from the host to the device is very small compared to the total execution time (less than 0.1%) and is thereby omitted from figure 3.16(b). The reason behind this is that the entire computation, from the elemental stiffness calculation to the solution of the linear system of equations, is executed on GPU with the help of custom kernels and *thrust* operations. It is observed from figure 3.16(b) that the matrix generation step consumes approximately 30% - 45% of the total FE analysis time on GPU.

From algorithm 7 of value computation, it can be seen that the elemental stiffness matrices are assembled into the global stiffness matrix. To compute the element stiffness matrices, the numerical integration is required. Two approaches have been considered to generate the elemental stiffness matrices as discussed in Section 3.6 using the same and separate kernel approaches. Figure 3.17 shows the computation time for both approaches using HEX20 and TET4 mesh types respectively considering ELL format. For both mesh types, the separate kernel approach requires less time for assembly than the same kernel approach. This observation is counter-intuitive, since the same kernel approach obviates the need to read and write the elemental stiffness matrices to the global memory of GPU. However, in the same kernel approach more computational load has been assigned to each of the threads of GPU and thread synchronization is performed that can make this approach inferior than the separate kernel approach. In table 3.5, details of the element type, number of nodes and elements, runtime of different stages and global memory usage are listed for the connecting rod example.



(a) HEX20 (b) TET4

Figure 3.17: Matrix generation time for (a) HEX20 and (b) TET4 type elements based on position of the numerical integration kernel for connecting rod example.

Table 3.5: Element type, number of nodes, number of elements, runtime and memory used for storing the global stiffness matrix on GPU global memory are presented for example 2. S1, S2 and S3 under the runtime column denote mesh preprocessor, index computation and value computation stages respectively.

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

3.7.3 Closure

A three-stage FEA matrix generation strategy has been presented for unstructured meshes on GPU. Numerical integration and assembly operations have been implemented for same kernel and separate kernel approaches. This step of FEA was chosen for acceleration because of several studies reporting it to be the most time-consuming step following the linear solver, for which several optimized parallel libraries exist. A novel approach of dividing the workload into a mesh-preprocessor, an index computation stage, and a value computation stage was presented and simulations were run for meshes having up to 3 million DOFs on a single GPU with two different element types.

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

second and third stages respectively. For these two stages, a generalized algorithm with different sparse formats was presented.

The proposed strategy was tested on two examples and it was found to outperform the sharedNZ implementation of Cecka et al. (2011), showing a speedup of 4× to 6× for all element types over a wide range of mesh sizes. Higher order hexahedral elements are found to achieve more speedup than the low order tetrahedral elements on a particular benchmark. The mesh preprocessor stage was found to consume a very less amount of time with respect to the total assembly time. The index computation kernel was found to consume significantly less percentage of total assembly especially for the lower order element. It was also concluded that performing assembly and numerical integration in the same kernel performed worse than having separate kernels for the two processes. This difference is more pronounced for lower order elements.

Although assembly-based methods in FEA are relatively simpler to implement, they can become intractable for large problems, where, despite using the most optimized sparse storage schemes, the memory requirements and cost of data movement become prohibitively high. This leads to another class of solvers called matrix-free or assembly-free methods, where all the FEA computations are carried out at the element level, obviating the need to explicitly store the entire global stiffness matrix at any point of the application. Matrix-free methods are discussed in the following chapters in the context of large-scale FEA acceleration of topology optimization problems.

Chapter 4

Mesh reduction strategy for structural topology optimization

In the second objective of the thesis, we aim to combine the algorithm-level and HPC-based modifications in order to significantly reduce the computational cost of density-based topology optimization using solid isotropic material with penalization (SIMP) method.

Two of the most successful algorithm-level modifications for GPU-based large-scale topology optimization are the use of matrix-free FEA methods and the reduction of design variables by a local update strategy for the optimization algorithm. In this work, these two algorithm-level modifications are combined with efficient GPU-based acceleration using a novel mesh reduction strategy that aims to reduce the computational complexity of conventional structural topology optimization. In the proposed strategy, the effective number of design variables is reduced by using the concept of *active* nodes and *active* elements in the finite element mesh. A node is considered to be *active* if it is a part of at least one element with non-zero density. An element, on the other hand, is considered to be *active* if it contains at least one *active* node. Nodes and elements that do not satisfy these conditions, are considered *inactive*, and can be expelled from the computation without significant effect on the final outcome. Introducing this concept to GPU-accelerated density-based topology optimization algorithms is not straightforward due to the parallel nature of the computation and the need to dynamically update the finite element mesh at every optimization iteration. Another important challenge is locating the boundary condition nodes after performing mesh reduction to remove the *inactive* nodes and elements on GPU. This issue is addressed by introducing a novel mesh numbering scheme to facilitate parallel identification of *active* nodes using the proposed GPU-based algorithm. The preconditioned conjugate gradient (PCG) solver is further developed using the proposed strategy and the numbering scheme.

4.1 FEA and matrix free PCG

In order to find the responses of a structure under the given loading conditions, FEA needs to be performed at each iteration of topology optimization. Typically, regular grids are used for topology optimization in the literature (Martínez-Frutos et al., 2017, Sharma et al., 2011, 2014, Sharma and Deb, 2014, Ram and Sharma, 2017). In the present work also, a structured grid with elements of identical dimensions is used. This ensures that the elemental stiffness matrices for all the elements are numerically identical and, hence, need to be calculated only once for the entire topology optimization process. This elemental stiffness calculation is done at the host, and the data is sent to GPU at the beginning. A matrix-free approach is taken for solving the linear system of equations after applying the boundary conditions. This is an on-the-fly approach in which the entire final global stiffness matrix is never constructed. The preconditioned conjugate gradient method is used as the solution technique. The details of the matrix-free PCG with Jacobi pre-conditioner (Shewchuk, 1994) is given in algorithm 8.

Algorithm 8: Preconditioned conjugate gradient

Input : F: global load vector; K: global stiffness matrix; M: preconditioner
Output: $[x]$: array of displacements ;

1 $i \leftarrow 0$;
2 $r \leftarrow b - Ax$; SpMV | SAXPY
3 $d \leftarrow M^{-1}r$;
4 $\delta_{new} \leftarrow r^T d$; IP
5 $\delta_0 \leftarrow \delta_{new}$;
6 **while** $i \leq i_{max} \wedge \delta_{new} > \epsilon^2 \delta_0$ **do**
7 $q \leftarrow Ad$; SpMV
8 $\alpha \leftarrow \frac{\delta_{new}}{d^T q}$; IP
9 $x \leftarrow x + \alpha d$; SAXPY
10 $r \leftarrow r - \alpha q$; SAXPY
11 $s \leftarrow M^{-1}r$;
12 $\delta_{old} \leftarrow \delta_{new}$;
13 $\delta_{new} \leftarrow r^T s$; IP
14 $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$;
15 $d \leftarrow s + \beta d$; SAXPY
16 $i \leftarrow i + 1$;
17 **end**

It can be seen from algorithm 8 that it consists of several linear algebraic operations which contribute to all the computations in the algorithm. The following three different types of linear algebraic operations are identified in this algorithm.

- Sparse matrix-vector multiplication (SpMV)
- Inner product (IP)
- A X plus Y (SAXPY)

From lines 1 to 5, exactly one SpMV, one IP and one SAXPY operation is performed. Within the loop (between line 6 and line 17), one SpMV, two IPs and three SAXPYs are performed in each iteration. Among these operations, SpMV consumes the most amount of time. The matrix-free SpMV is shown in algorithm 9 that is adapted from (Markall et al., 2013). The GPU kernel is designed in a node-by-node manner for executing SpMV. This implementation is referred to as the standard implementation in the rest of the paper. Inside the kernel, in line 1, a unique thread id is assigned to each node of the mesh. Following this, the elemental stiffness matrix is loaded into the shared memory in line 2. Thereafter, a thread synchronization is performed in line 3 to ensure that the process of copying into shared memory is complete. For each node, a loop is launched (line 6) that iterates through all of its neighboring elements. Another nested loop at line 10 iterates through all four nodes of a particular element. In lines 12 - 15, the multiplication is performed at the nodal level for each neighboring element of the node assigned to the thread. The results are written to the global memory in lines 18 and 19. In the next section, the proposed mesh reduction strategy along with its implementation for matrix-free PCG is discussed.

Algorithm 9: Standard matrix-free SpMV kernel for CG

Input : $[v]$: vector to be multiplied; $[Ke]$: elemental stiffness matrix; n: total nodes
Output: $[result]$: result of SpMV operation;

1 $tx = blockIdx.x \times blockDim.x + threadIdx.x$;
2 Load elemental stiffness in shared memory;
3 __syncthreads();
4 **if** $tx < n$ **then**
5 $eId = tx$;
6 **for** All neighboring elements of tx **do**
7 $e = elementNumber$;
8 $l = localposition$;
9 $d = density[e]$;
10 **for** $j \leftarrow 0 : 4$ **do**
11 $p = connect[4e + j]$;;
12 $temp += v[2p] \times (sh\_Ke[2l][2j] \times pow(d, penal))$;
13 $temp1 += v[2p] \times (sh\_Ke[2l + 1][2j] \times pow(d, penal))$;
14 $temp += v[2p + 1] \times (sh\_Ke[2l][2j + 1] \times pow(d, penal))$;
15 $temp1 += v[2p + 1] \times (sh\_Ke[2l + 1][2j + 1] \times pow(d, penal))$;
16 **end**
17 **end**
18 $result[2 \times tx] = temp$;
19 $result[2 \times tx + 1] = temp1$;
20 **end**

4.2 Proposed strategy

The key idea behind the present implementation is a mesh reduction strategy that reduces the dimensions of the mesh in each iteration of the SIMP-based topology optimization. In the density-based topology optimization methods, each element in the mesh has a density variable that is updated during the design update stage. During the SpMV operation of FEA, the elemental densities are multiplied to the elemental matrices. The elemental densities are often initialized to the volume fraction. Through the design update, while some of these densities reach their minimum value, some are increased to varying extents. During a particular topology optimization iteration, if the density of a certain element becomes zero, it does not contribute to the FEA process due to its elemental contribution becoming zero. With this knowledge, we separate all the elements in the mesh into *active* and *idle* elements based on their densities being non-zero and zero, respectively. Here, it is noted that "zero" density, in this context essentially means the minimum density set for the optimization algorithm. By design of the optimality criteria update scheme in equation (2.10), the values of densities cannot reach a value lower than the minimum density. Furthermore, we also separate the nodes in a similar manner into *active* and *idle* nodes. If all of the neighboring elements of a certain node have zero density, we consider the node to be *idle*. This is illustrated in figure 4.1 by using an example mesh where active and idle elements and nodes are shown using different colors. The active grid presents non-zero elemental density and the idle grid represents zero elemental density.

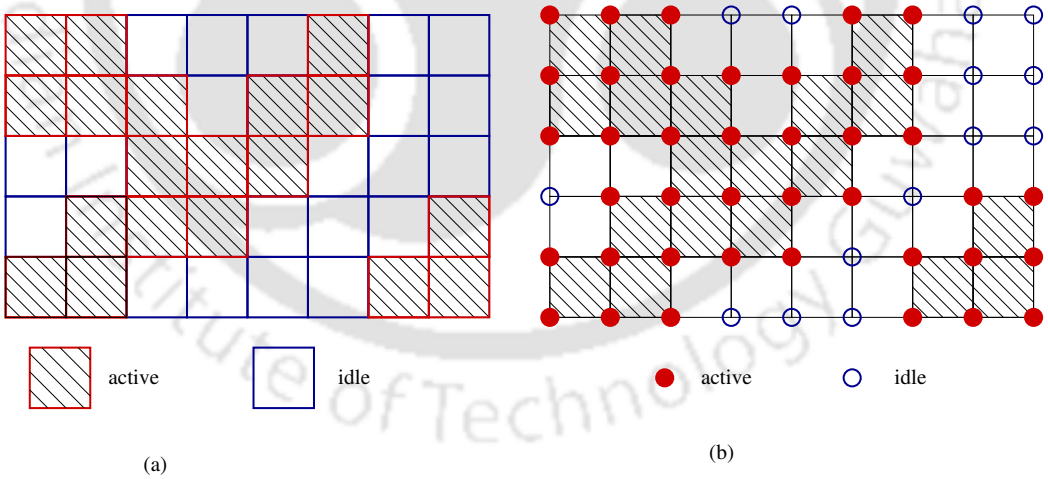


Figure 4.1: Active and idle (a) elements, and (b) nodes for an example topology optimization mesh. The active grid represents non-zero elemental density and the idle grid represents zero elemental density.

Figure 4.2 shows the variation of the number of active nodes with topology optimization iteration for a cantilever domain with one million nodes. It can be observed that within mere $10-15$ iterations, the numbers of active nodes dropped to approximately 40% of the initial number. In other words, after $10-15$ iterations, approximately 60% of the total nodes stop contributing to the FEA and can be

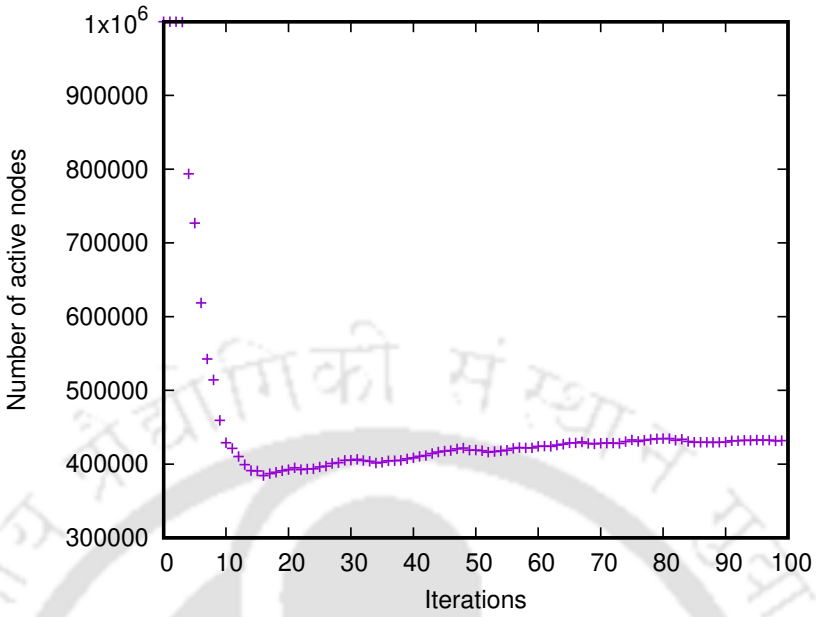


Figure 4.2: Variation of number of active nodes with iterations of topology optimization for a cantilever domain containing one million nodes.

neglected from the entire process. This observation is the motivation behind the present implementation. The aim of the present study is to adaptively modify the finite element meshes to include only the *active* nodes in each topology iteration and perform FEA on the updated mesh. This is achieved by keeping a list of the active nodes and updating it in each iteration. This list is fed to all the GPU kernels during FEA, and all the matrix and vector operations are performed only on the active nodes of that particular iteration instead of all the nodes in the mesh. For the implementation, four major modifications over the standard implementation algorithm (Markall et al., 2013) are proposed in the following subsections.

4.3 Mesh Numbering

In order to implement the mesh reduction strategy to topology optimization, one key difficulty is the implementation of Dirichlet and Neumann boundary conditions. For the standard implementation, the nodes are known beforehand on which the boundary conditions are to be applied. But in case of the mesh reduction strategy, the mesh gets shortened and the FEA is run on the active nodes only. In this case, although the boundary condition node numbers are known, but their location gets distorted on the shortened arrays for the reduced mesh. Hence, to find the boundary nodes, expensive search operation needs to be performed on GPU. It is noted here that this problem is exclusive to parallel implementation using GPU only. On a CPU, because of the sequential nature of the computation, the

boundary nodes can be accessed at all times without any difficulty.

A modified node numbering scheme is adopted in this study as a remedy to this issue, which is shown in figure 4.3. The key idea behind the renumbering is that the Dirichlet and Neumann boundary nodes should always have the beginning node numbers starting from 0. This will ensure that the nodes are always found with ease without the need for any search operation even for the reduced mesh since these boundary nodes would never become idle by design of the algorithm. During meshing, two variables $numBC1$ and $numBC2$ are created to store the numbers of nodes where Dirichlet and Neumann boundary conditions are applied. These variables are passed to FEA for boundary condition implementation. As shown in figure 4.3 for the cantilever beam (I), L-shaped beam (II) and displacement inverter mechanism (III), the standard numbering scheme is shown in (b), whereas the modified scheme is shown in (c). The boundary nodes are marked with red dots. In case of modified numbering scheme of the cantilever, node $1-5$ are the nodes with the displacement boundary condition. Node 6 is the only node with force boundary condition. Hence in this case, the values of $numBC1$ and $numBC2$ are 5 and 1, respectively. For the L-shaped beam, displacement boundary conditions are on node $1-3$ and the force boundary condition is on the node $4-6$. Hence in this case, $numBC1$ and $numBC2$ both have a value of 3. For the inverter mechanism, node $1-9$ are the nodes where the displacement boundary conditions are specified. Hence, the value of $numBC1$ becomes 9. Node 1 is the only node with force boundary condition. Since node 1 is already contained in the list of displacement boundary condition nodes, the value of $numBC2$ is kept 0. It should be noted here that a renumbering scheme such as the one used in this study would affect the banding of the stiffness matrix. However, since an iterative solver is used in this study, the solution is not affected.

4.4 Computation of Active Nodes

Since the present implementation follows a node-by-node implementation for matrix-free CG, an array containing all the active nodes in the mesh must be prepared before executing FEA. This is performed by a combination of kernel calls and thrust operations as shown in algorithm 10. In line 1 of the algorithm, a thrust device vector `t_aN` is declared of size equal to the total number of nodes (n) in the mesh and the raw pointer is extracted in line 2. Next, a call to *thrust::sequence* is made to fill the array with a sequence of numbers from 0 to $(n-1)$. These numbers signify the node numbers. This is demonstrated using an example 2D mesh with 9 elements in figure 4.4. On the left, the mesh is shown where the active elements are greyed out. It can be seen that only four elements are *active*. Corresponding to these four elements, there are a total of ten nodes that are *active*, that are node numbers 1, 2, 3, 5, 6, 7, 10, 11, 14, and 15. On the right of the figure, thrust array `t_aN` (or its raw pointer `aN`) is shown after calling *thrust::sequence*. Following this, a node-by-node kernel call is made on line 4 (`aN_kernel`) of algorithm 10 with total number of threads equal to n. Inside the kernel, each

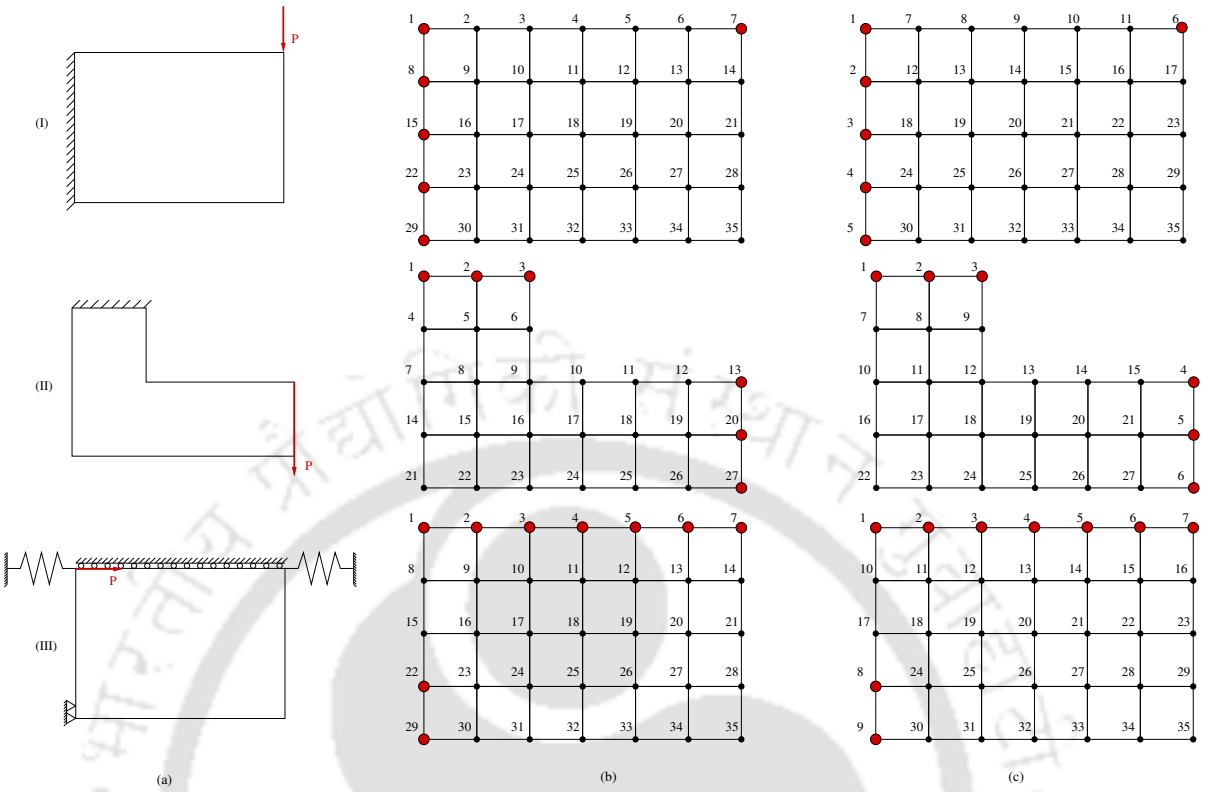


Figure 4.3: Problem domain is shown in (a). The standard and modified numbering schemes are shown in (b) and (c) for (I) cantilever beam, (II) L-beam and (III) displacement inverter mechanism.

thread checks the neighboring elements' densities of its assigned node. If a node has all the neighboring nodes with zero densities, the corresponding node is deemed idle and the number in `t_aN` is changed to an arbitrary negative value, say (-5). This is shown in the figure where node numbers 4, 8, 9, 12, 13, and 16 are all changed to -5. Following this a *thrust::count_if* is called to calculate the number of positive numbers in array `t_aN`. This call returns an integer, which is the number of active nodes in the current mesh (*count_active*). For the example mesh in figure 4.4, the value of *count_active* is 10. Finally in line 6, *thrust::remove_if* is called to remove all the negative entries (-5 signifying idle nodes) from `t_aN`. This is also shown in figure 4.4 in which all the negative values are removed from the array after calling *thrust::remove_if*. The resulting thrust array `t_aN` (or its raw pointer `aN`) now contain all the active node numbers from 0 to *count_active*.

4.5 PCG with Active Nodes

For the proposed implementation, the PCG algorithm (as shown in algorithm 8) is modified to only work with the active nodes in the mesh at that particular iteration. After the list of active nodes is prepared as discussed in section 4.4, the PCG algorithm starts as shown in algorithm 8. The key

Algorithm 10: Computing active nodes

Input : n: Total number of nodes; e: Total number of elements; $D[e]$: Array containing density of all the elements

Output: $[aN]$: array containing the active nodes; new_count: number of active nodes

1 $thrust :: device\_vector < int > t\_aN[n]$;
2 $aN \leftarrow thrust :: raw\_pointer\_cast(t\_aN)$;
3 $thrust :: sequence(t\_aN.begin(), t\_aN.end(), 0)$;
4 $aN \leftarrow aN\_kernel(aN)$;
5 $new\_count = thrust :: count\_if(t\_aN.begin(), t\_aN.end(), is\_positive())$;
6 $thrust :: remove\_if(t\_aN.begin(), t\_aN.end(), is\_negative())$;

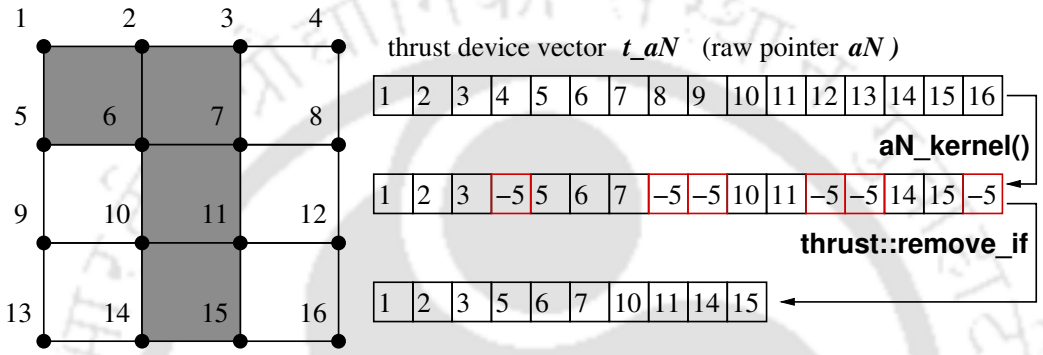


Figure 4.4: Calculation of active nodes of the example mesh.

difference is that, now instead of declaring the arrays r in line 2, x in line 2, d in line 3 and q in line 7 with a size of $(n \times ndof)$, they are declared with a reduced size of $(size[aN] \times ndof)$. Here, n is the total nodes in the mesh and $ndof$ is the DOF per node. The SpMV operations are performed using a custom GPU kernel, whereas, IP and SAXPY operations are performed using thrust on the reduced size arrays. These reduced size arrays also include the displacement array resulting from FEA. Since the other parts of the SIMP algorithm are designed to work with the full sized displacement array, a thrust scatter operation is performed after each FEA to increase the size of the array from $(size[aN] \times ndof)$ to $(n \times ndof)$. Here, the `aN` array is used as the map required by the thrust scatter operation. The details of SpMV kernel and thrust operations are given in the following sections.

4.5.1 SPMV kernel

The matrix-free SpMV kernel with the reduced mesh strategy is shown in algorithm 11. The array containing the active nodes `aN` (as discussed in section 4.4) is computed and sent to GPU before launch of the kernel as shown in line 1. A none-by-node implementation is used for uniform comparison with the standard implementation. At the start of the kernel, the elemental stiffness matrix is loaded into the shared memory as shown in line 3. This is followed by a barrier synchronization in line 4 for ensuring that the data is copied entirely before any computation can be performed. This kernel is launched

Algorithm 11: Proposed matrix-free SpMV kernel for CG

Input : $[v]$: vector to be multiplied; $[Ke]$: elemental stiffness matrix; `[aN]`: array of ative nodes

Output: $[result]$: result of SpMV operation;

```
1  Compute [aN] using algorithm 10;
2  tx = blockIdx.x × blockDim.x + threadIdx.x;
3  Load elemental stiffness in shared memory;
4  __syncthreads();
5  if tx < count_active then
6  |  eId = aN[tx];
7  |  for All neighboring elements of tx do
8  |  |  e = elementNumber;
9  |  |  l = localposition;
10 |  |  d = density[e];
11 |  |  for j ← 0 : 4 do
12 |  |  |  p = reverse_aN[connect[4 × e + j]];
13 |  |  |  temp += v[2p] × (sh_Ke[2l][2j] × powf(d, penal));
14 |  |  |  temp1 += v[2p] × (sh_Ke[2l + 1][2j] × powf(d, penal));
15 |  |  |  temp += v[2p + 1] × (sh_Ke[2l][2j + 1] × powf(d, penal));
16 |  |  |  temp1 += v[2p + 1] × (sh_Ke[2l + 1][2j + 1] × powf(d, penal));
17 |  |  end
18 |  end
19 |  result[2 × tx] = temp;
20 |  result[2 × tx + 1] = temp1;
21 end
```

with threads equal to the number of active nodes in the mesh (*count_active*) in that iteration of SIMP method as shown in line 5. The element id *eId* is calculated from the thread id *tx* in line 6. Following this, a loop is called that iterates through all the neighboring elements of the assigned node in line 7. The values of the global element number, local position of the node in that element and the elemental density are loaded in lines 8, 9, and 10. Following this, another loop is called that iterates through all the nodes of a neighboring element in line 11. Inside this loop, the node-level multiplications and additions are performed from line 13 to line 16. Finally, the results are stored on the global memory in the *result* array in lines 19 and 20.

It is noted that all the arrays in the CG algorithm are shortened according to the size of *count_active*. This includes the array $v[]$ to which the matrix is to be multiplied. This creates out-of-bound memory accesses for lines 13 to 16 in algorithm 11. The reason is that in line 12, the value of node number p, that is read from the connectivity matrix can range between anything from 0 to n. However, for accessing the shortened $v[]$ vector, the maximum permissible value of p is *count_active* instead of n. To remedy this problem, we introduce another map array called `reverse_aN` that is computed at the start of CG iterations. It has the size of maximum nodes in the mesh (n). It is calculated from the `aN` array in the following manner.

$$\texttt{reverse\_aN}[\texttt{aN}[i]] = i, \quad \text{for } i \leftarrow 1 : n \tag{4.1}$$

It can be seen that by design of the proposed implementation, only small changes need to be made to the standard GPU kernel for utilizing the mesh reduction strategy. In case of a element-by-element, or DOF-by-DOF implementation of matrix-free, one needs to construct the list of active elements or active DOFs in a similar manner as the nodes. However, due to the node-by-node implementation, the well-known problem of race condition is not observed due to the absence of sharing. In case of other thread allocation strategies, this issue needs to be handled.

4.5.2 Thrust operations

Apart from the SpMV operation, all the vector operations in PCG are performed using thrust. In algorithm 8, the lines (2, 4, 8, 9, 10, 13, 15) where SAXPY and IP are mentioned, are the ones signifying vector operations on GPU using thrust. In the standard implementation, these operations are performed on arrays of size $(n \times 2)$. However, on the proposed implementation, this number is reduced to $(count\_active \times 2)$. This strategy imparts two key benefits to the matrix-free PCG. Firstly, it reduces the memory required by thrust to allocate the arrays in the standard implementation. Secondly, the number of floating point operations are also significantly reduced. However, one key issue after shortening all the arrays is that the locations of the boundary nodes get distorted and undesirable search operations are required in order to find the location of those nodes. This is handled by renumbering the mesh so that the boundary nodes are always stored starting from node number zero as discussed in section 4.3. The other nodes are not numbered unless all the boundary nodes are numbered already. At the end, the shortened displacement array needs to be expanded into the full displacement array so that it can be used by the topology optimization operators. This is done by the *thrust::scatter* operation at the end of each FEA. The map required by the scatter operation is the `aN` array of active nodes.

4.6 Sensitivity, filter and design update

Apart from FEA, the other key stages in topology optimization are sensitivity analysis, mesh filtering and design update. In the present implementation, these three stages are computed on GPU. Despite consuming small amounts of time, acceleration of these stages is important because it facilitates the entire optimization algorithm to be performed on GPU, obviating the need for repeated and expensive CPU-GPU memory transfer. Furthermore, these three stages fall in the category of *embarrassingly parallel* problems in the parallel computing terminology. This is because of the absence of any dependency or need for communication between the parallel tasks or their individual results. For the acceleration of these three stages, the strategy presented by Martínez-Frutos et al. (2017) is implemented.

4.6.1 Sensitivity analysis

The sensitivity analysis is performed on GPU using an element-by-element kernel, where each thread is assigned to one element of the finite element mesh. As explained by Martínez-Frutos et al. (2017), the element-by-element decomposition of the sensitivity computation exploits the parallelization potential of GPU architecture where each thread is responsible for computing the multiplication of each elemental $u_i^T K_i u_i$ in equation (2.12). The objective function is computed using the thrust library.

4.6.2 Mesh filtering

In the present implementation, the mesh sensitivity filter is applied by a combination of custom kernel and thrust calls and is computed entirely on GPU. Similar to the study by Martínez-Frutos et al. (2017), a one-dimensional grid is launched by the filter kernel where each thread is assigned to one element of the mesh. The regularity of the mesh is utilized for efficiently finding the neighbor set of each element.

4.6.3 Design update

The update of elemental densities is performed on GPU according to the strategy presented by Martínez-Frutos et al. (2017). Two custom kernels and thrust reduction operations are used to parallelize this stage on GPU. Both the kernels follow an element-by-element strategy, where one thread is assigned to one element of the finite element mesh.

In the next section, the performance analysis of the proposed strategy is discussed.

4.7 Results and Discussion

The performance of the mesh reduction strategy on GPU is now evaluated using three benchmark problems. The computation analyses are performed using a workstation with Intel Xeon ES1650 (6 core, 3.2 GHz) processor, 12 GB RAM, and K40c NVIDIA GPU. The GPU has 12 GB of global memory with 15 multiprocessors and 192 cores per multiprocessor. The parameters used are listed in table 4.1. In figure 4.5, the problem domains with boundary conditions are shown along with the final topologies obtained using the proposed mesh reduction strategy. For performance analysis of the proposed strategy, it is compared with the standard implementation shown in algorithm 8 and algorithm 9.

Figure 4.6 shows the global memory usage of all three problems with increasing mesh size. It can be seen that for both cantilever and L-beam, the global memory usage is significantly lower than the inverter mechanism. Furthermore, the amount of memory required is almost exactly the same for both cantilever and L-beam for all the mesh sizes. Hence, for the same problem, the total amount of global

Table 4.1: Parameters for CG and topology optimization

Parameter	Value
CG tolerance	1e-3
CG max iterations	15,000
Volume fraction	0.3
Penalty	3
Min radius	3.0
SIMP tolerance	1e-3
SIMP max iterations	100

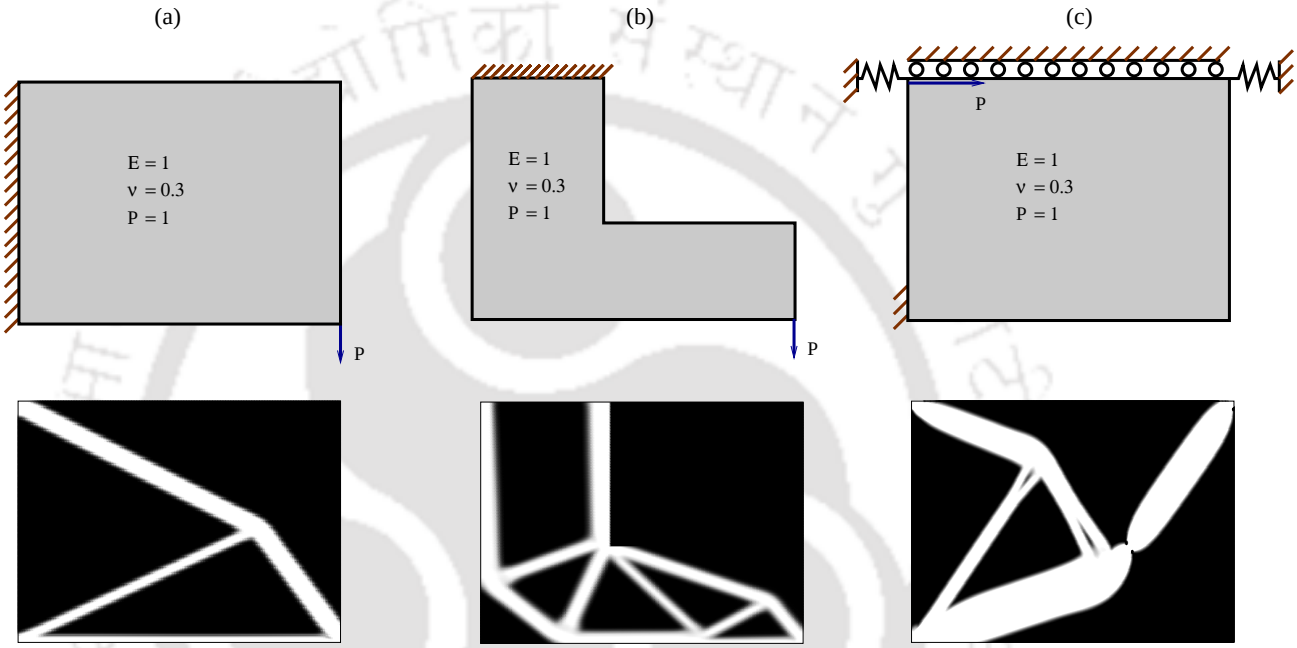


Figure 4.5: Problem domains for (a) cantilever (b) L-beam and (c) displacement inverter mechanism along with optimal topologies achieved using the proposed mesh reduction implementations are shown.

memory is correlated with node numbers and independent of the geometry of the domain. Furthermore, a flat portion is observed between node numbers 100k and 150k followed by a steep incline for all the three problems.

Figure 4.7 shows the initial and reduced node numbers of four different mesh sizes for all three problems using clustered bars. The reduced nodes are taken for the final iteration of topology optimization. Meshes are chosen for all three problems such that all of them have approximately the same node numbers. The green bar denotes the initial node numbers for all problems. The blue and red and yellow bars denote the number of nodes at the last iteration of SIMP method. On the same plot, the blue and red and yellow lines denote the percentage node reduction of cantilever, L-beam, and displacement inverter mechanism problems, respectively. This plot reveals that the amount of reduction achieved is more in case of cantilever than the other two problems. It is further observed that the percentage reduction is

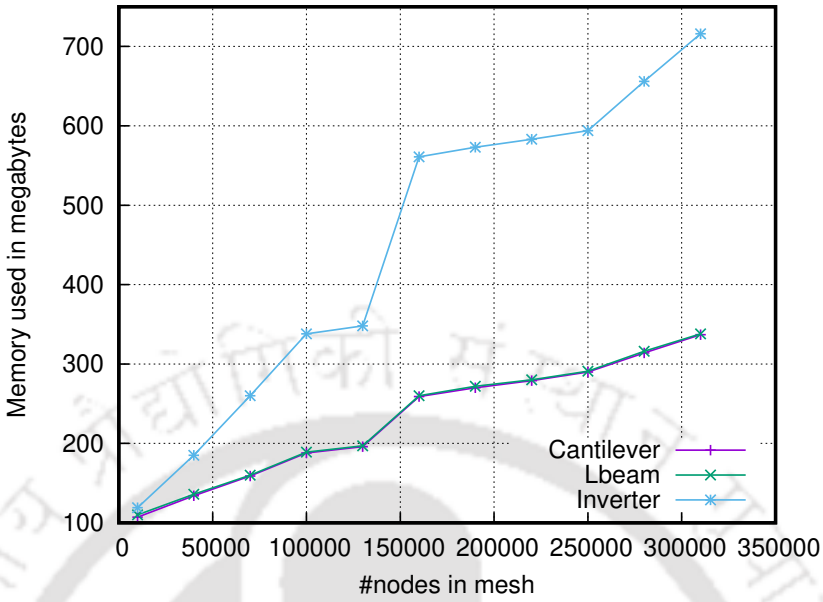


Figure 4.6: Global memory used for all three problems

Table 4.2: Node numbers of initial mesh sizes for all three problems

Mesh	cantilever	L-beam	Inverter
mesh_1	40,000	40,501	39,762
mesh_2	130,321	130,501	130,050
mesh_3	219,961	220,195	220,448
mesh_4	310,249	310,527	310,472

more pronounced at higher mesh sizes for both the cantilever and L-beam problems, whereas, for the inverter mechanism problem, a more flat trend is observed. The initial and reduced node numbers for all four mesh sizes are given in table 4.2 and 4.3, respectively.

Table 4.4 lists the average time of one iteration of SIMP-based topology optimization broken into three main parts, FEA, mesh filter and density update. Execution time for both the implementations for all three problems are presented. Mesh dimensions are chosen in a manner so that all the six meshes have close to the same number of (310k) nodes for the validity of comparison. From table 4.4 it can be clearly seen that FEA consumes majority of the total execution time in all cases. It should be noted that the mesh filter and density update stages are computed on GPU in parallel. Furthermore, the most

Table 4.3: Node numbers of reduced mesh sizes for all three problems

Mesh	cantilever	L-beam	Inverter
mesh_1	10,087	18,295	14,448
mesh_2	19,158	45,331	46,118
mesh_3	26,537	62,658	77,095
mesh_4	32,089	76,332	116,332

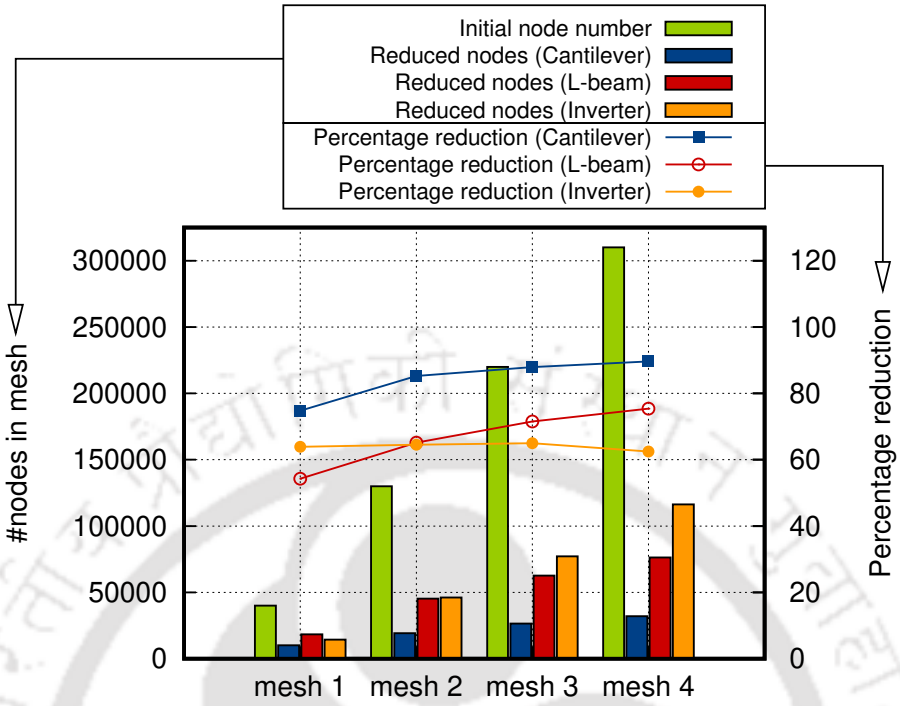


Figure 4.7: Original and reduced nodes are shown for cantilever, L-beam and inverter mechanism problems. The percentage reduction after the final iteration is also shown.

Table 4.4: Average time (in seconds) per iteration of SIMP-based topology optimization for FEA, mesh filter and density update is presented. All six meshes are of the same size containing approximately 310k nodes each.

Implementation	FEA	Mesh filter	Density update
canti_standard	17.05	0.001	0.013
canti_reduced	3.36	0.001	0.013
lbeam_standard	25.45	0.005	0.013
lbeam_reduced	9.75	0.005	0.013
inverter_standard	10.02	0.004	0.05
inverter_reduced	4.17	0.004	0.05

time consuming part of the FEA is the PCG. Figure 4.8 shows the iteration-wise convergence of PCG for one iteration of SIMP-based topology optimization.

As discussed earlier, FEA is the single most time consuming stage in structural topology optimization. The time taken by FEA directly affects the total time consumed. The number of active nodes in a particular iteration directly impacts the time consumed by FEA in that iteration. In figure 4.9, both the number of active nodes and time taken in FEA are plotted for every iteration of SIMP-based topology optimization. A cantilever domain with 1M nodes is chosen for this plot. It can be observed that the number of active nodes drops sharply within 20 iterations of SIMP method and becomes less than 10% of the total node number. After the 20th iteration, there can be seen very small variation in the number of active nodes. The green lines in this plot represent the time taken by FEA in a particular iteration.

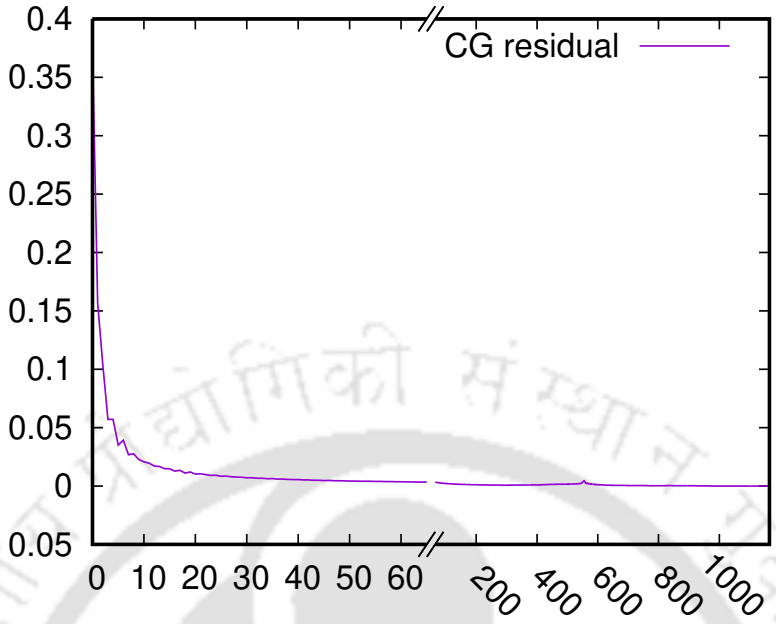


Figure 4.8: Iteration-wise convergence plot of CG residual

This value is seen to rise with a steep slope and decrease rapidly around iteration #20. Thereafter, little variations are observed but the execution time does not increase. Figure 4.10 shows the same plot for a L-beam mesh with 1M nodes. Similar to cantilever, at around 20th iteration, the number of active nodes becomes approximately 15% of the total nodes. After the 20th iteration, the number again remains the same with very small variations. Similar to cantilever problem, a steep rise followed by a sharp decline in FEA time is observed till iteration #20. After that, the time becomes stable and does not increase significantly. Figure 4.11 shows the variation of active nodes and FEA time with increasing iterations for the displacement inverter mechanism. It can be seen that after approximately 20 iterations, the number of active nodes drop to less than 40% of the initial numbers. For the rest of the iterations, only slight decrease in the active nodes can be observed. Similar to the cantilever and L-beam problems, the FEA time is seen to decrease sharply till iteration 20 and thereafter only slight variations are noted.

Comparing the results of cantilever in figure 4.9, L-beam in figure 4.10, and inverter mechanism in 4.11 it can be seen that the drop in number of active nodes is the most pronounced in case of cantilever beam problem and the least in case of the inverter mechanism. FEA time is approximately 50%-60% higher in case of L-beam compared to the cantilever. Furthermore, it is observed that the variation of FEA time is much more pronounced in case of cantilever and inverter mechanism compared to L-Beam.

In figure 4.12, the percentage reduction in active nodes at the end of all SIMP method iterations is plotted against increasing nodes in the mesh. It can be observed that the percentage reduction is significantly less in case of L-beam and inverter mechanism compared to cantilever. Furthermore, the

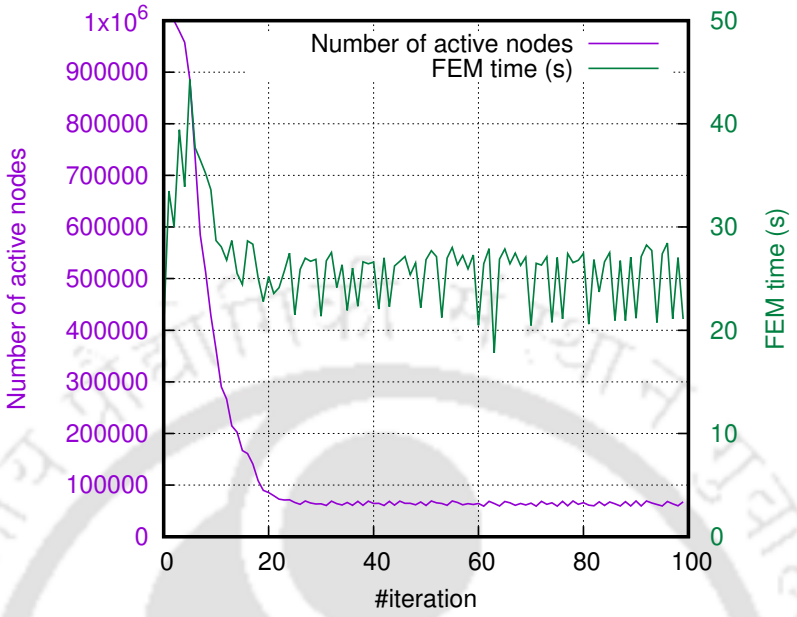


Figure 4.9: Iteration-wise variation in the number of active nodes and time taken by FEA for cantilever problem with mesh reduction strategy.

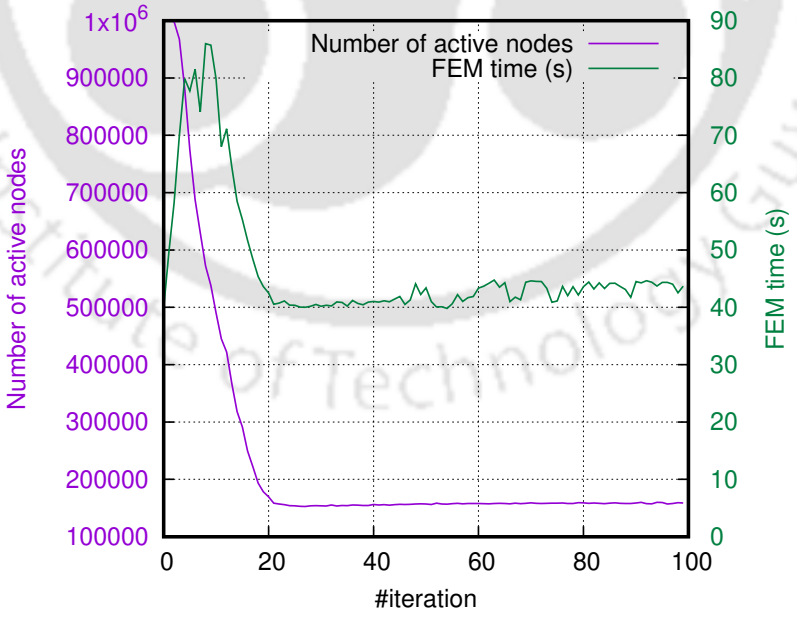


Figure 4.10: Iteration-wise variation in the number of active nodes and time taken by FEA for L-beam problem with mesh reduction strategy.

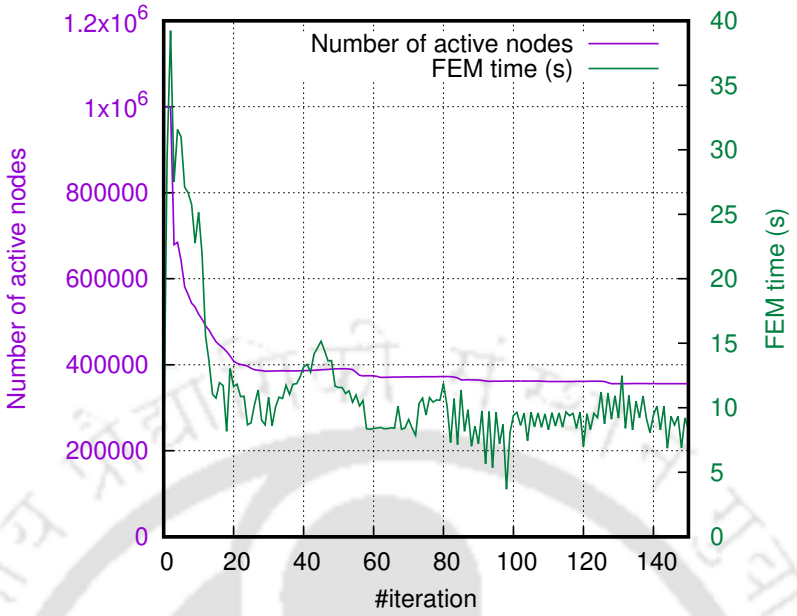


Figure 4.11: Iteration-wise variation in the number of active nodes and time taken by FEA for displacement inverter problem with mesh reduction strategy.

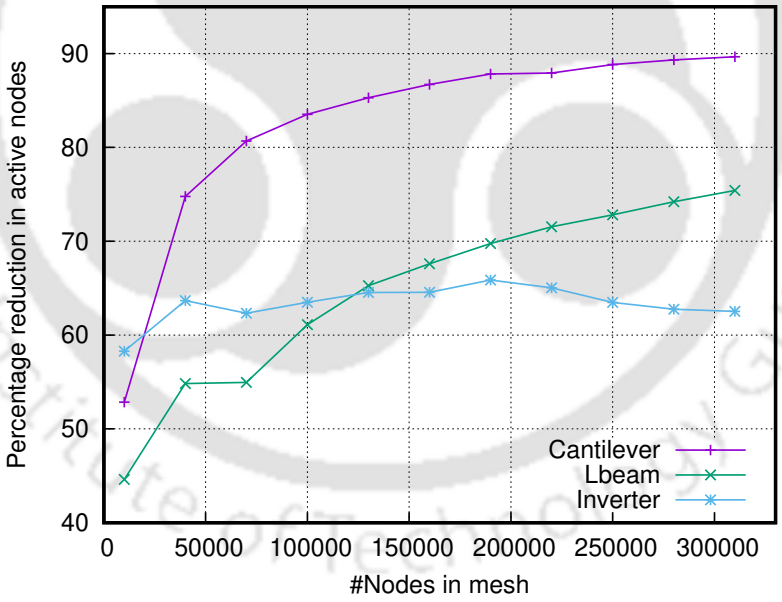


Figure 4.12: Percentage reduction in active nodes with increasing mesh size for cantilever, L-beam and inverter mechanism problems.

reduction percentage is observed to increase with increasing node numbers for cantilever and Lbeam problems, whereas for the inverter mechanism, a flat trend is observed.

In the entire topology optimization, FEA consumes the majority of the execution time. Furthermore, among the different stages of FEA, the solution of the linear system of equations consumes the most

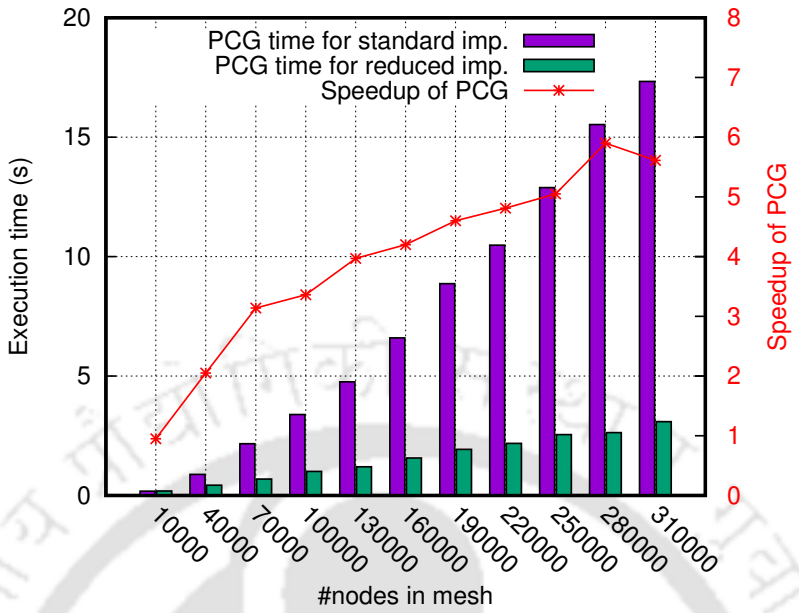


Figure 4.13: PCG time for the standard and proposed implementations for cantilever problem. The speedup of the proposed implementations over the standard GPU implementation is also shown.

time. In the present study, PCG is used as the solution method for solving the system of equations. In this particular implementation, the use of PCG is particularly important because of its matrix-free feature. In figure 4.13, the execution time of PCG is shown for both the standard and reduced-mesh implementation of cantilever beam. In the same plot, a red line is used to show the variation of speedup of PCG time of the proposed implementation over the standard GPU implementation with increasing node numbers. Figure 4.14 shows the same data and observation for the L-beam problem. It can be observed that the speedup starts low for low number of nodes and goes up to approximately 3× and 6× in case of L-beam problem and cantilever beam problem, respectively for higher mesh sizes. A similar plot is drawn for the inverter mechanism in figure 4.15. In this case, PCG speedup values of up to 2× are observed for higher mesh sizes. Both the execution times and speedup values for the inverter mechanism are observed to be lower than the cantilever and L-beam problems. Furthermore, in all three problems, the execution time seems to vary linearly with increasing node numbers.

As discussed in section 4.1, the three main parts of a PCG solver are SpMV, SAXPY and IP. In figure 4.16, the execution time of different parts of PCG solver for the cantilever problem are plotted using stacked and clustered bars. The green bar represents the total time consumed in different steps, whereas the different shades represent different parts of PCG. The red bar similarly denotes time consumed in different parts of PCG in the proposed implementation with different shades of red. It can readily be seen that each stage of the proposed implementation consumes significantly less time than the counterparts of the standard implementation. It can also be observed here that in both implementations, SpMV

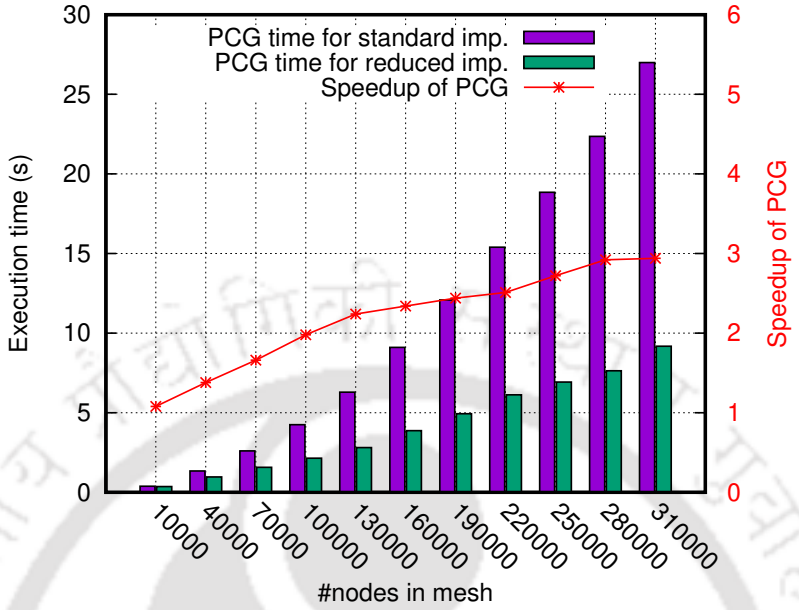


Figure 4.14: PCG time for the standard and proposed implementations for L-beam problem. The speedup of the proposed implementations over the standard GPU implementation is also shown.

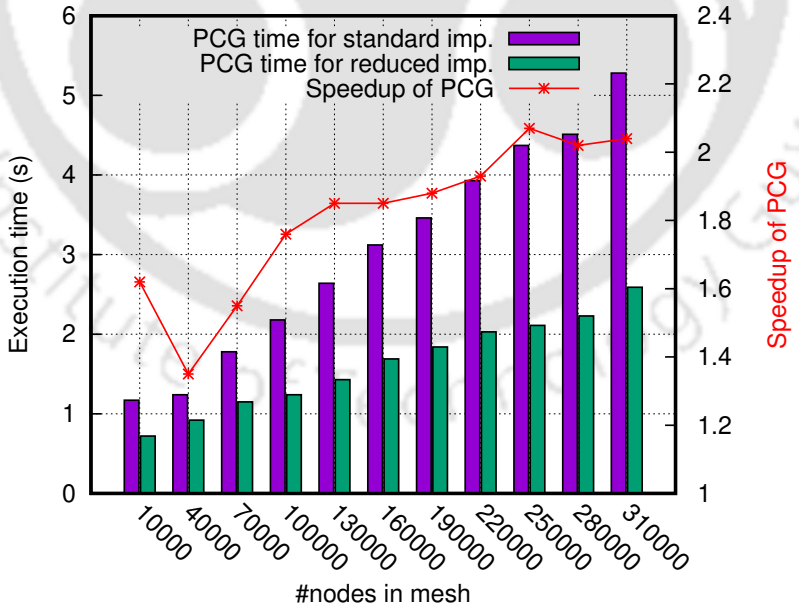


Figure 4.15: PCG time for the standard and proposed implementations for displacement inverter problem. The speedup of the proposed implementations over the standard GPU implementation is also shown.

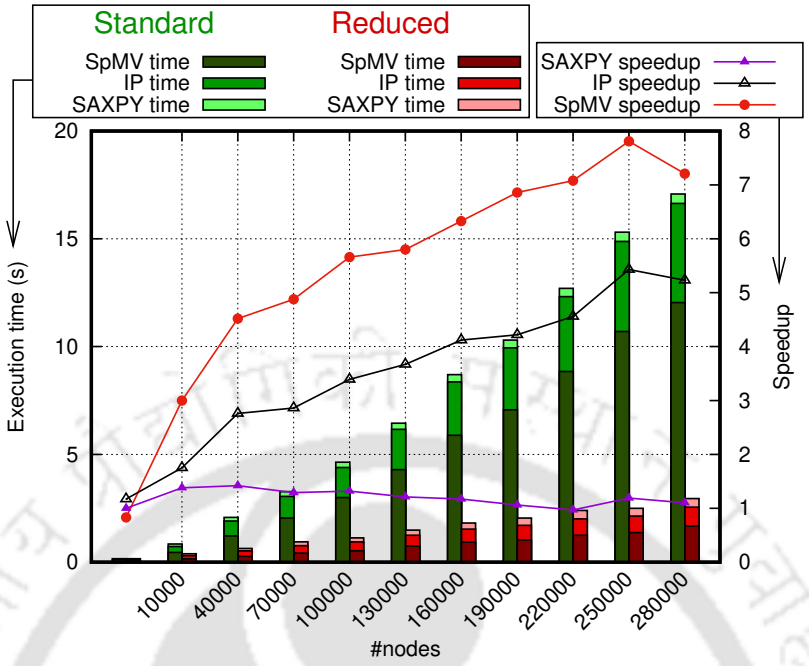


Figure 4.16: Red stacked bar represents the time required by different parts of PCG with mesh reduction in the cantilever problem. The green bar shows the times required by different parts of the standard implementation. The speedups of individual parts in the proposed implementation over the standard implementation are shown using lines.

takes the most amount of time followed by IP and SAXPY, respectively. As a measure of the difference between the times for both implementations, speedup is calculated individually for each of SpMV, IP and SAXPY. These speedups are shown on the same plot using lines. It can be observed that for low mesh sizes, the speedups are all close to 1, which means that the time for both implementations are approximately the same. However, the speedups keep increasing with increasing node numbers. It can be observed that the most speedup is achieved for SpMV, followed by IP and SAXPY. Figure 4.17 shows the same data for the L-beam problem. Again, it is observed that the SpMV consumes the most time followed by IP and SAXPY for both implementations and all mesh sizes. The individual speedups achieved are comparatively lower than those achieved in the cantilever problem. The highest speedup is observed for the SpMV stage followed by IP and SAXPY in all stages. The same plot is drawn for the inverter mechanism in figure 4.15. A similar trend of SpMV consuming the most amount of time followed by IP and SAXPY is observed for all cases. Both the SpMV and IP speedup values are observed to be around 2×, whereas the SAXPY speedups lie in the vicinity of 1× . In all figures 4.16, 4.17, and 4.18 all the execution times are observed to be varying linearly with increasing node numbers.

Table 4.5 lists the time consumed by the mesh filter and density update stage for all three problems. Both the CPU and GPU execution times are listed along with the respective speedups. It can be observed that the speedup values are significantly high compared to the ones achieved in the different

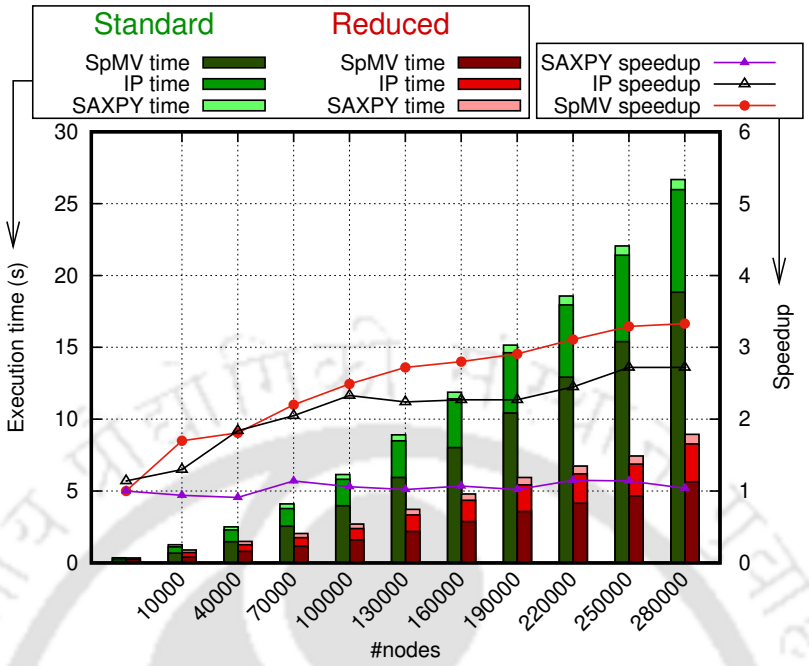


Figure 4.17: Red stacked bar represents the time required by different parts of PCG with mesh reduction in the L-beam problem. The green bar shows the times required by different parts of the standard implementation. The speedups of individual parts in the proposed implementation over the standard implementation are shown using lines.

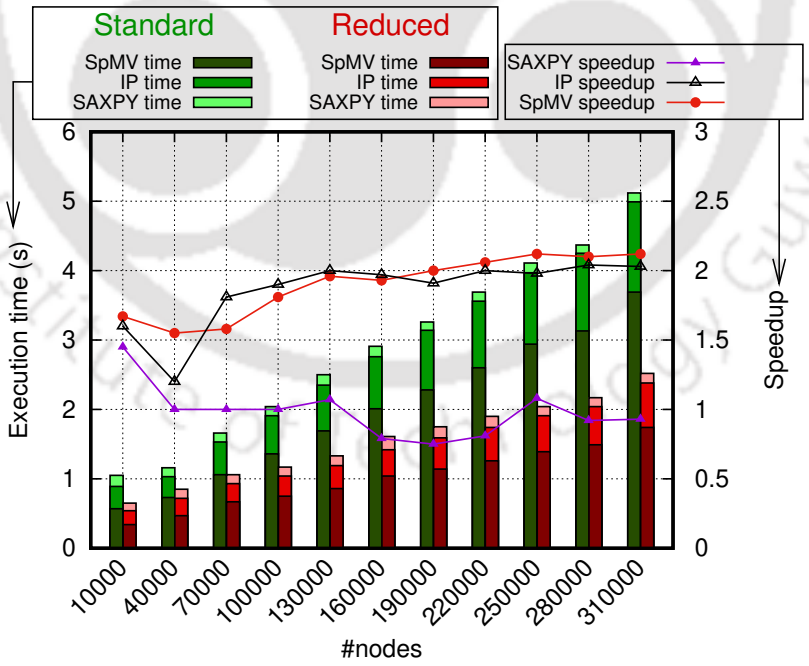


Figure 4.18: Red stacked bar represents the time required by different parts of PCG with mesh reduction in the displacement inverter problem. The green bar shows the times required by different parts of the standard implementation. The speedups of individual parts in the proposed implementation over the standard implementation are shown using lines.

Table 4.5: CPU and GPU execution times for all three problems along with corresponding speedup values are listed. Meshes with similar node numbers are taken for all three problems for a fair comparison.

	Mesh filter			Density update		
	CPU time	GPU time	speedup	CPU time	GPU time	speedup
Cantilever	33.39	0.09	371.00	95.58	1.33	71.86
L-beam	32.66	0.50	65.32	101.82	1.34	75.98
Inverter	127.50	0.38	335.53	859.04	4.95	173.54

stages of FEA. This is due to two primary reasons. The first reason is that the comparison in table 4.5 are done with respect to the CPU implementation, whereas for the previous comparisons of FEA stage timings, the values are compared with a standard GPU implementation. The second reason is that unlike the FEA stage, these stages are *embarassingly parallel*. Due to the absence of need for communication between parallel tasks, greater levels of acceleration is achieved on GPU architecture.

4.8 Closure

A mesh reduction strategy was presented which aims at reducing the computational complexity of structural topology optimization by reducing the active design variables along with acceleration on GPU. These two objectives were fulfilled by the proposed modifications to the PCG and SpMV algorithm, along with a mesh numbering scheme for efficient GPU implementation. The proposed strategy was tested on three benchmark problems from the literature against the standard SIMP using node-by-node GPU-based implementation of FEA. A hotspot analysis of the application revealed more than 90% of the total application time being consumed by FEA. Thus, all the computing effort was concentrated on FEA and more specifically PCG, which is the heart of FEA. Furthermore, the key operations of the PCG process were identified and accelerated on GPU. Special kernel was written for the matrix-free SpMV operation, whereas well-optimized thrust library was used for the inner product and SAXPY operations. The standard PCG was modified with a map and a reverse map array to work with only the active nodes at a particular iteration of SIMP. Furthermore, a custom GPU kernel was written for computing and storing the active nodes at each iteration of SIMP in parallel.

In the performance analysis of the proposed strategy, the first conclusion is somewhat intuitive. The execution time of all the stages for both the implementations and both problems are seen to increase almost linearly with increasing node numbers in the mesh. Such behavior is commonplace in the context of HPC applications and is indicative of good scalability of the method with increasing problem size. Analysis of speedups and execution times reveal that using the proposed algorithm, up to 90% reduction in active nodes is observed within 20 iterations of optimization. As a direct outcome, the execution time of FEA is seen to decrease significantly. From the results, it can be concluded that the percentage reduction in active nodes is a good indicator of the reduction in the total application

time. By analyzing different mesh sizes, it is further concluded that the percentage reduction in active nodes is more pronounced for meshes with higher node numbers. Furthermore, it can be seen that the percentage reduction in active nodes is approximately 15% and 27% less in case of L-beam problem and inverter mechanism problem compared to cantilever problem. The likely reason behind this is the presence of more bars in the final truss-like structure obtained in the case of L-beam and the presence of much thicker bars with higher filter radius in the inverter mechanism compared to cantilever problem. In accordance to the less percentage reduction in active nodes for L-beam and inverter mechanism, the speedups achieved for PCG are $40\% - 60\%$ and $30\% - 50\%$ of the cantilever problem. Thus, it can be concluded that the domain geometry and problem type plays a key role in the amount of nodes that can be reduced from the mesh and thereby, the amount of execution time decreased as well. Comparison of individual execution times and speedup values for SpMV, IP and SAXPY reveal that the highest amount of speedup is observed in case of SpMV, followed by IP and SAXPY. This turns out to be highly appropriate in the context of acceleration, since the execution times of these three stages follow the same descending order. After performing the modifications on the standard implementation and thereby decreasing the total execution time, FEA is still the bottleneck of the application consuming majority of the execution time for both the problems.

The proposed strategy is also found to be suitable for other topology optimization methods. In the following chapter, a complete GPU acceleration of the BESO method pipeline is discussed. Furthermore, BESO is also coupled with the mesh reduction strategy developed in this chapter.

Chapter 5

GPU Acceleration of Large-Scale BESO Method

For achieving the third objective, a GPU-accelerated implementation for large-scale 2D BESO method is presented. This work addresses the primary challenge of high computational complexity in performing large-scale topology optimization through a complete GPU acceleration of the entire BESO method pipeline. The second major challenge of prohibitively high memory consumption is handled by implementing a matrix-free PCG solver in the finite element analysis stage. An element-by-element strategy has been adopted to parallelize the FEA, sensitivity and compliance calculation, mesh filter, and design update stages. Among all the optimization steps, FEA is observed to consume up to 92% of the total execution time. Consequently, the focus of this work is kept on an efficient parallelization of FEA using the PCG method with Jacobi preconditioner. It is observed that the PCG algorithm consists of different linear algebra operations, which contribute to all the computations in the algorithm.

- Sparse matrix-vector multiplication (SpMV)
- Inner product (IP)
- A X plus Y (SAXPY)

These individual building blocks of the PCG algorithm are accelerated using suitable strategies that combine the use of the thrust library and custom GPU kernels. Apart from PCG, the other stages of BESO including sensitivity calculation, compliance calculation, mesh filter, stabilization, and design update are also accelerated on GPU using a combination of thrust calls and custom kernels.

Utilizing the acceleration of the BESO method on GPU, the mesh reduction strategy is implemented to develop a novel GPU-based hybrid BESO method that combines the soft-kill and hard-kill strategies.

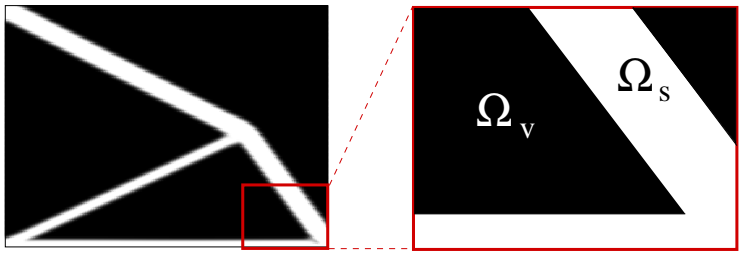


Figure 5.1: The solid (Ω_s) and void (Ω_v) subdomains for an example geometry.

The hard-kill formulation uses a solid/void design with discrete values of the densities for the structures. The void elements are removed from the model altogether at every optimization iteration. The soft-kill formulation, on the other hand, uses very small values for the densities of void elements with a material interpolation scheme. This solves some of the computational issues with hard-kill at the expanse of increased computation. The proposed hybrid BESO uses the hard-kill approach for the FEA stage and soft-kill approach for all other stages to eliminate the redundant computational cost incurred due to performing FEA for non-functional degrees of freedom on GPU. This allows the key advantage of the hard-kill approach to be incorporated into the soft-kill without the latter inheriting any of its several drawbacks, such as the breakage of boundary support and other convergence issues. A numbering scheme similar to the one used in Chapter 4 is implemented for on-the-fly calculation of active DOFs on GPU.

In the following sections, details of hybrid BESO, which combines a complete GPU acceleration of BESO method with the mesh reduction strategy from Chapter 4, is discussed.

5.1 GPU-adapted Hybrid BESO

The proposed GPU-adapted Hybrid BESO utilizes both soft-kill and hard-kill approaches. A soft-kill approach with material interpolation scheme is used for the entirety of optimization procedure and a hard-kill approach is adopted for the FEA stage. In order to implement hard-kill approach, on-the-fly calculation of active DOFs is required at every optimization iteration before performing FEA. This is achieved by separating the design domain Ω into Ω_s and Ω_v for the solid and the void parts respectively, as shown in Figure 5.1. Thereafter, the hybrid scheme is implemented by using a soft-kill approach throughout and assuming an infinite penalty for the FEA stage to force a completely solid/void or 1/0 structure. By using infinite value of the penalty parameter, p, equation (2.17) and equation (2.18) from Chapter 2 can be reduced to,

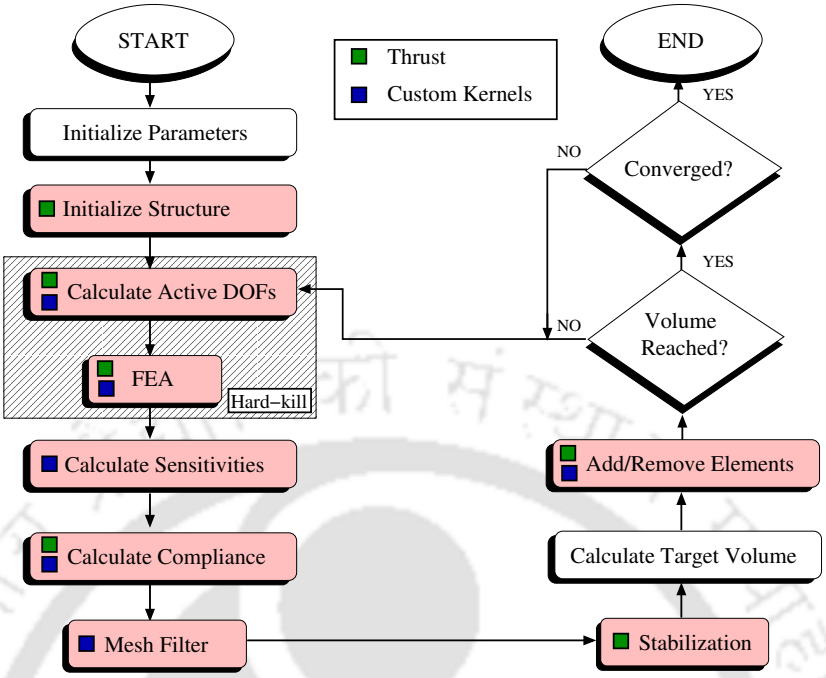


Figure 5.2: Flow chart of GPU-accelerated hybrid BESO.

$$E(\rho_i) = \begin{cases} E_0 & \text{for} \quad i \in \Omega_s, \\ 0 & \text{for} \quad i \in \Omega_v, \end{cases} \tag{5.1}$$

$$[K] = \sum_{i=1}^{n_s} [K]_i, \quad i \in \Omega_s, \tag{5.2}$$

where n_s is the number of solid elements in the mesh at that iteration. The present implementation targets complete parallelization of the BESO procedure on the GPU. This is achieved using a combination of CUDA Thrust library functions and custom CUDA kernels. The flow chart of the hybrid BESO shown in Figure 5.2 marks the GPU-accelerated parts in pink. The figure also shows the parallelization strategy for each individual stage by depicting steps accelerated using Thrust, steps accelerated using custom kernels, and steps that make use of both. In the following sections, discussion about different parts of the GPU-based hybrid BESO is presented.

5.1.1 Mesh numbering and on-the-fly DOF calculation

For performing FEA using only the solid elements, a list of functional DOFs needs to be prepared. For this purpose, we propose a parallel strategy that takes the mesh connectivity information and the vector of design variables ρ_i to output the list of functional DOFs. Removing DOFs from the structure comes with two key issues.

1. **Removal of DOFs with boundary conditions:** DOFs containing boundary conditions can become inactive and thereby, removed from the structure. This is a known issue in BESO that hinders convergence. Huang and Xie (2008) suggested checking for such breakage of boundary support at every iteration to alleviate the problem. However, this strategy becomes prohibitively expensive for large-scale structures on the GPU.

2. **Locating DOFs with boundary condition from the reduced list:** Another issues is locating DOFs containing boundary conditions on the GPU from the list of functional DOFs to implement the Dirichlet and Neumann boundary conditions. For example, if a load is applied at the n^{th} DOF, the n^{th} entry in the global load vector needs to be modified. However, if, instead of the entire global load vector, a shortened vector is used with only the functional DOFs, there is no easy way to determine the location of the boundary node n from the shortened load vector stored in GPU global memory. This is not a problem for CPU implementations due to the sequential nature of the computation, whereas for a GPU implementation this creates a major issue. Although a search operation can be implemented to look for the target nodes at every iteration, such operations are considered highly unsuited to the GPU architecture for causing massive thread divergence and thus, prove highly detrimental to the performance of the application.

It is worth noting here that both the issues discussed above are only relevant to parallel GPU-based implementations of large-scale problems. For small-scale problems and for sequential CPU implementations, these are not causes of major concern. In order to circumvent these issues, a new numbering scheme is used for the finite element mesh. The key idea behind the numbering scheme is to number the DOFs with boundary conditions at the beginning and then proceed to number the rest of the DOFs in the mesh. The scheme, illustrated in Figure 5.3 with the help of a cantilever beam and a L-beam problem, is similar to the mesh reduction strategy discussed in Chapter 4. As illustrated in the figure, the Neumann and Dirichlet boundary DOFs are numbered first, followed by the rest of the mesh. The total number of boundary condition DOFs (n_{BC}) is passed to the kernels and during calculation of functional DOFs, the DOFs containing boundary conditions are not considered. Since they are always numbered $1 - n_{BC}$, for all problems and all boundary conditions, there is no need to locate the boundary DOFs and issue (2) is resolved immediately. Furthermore, since boundary DOFs ($1 - n_{BC}$) cannot be removed from the vector of functional DOFs [$fDOF$], there is no risk of removing boundary support and issue (1) becomes non-existent. Using the numbering scheme presented in Figure 5.3, [$fDOF$] is calculated according to the steps presented in Algo. 12. As shown in Figure 5.2, this step is parallelized with a combination of thrust calls and a custom CUDA kernel. At the start of Algo. 12, the vector [$fDOF$] is initialized using thrust and a raw pointer is extracted for use in the kernel. Following this, [$fDOF$] is filled with DOF numbers starting from 1 to the number of total DOFs in the mesh, n_{DOF}. In line 4, CUDA kernel `calcfDOFkernel` is launched that checks the ρ values of neighbor elements for every

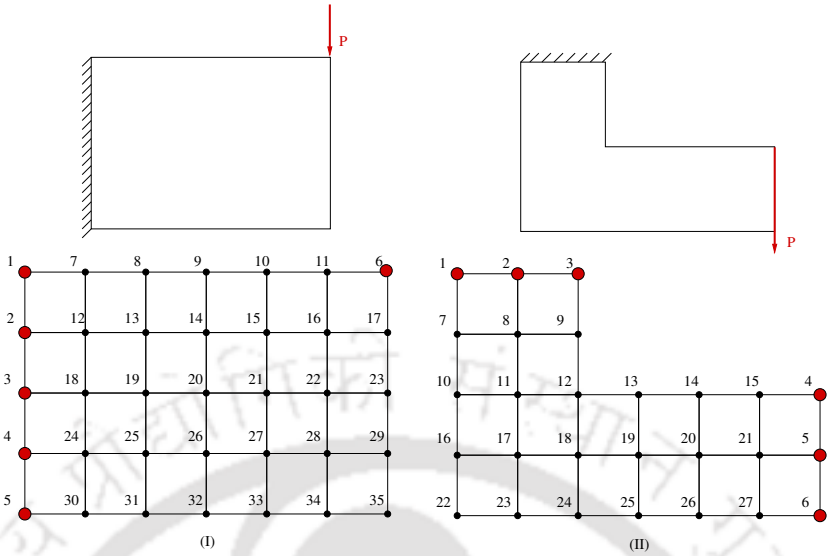


Figure 5.3: Numbering scheme for hybrid BESO.

DOF. The non-functional DOF numbers are replaced with an arbitrary negative number in $[fDOF]$. A DOF is deemed non-functional if it does not have at least one solid element in its neighborhood. Following the kernel call, $thrust :: count\_if$ is used to count the positive numbers in $[fDOF]$, which gives the number of functional DOFs in the mesh (n_{fDOF}). Finally, the negative numbers are removed from $[fDOF]$ using $thrust :: remove\_if$ in line 6 to obtain the final list of functional DOFs for the optimization iteration. Steps performed in `calcfDOFkernel` are shown in Algo. 13. At the beginning, a

Algorithm 12: Calculating functional DOFs

Input : $[\rho]$
Output: $[fDOF]$, n_{fDOF}
1 Initialize vector $[fDOF]$ using Thrust;
2 Extract raw pointer for use in kernel;
3 Fill $[fDOF]$ using $thrust :: sequence$;
4 `calcfDOFkernel`();
5 $n_{fDOF} = thrust :: count\_if(fDOF.begin(), fDOF.end(), is\_positive())$;
6 $thrust :: remove\_if(fDOF.begin(), fDOF.end(), is\_negative())$;

unique thread id tx is set for every GPU thread. Line 2 of Algo. 13 indicates that $1-n_{BC}$ DOF numbers are exempted from functionality checking of DOFs and thus, cannot be removed from $[fDOF]$. For the cantilever example in Figure 5.3, n_{DOF} is 35 and n_{BC} is 6. Therefore, only DOFs $7-35$ go through the functionality check. Similarly for the L-beam example, only DOFs $7-27$ go through the functionality check. Each thread performs the computation in parallel where the sum of densities of all neighboring elements is checked. A sum value less than 1 indicates a non-functional DOF. The corresponding DOF number is updated in $[fDOF]$ to an arbitrary negative value as shown in lines $7-9$. In the next section, GPU implementation for hard-kill FEA using $[fDOF]$ array is discussed.

Algorithm 13: calcfDOFkernel

Input : $[\rho]$, n_{DOF}, $[fDOF]$, n_{BC}
Output: Modified $[fDOF]$

```
1  tx ← threadIdx.x + blockIdx.x × blockDim.x; // Assign a unique thread ID to every
     GPU thread
2  for thread ID tx ← n_BC + 1 to n_DOF do
     // on GPU
3    sum ← 0;
4    for i ← 1 to n_ne do                                   // n_ne is the number
       of neighbors
5      sum ← sum + ρ[i];                                    // Sum of ρ for all neighbors
6    end
7    if sum < 1 then
8      fDOF[tx] ← −1;                                       // Arbitrary negative value
9    end
10 end
```

5.1.2 FEA using hard-kill approach

FEA is the most time consuming step in the entire optimization process, which was also observed in the last chapters. The most time-consuming part of the FEA step is the solution of linear system of equations (Georgescu et al., 2013). A Jacobi-preconditioned conjugate gradient (JPCG) method (Shewchuk, 1994) is used for that purpose in this work. As a standard practice for FEA of large-scale structures on GPU, assembly-free method is used for the linear solver. Details of the assembly-free JPCG is provided in Algo. 14. It can be seen that the entire Algo. 14 is made up of a series of linear algebra operations. The three key linear algebra operations performed at this stage are assembly-free matrix-vector product (afMVp), vector-vector product (VVp), and linear transformation of vectors (Vlt). Among these three, the afMVp operations are performed using a custom CUDA kernel `afMVpKernel` in line 2 and line 7 of Algo. 14. The rest of the linear algebra operations are performed using thrust calls. It is noted that all the computation in Algo. 14 is performed only for DOFs belonging to Ω_s, stored in $[fDOF]$. The details of the `afMVpKernel` are presented in Algo. 15. In order to locate the DOFs of the shortened structure in connectivity matrix, C_M, a $[map]$ array is used, where $map[fDOF[i]] = i$, for $i \leftarrow 1$ to n_{DOF}.

Algo. 15 takes the vector to be multiplied $[U]$, elemental stiffness matrix $[K_e]$, $[fDOF]$, $[map]$, $[C_M]$, and n_{fDOF} as input and outputs $[\tilde{U}]$, the result vector. In line 1, a unique thread id is assigned to each DOF followed by storing $[K_e]$ into shared memory and a barrier synchronization in lines 2 and 3. Thereafter, each thread performs its assigned multiplication in parallel as shown in line 4. In line 5, the DOF number is extracted for each thread from $[fDOF]$. Then, in line 7, a for-loop is run for all neighboring elements of DOF, d. Inside this loop, in line 8 and line 9, the element number and the corresponding local DOF number are extracted for DOF, d. Another for-loop is then run in line 10 for all the DOFs of element e. The $[map]$ array is used for extracting target index of multiplication using

Algorithm 14: assembly-free preconditioned conjugate gradient

Input : F, K, J, ϵ
Output: u

$i \leftarrow 0$; // #iteration counter
$r \leftarrow F -$ afMVpKernel(); // Assembly-free matrix-vector product, multiplying K, u
$d \leftarrow J^{-1}r$; // Vector-vector product
$\delta_{new} \leftarrow r^T d$; // Vector-vector product
$\delta_0 \leftarrow \delta_{new}$; // Scalar operation
while $i \leq i_{max} \wedge \delta_{new} > \epsilon^2 \delta_0$ **do**
  $q \leftarrow$afMVpKernel(); // Assembly-free matrix-vector product, multiplying K, d
  $\alpha \leftarrow \frac{\delta_{new}}{d^T q}$; // Vector-vector product
  $u \leftarrow u + \alpha d$; // Vector linear transformation
  $r \leftarrow r - \alpha q$; // Vector linear transformation
  $v \leftarrow J^{-1}r$; // Vector-vector product
  $\delta_{old} \leftarrow \delta_{new}$; // Scalar operation
  $\delta_{new} \leftarrow r^T v$; // Vector-vector product
  $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$; // Scalar operation
  $d \leftarrow v + \beta d$; // Vector linear transformation
  $i \leftarrow i + 1$; // Scalar operation
end

C_M. The multiplication is performed in line 12, results are added up for all neighboring elements and finally the result is stored in $[\tilde{U}]$ at line 15. At the end of each FEA, the shortened displacement vector

Algorithm 15: afMVpKernel

Input : $[U]$, $[K_e]$, $[fDOF]$, $[map]$, C_M, n_{fDOF}
Output: $[\tilde{U}]$

$tx \leftarrow threadIdx.x + blockIdx.x \times blockDim.x$; // Assign a unique thread ID to every GPU thread
Store $[K_e]$ into shared memory for each thread block;
Thread synchronization;
for thread ID $tx \leftarrow 1$ **to** n_{fDOF} **do** // on GPU
  $d \leftarrow fDOF[tx]$;
  $res \leftarrow 0$; // Temporary variable res for accumulating products
  for $i \leftarrow 1$ **to** $n_{ne}(d)$ **do** // For all neighbors of d
    $e \leftarrow$ Element Number;
    $l \leftarrow$ Local DOF number;
    for $j \leftarrow 1$ **to** 8 **do** // For 8 local DOFs
      $t \leftarrow map[C_M[8 \times e + j]]$;
      $res \mathrel{+}= U[t] \times K_e[l][j]$;
    end
  end
  $\tilde{U}[tx] = res$; // Write to global memory
end

is expanded to include all DOFs by performing a *thrust* :: *scatter* operation with [*map*] array for use in the rest of soft-kill BESO.

5.1.3 Compliance and sensitivity computation

As shown in Figure 5.2, the sensitivity computation stage is performed by using custom kernels, whereas the structural compliance is computed using a combination of thrust call and custom kernel. In the present implementation both the elemental stiffness and elemental compliance are calculated using one single kernel. This is followed by a call of *thrust* :: *reduce* that sums all the individual compliance to give the total compliance of the structure. Details of the `compSensKernel` for computing the individual compliances and sensitivities are given in Algo. 16

Algorithm 16: compSensKernel

Input : $[\rho]$, $[\bar{U}]$, $[C_M]$, $[Ke]$, n_{EL}, p
Output: $[comp]$, $[\alpha]$

1 $tx \leftarrow threadIdx.x + blockIdx.x \times blockDim.x$; `// Assign a unique thread ID to every GPU thread`
2 **for thread ID** $tx \leftarrow 1$ **to** n_{EL} **do**
`// on GPU`
3 $temp \leftarrow [\bar{U}e]^T[K_e][\bar{U}e]$; `// __device__ function`
4 $\alpha[tx] \leftarrow 0.5 \times pow(\rho[tx], p - 1) \times temp$; `// sensitivity`
5 $comp[tx] \leftarrow 0.5 \times pow(\rho[tx], p) \times temp$; `// compliance`
6 **end**

`compSensKernel`, as shown in Algo. 16 is launched with element-level granularity in which one thread is assigned to one element. This kernel takes the final resulting displacements $[\bar{U}]$, $[\rho]$, $[C_M]$, $[K_e]$, number of elements in the mesh n_{EL}, and p as inputs and outputs the array of compliances $[comp]$ and array of sensitivities $[\alpha]$. A unique thread id is calculated and assigned to each element of the finite element mesh in lines 1 and 2. At line 3, the multiplication of $[\bar{U}e]^T[K_e][\bar{U}e]$ is performed at the elemental level using a device function. This multiplication is required for the calculation of both the elemental sensitivities and elemental compliance at lines 4 and 5.

5.1.4 Mesh filter and stabilization

The mesh-independency filter is applied to the elemental sensitivities $[\alpha]$ to obtain the filtered sensitivities $[\alpha_f]$. This stage is parallelized on the GPU using a custom kernel as shown in Figure 5.2. The details of the kernel `filterKernel` is given in Algo. 17. This kernel takes $[\rho]$, $[\alpha]$, and filter radius r_f as input and outputs the vector of filtered sensitivity $[\alpha_f]$. `filterKernel` is launched with element level granularity, where each thread is responsible for one element of the mesh. Subdomain Ω^e_{ne} containing the neighbor elements are constructed for each element e for applying the filter. A for-loop is used to

loop over all the elements in Ω^e_{ne} as shown in line 5. The weights are calculated using device function *dist*() that returns the distance between two elements in line 6. The weighted sums are calculated in line 8 and finally divided by the total weight in line 10 to give the filtered density vector $[\alpha_f]$.

Algorithm 17: filterKernel

Input : $[\rho]$, $[\alpha]$, r_f
Output: $[\alpha_f]$

1 $tx \leftarrow threadIdx.x + blockIdx.x \times blockDim.x$; `// Assign a unique thread ID to every GPU thread`
2 **for thread ID** $tx \leftarrow 1$ **to** n_{EL} **do**
`// on GPU`
3 $temp[tx] = 0$;
4 $sum = 0$;
5 **for all** $e \in \Omega^{tx}_{ne}$ **do** `// ` Ω^{tx}_{ne} `: Subdomain consisting neighbors of element no.` tx
6 $w \leftarrow r_f - dist(tx, e)$; `// ` $dist(a,b)$ `: Distance between element a and element b`
7 $sum+ = max(0, w)$;
8 $temp[tx]+ = \alpha[e] \times max(0, w)$;
9 **end**
10 $\alpha_f[tx] = temp[tx]/sum$;
11 **end**

Following the calculation of $[\alpha_f]$, sensitivity stabilization is applied with the help of a *thrust* :: *transform* call.

5.1.5 Updating structures

Following the mesh filter and sensitivity stabilization, the structure is updated by using a combination of thrust calls and CUDA kernel. The update scheme proposed by Huang and Xie (2010a) is used for this purpose. The details are presented in Algo. 18. This algorithm takes $[\alpha_f]$, $[\rho]$, evolutionary volume ratio (ER), volume fraction (VF), and number of elements (n_{EL}) to output the updated design variables $[\rho_{new}]$. At line 1, the target volume V_t for the optimization iteration is determined. This is followed by thrust calls to *thrust* :: *min_element* and *thrust* :: *max_element* to initialize limits $l1$ and $l2$. This is followed by a while-loop for updating the variables till the pre-set tolerance value is reached. Kernel `newDensKernel` is launched to obtain the updated design variables $[\rho_{new}]$. Finally, the volume of the structure is checked using *thrust* :: *reduce* and the limits are updated accordingly in lines $8-12$. Details of `newDensKernel` are given in Algo. 19. This kernel takes $[\rho]$, $[\alpha_f]$, r_f, α_{th}, n_{EL}, and ρ_{min} as inputs and outputs the updated design variables $[\rho_{new}]$. In this kernel each thread is assigned to updating the ρ value of one element. This is done by checking the difference between α_{th} and filtered sensitivity of the element. If this difference is more than zero, ρ_{new} is assigned a value of 1 , as shown in line 7. If the difference is less than or equal to zero, ρ_{min} is assigned to ρ_{new}.

Algorithm 18: Updating structures

Input : $[\alpha_f]$, $[\rho]$, ER, VF, n_{EL}
Output: $[\rho_{new}]$

$V_t \leftarrow max(V_t \times (1 - ER), VF) \times n_{EL}$;
$l1 \leftarrow thrust :: min\_element([\alpha_f])$;
$l2 \leftarrow thrust :: max\_element([\alpha_f])$;
while $(l2 - l1)/l2 > 1.0e - 5$ **do**
    $\alpha_{th} \leftarrow (l1 + l2)/2.0$;
    $[\rho_{new}] \leftarrow$`newDensKernel()`;
    $sum \leftarrow thrust :: reduce([\rho_{new}])$;
    if $sum - V_t > 0$ **then**
        $l1 \leftarrow \alpha_{th}$;
    else
        $l2 \leftarrow \alpha_{th}$;
    end
end

Algorithm 19: newDensKernel

Input : $[\rho]$, $[\alpha_f]$, r_f, α_{th}, ρ_{min}, n_{EL}
Output: $[\rho_{new}]$

$tx \leftarrow threadIdx.x + blockIdx.x \times blockDim.x$; `// Assign a unique thread ID to every GPU thread`
for thread ID $tx \leftarrow 1$ **to** n_{EL} **do**
    `// on GPU`
    $check \leftarrow \alpha_f[tx] - \alpha_{th}$;
    if $check == 0$ **then**
        $\rho_{new}[tx] = \rho_{min}$;
    else if $check > 0$ **then**
        $\rho_{new}[tx] = 1$;
    else
        $\rho_{new}[tx] = \rho_{min}$;
    end
end

5.1.6 Penalty parameter values

Typically in the literature, for standard BESO implementations, the penalty parameter is set to 3. Accordingly, for the soft-kill BESO implementation, the value of p is set to 3. However, it is observed that setting $p = 3$ for hybrid BESO leads to the algorithm not converging to the optimal topology in most of the cases. This is due to the fact that during the hard-kill implementation for FEA, the displacements for all non-functional DOFs are set to zero. This includes DOFs that are immediately adjacent to the functional DOFs. These non-functional DOFs adjacent to functional DOFs would otherwise have non-zero displacements that are essentially truncated to zero in the hybrid BESO. However, since for the rest of the algorithm, a soft-kill approach is implemented, setting the displacements to zero for these DOFs leads to error in the sensitivity calculation and filter stage. As a remedy, a high penalty parameter is used ($p = 10$) for the hybrid approach that takes care of the convergence issue. Using such a high

Table 5.1: Number of nodes for the test meshes of four benchmark problems. Meshes 1, 2, 3, 4, 5, and 6 contain approximately 100k, 300k, 500k, 700k, 900k, and 1.1 million nodes for all problems.

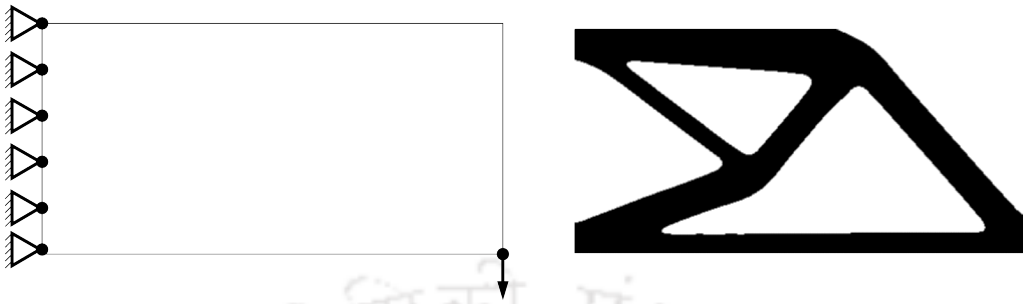
	CBEL	MBBB	OLM	LSB	
Mesh 1	100352	100467	99856	100647	$\approx$ 100k
Mesh 2	299538	299568	300304	301675	$\approx$ 300k
Mesh 3	500000	499392	499849	497377	$\approx$ 500k
Mesh 4	700928	699867	700569	700987	$\approx$ 700k
Mesh 5	900482	900912	900601	901075	$\approx$ 900k
Mesh 6	1101128	1101708	1100401	1100925	$\approx$ 1.1m

value of penalty in soft-kill BESO is generally avoided due to the chance of certain entries of the global stiffness matrix becoming too small during the FEA stage, leading to singularity in the matrix. The hybrid approach allows for such high value due to the altogether removal of the penalty parameter from the FEA formulation.

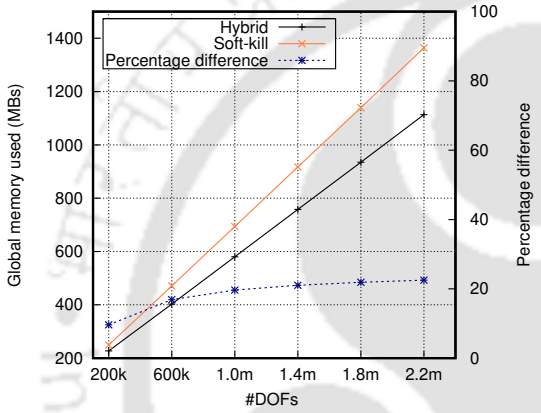
5.2 Results and Discussion

In this section, the effectiveness of GPU-based hybrid BESO is examined by comparing it with a GPU implementation of soft-kill BESO. The soft-kill implementation used follows the formulation by Huang and Xie (2010b). Since a standard GPU-based soft-kill implementation is not available in the literature, the soft-kill formulation is implemented with the accelerated BESO pipeline to ensure a fair comparison. The implemented soft-kill approach differs from the proposed hybrid BESO in the FEA stage, mesh numbering, and computation of functional DOFs. The performance analysis of the proposed method is performed on a workstation with Intel Xeon ES1650 (6 core, 3.2 GHz) processor, 12 GB RAM, and a K40c NVIDIA GPU that has 12 GB GDDR5 global memory. The GPU has peak single precision floating point performance of 4.29 Tflops, peak double precision floating point performance of 1.43 Tflops, and a peak bandwidth of 288 GB/s. It comes with a total of 2880 CUDA cores that are split into 15 streaming multiprocessors. Four problems are chosen for the performance analysis from a list of 2D benchmarks provided by Valdez et al. (2017): Cantilever beam with end load (CBEL), MBB beam (MBBB), one-load Michell (OLM), and L-shape beam (LSB). In order to ensure a uniform comparison, mesh sizes are chosen for each problem containing approximately the same number of DOFs. Details of the meshes used for analysis are given in Table 5.1.

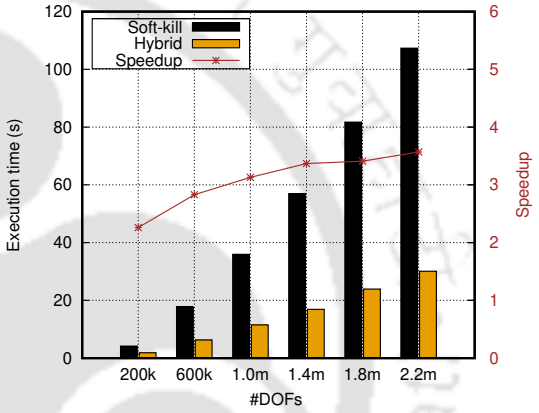
The boundary conditions and final optimized topology for CBEL problem are given in Figure 5.4(a). The topology is found to be identical with the ones reported in the literature (Valdez et al., 2017). Figure 5.4(b) shows the variation of global memory consumption for the soft-kill and the hybrid BESO with increasing DOFs in mesh. The percentage difference in global memory consumed is shown using the secondary Y-axis. In both soft-kill and hybrid BESO, linearly increasing trends are observed. Furthermore, it can be seen that the soft-kill BESO consumes up to 20% more memeory compared to



(a) Boundary condition and optimized topology



(b) Global memory

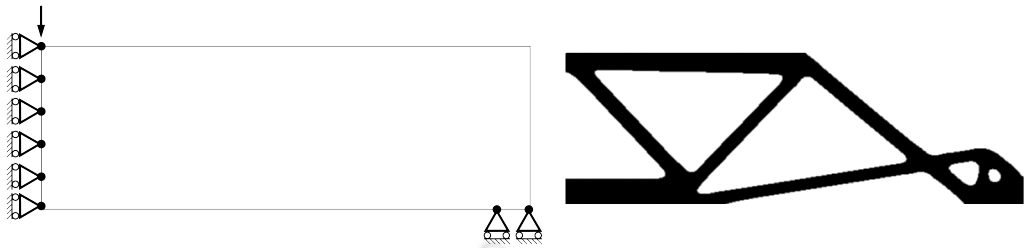


(c) Execution time and speedup

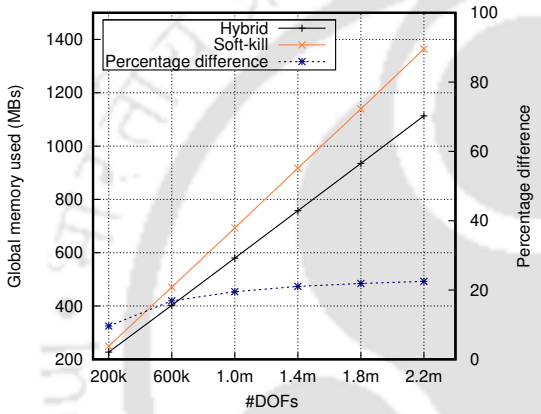
Figure 5.4: Results of CBEL problem. In (a) the boundary condition is shown with the optimized topology using hybrid BESO. Plot (b) shows the variation of GPU global memory used using soft-kill and hybrid approaches. In (c) the individual execution times are shown along with the speedup for hybrid BESO.

the hybrid approach for higher mesh sizes. Figure 5.4(c) shows the execution time per iteration for soft-kill and hybrid BESO using clustered bars. It is observed that in all mesh sizes, soft-kill BESO consumes more time than hybrid BESO. This difference is more pronounced for higher mesh sizes. This is reflected by the speedup line in the same plot that shows the ratio of the execution time of soft-kill BESO to the execution time of hybrid BESO. The values of speedup increase with increasing mesh size and approximately 3.5× speedup is achieved for the highest mesh size.

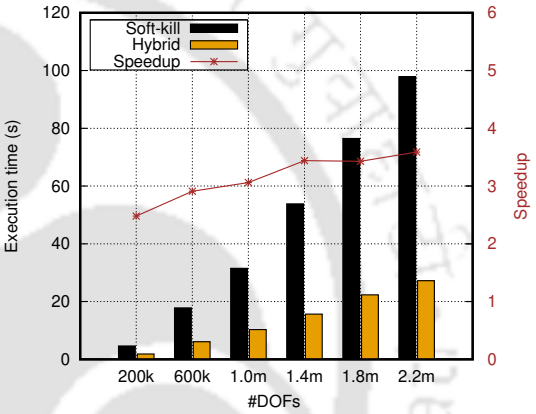
Figure 5.5(a) shows the boundary condition of the MBBB problem along with its optimized topology using hybrid BESO. The topology is found to be identical with the ones reported in the literature (Valdez et al., 2017). Figure 5.5(b) shows the global memory consumed by soft-kill and hybrid BESO for different mesh sizes. The percentage difference in memory consumed is also shown in the same plot with a blue line. For the highest mesh size having 2.2 million DOF, approximately 20% difference in memory values



(a) Boundary condition and optimized topology



(b) Global memory

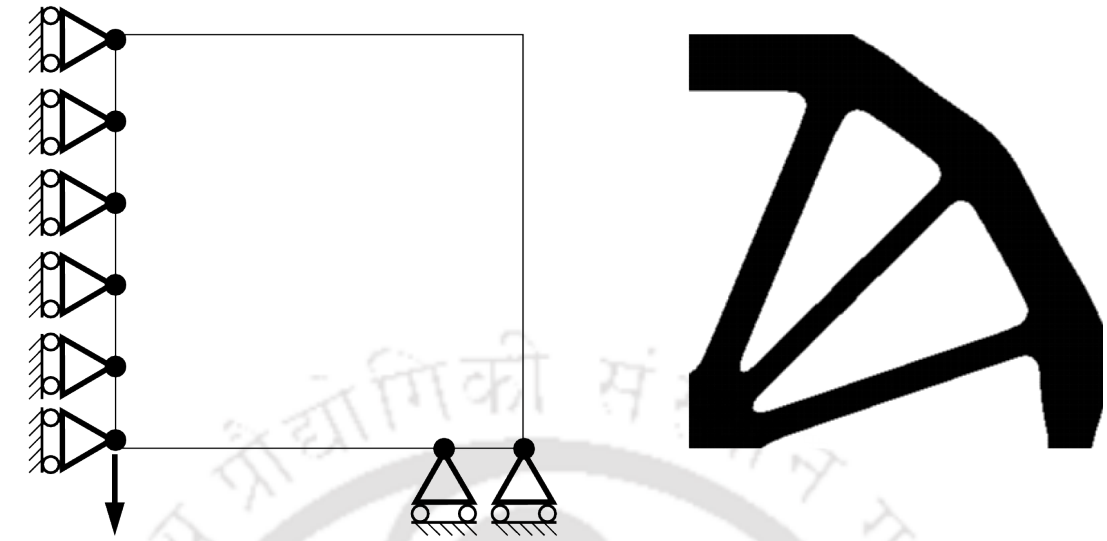


(c) Execution time and speedup

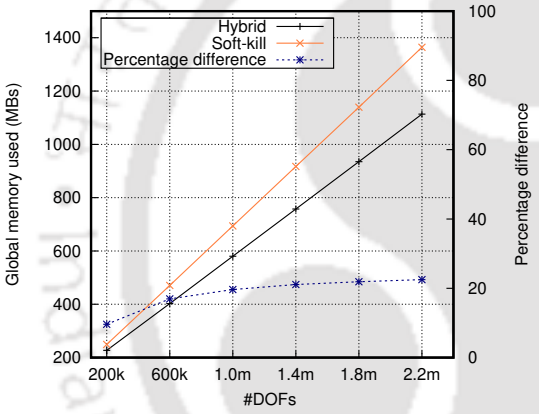
Figure 5.5: Results of MBBB problem. In (a) the boundary condition is shown with the optimized topology using hybrid BESO. Plot (b) shows the variation of GPU global memory used using soft-kill and hybrid approaches. In (c) the individual execution times are shown along with the speedup for hybrid BESO.

is observed. Figure 5.5(c) shows the execution times for soft-kill and hybrid BESO along with the speedups obtained for the hybrid BESO over soft-kill. It is observed that hybrid BESO outperforms soft-kill BESO in terms of execution time for all mesh sizes wielding speedups of approximately $2\times - 3.5\times$ for different mesh sizes. Furthermore, comparing with Figure 5.4(c), it can be seen that for both soft-kill and hybrid BESO, the execution times are higher in case of CBEL problem compared to MBBB problem for the same mesh sizes. The speedup values, however, are found to be similar in both cases.

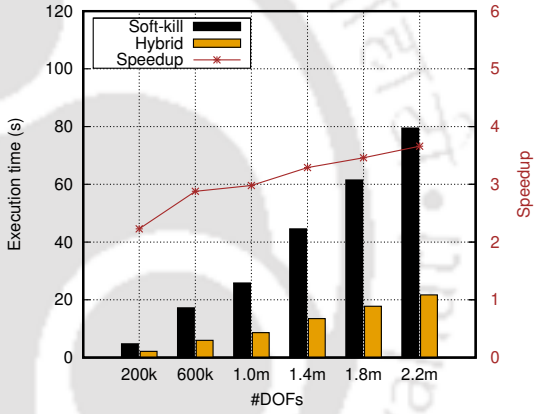
The OLM problem domain with boundary conditions and the final optimized topology are shown in Figure 5.6(a). The optimized topology is found to be identical with the ones reported in the literature (Valdez et al., 2017). The global memory consumed for the OLM problem using soft-kill and hybrid BESO is shown in Figure 5.6(b). By comparing memory consumed and percentage memory difference for difference mesh sizes, it is observed that for higher mesh sizes of OLM problem, soft-kill BESO consumes



(a) Boundary condition and optimized topology



(b) Global memory



(c) Execution time and speedup

Figure 5.6: Results of OLM problem. In (a) the boundary condition is shown with the optimized topology using hybrid BESO. Plot (b) shows the variation of GPU global memory used using soft-kill and hybrid approaches. In (c) the individual execution times are shown along with the speedup for hybrid BESO.

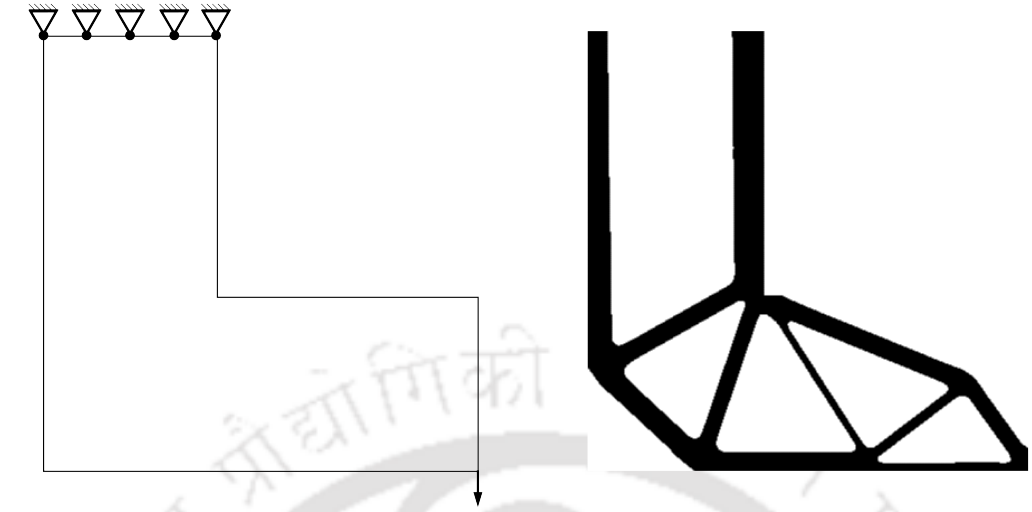
approximately 20% more memory than hybrid BESO. Furthermore, comparing Figures 5.4(b) and 5.5(b) with Figure 5.6(b), it can be seen that the global memory consumption as well as the percentage memory difference is almost identical for all three CBEL, MBBB and OLM problems. Figure 5.6(c) shows the variation of execution times for soft-kill BESO and hybrid BESO with the corresponding speedup values. For higher mesh sizes, speedups close to 4× are achieved. By comparing the same plots from CBEL and MBBB problem, it can be seen that for the OLM problem, both soft-kill BESO and hybrid BESO take significantly less amount of time.

Figure 5.7 shows the boundary condition, optimized topology and performance of the LSB problem using soft-kill and hybrid BESO. The topology is found to be similar to the ones reported in the

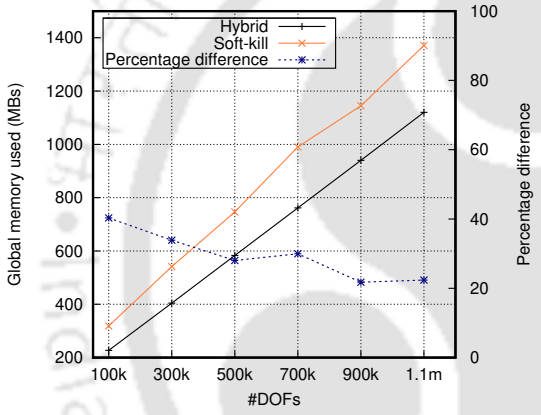
literature (Valdez et al., 2017). The amount of global memory consumed for the test mesh sizes using both approaches is presented in Figure 5.7(a). It can be observed that for the same mesh sizes, LSB problem consumes slightly more global memory compared to the other three problems. Moreover, unlike the other three problems, the percentage difference in global memory exhibits a downward trend with higher mesh sizes. For lower mesh sizes, soft-kill BESO consumes approximately 40% more memory than hybrid BESO, whereas for higher mesh sizes, this value drops to about 20%. Figure 5.7(c) shows the execution times and speedup values for the two approaches. The speedup values are calculated for the comparatively faster hybrid BESO over the soft-kill BESO. Comparison of Figure 5.7(c) with Figures 5.4(c), 5.5(c), and 5.6(c) reveals that similar to memory consumed, LSB problem consumes the most amount of execution time among all problems for both soft-kill and hybrid BESO. Furthermore, the speedup values are observed to be slightly lower compared to other problems. Also, unlike other three problems, no increasing trend is observed for the speedup values of LSB problem using higher mesh sizes.

Both soft-kill and hybrid BESO implementations in this work start from a completely filled structure. This essentially means that for both the implementations, the entire vector of design variables $[\rho]$ is initialized to 1. The number of functional DOFs (n_{fDOF}) stays the same for soft-kill, but keeps on decreasing iteration-wise for the hybrid approach. The extent of this decrease depends on the target volume fraction, whereas the rate of decrease is dependant on the element removal rate of BESO. The final percentage reduction of functional DOFs at the end of the optimization process is plotted in Figure 5.8 for all four problems. It is observed that similar level of reduction is achieved for all problems having similar mesh sizes. Furthermore, with higher mesh sizes, the amount of reduction in functional DOFs becomes more pronounced in all cases. Figures 5.4(c), 5.5(c), 5.6(c), and 5.7(c) provide insights on the execution times of soft-kill and hybrid BESO for the corresponding problems. The three key parts in BESO are FEA, sensitivity filter, and structure updation. Due to the disparity in the orders of magnitudes among these stages' execution time, the data is presented in Table 5.2. A single mesh size containing approximately 2.2 million DOFs is taken for all four benchmark problems to ensure uniform comparison. From the table, it can be observed that FEA is by far the most computationally expensive stage in the entire process consuming more than 99% of the total execution times. This observation is in line with several studies in the literature targeting GPU acceleration of topology optimization methods (Ram and Sharma, 2017, Ramírez-Gil et al., 2016, Martínez-Frutos et al., 2017). It is also observed that design update is the least computationally expensive part of the process. Furthermore, from the execution times, it can be seen that the LSB problem consumes the most time among all four problems for both soft-kill and hybrid BESO.

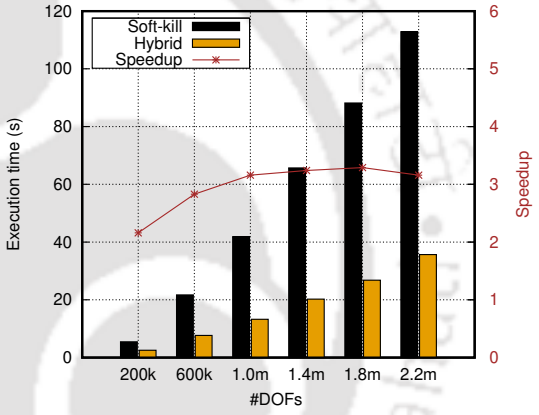
Table 5.2 makes it apparent that the most important part of the entire BESO method pipeline is the FEA stage. In order to obtain a better understanding and to have a deeper analysis into the FEA stage, three key parts are isolated from the assembly-free PCG: assembly-free matrix-vector product (afMVp),



(a) Boundary condition and optimized topology



(b) Global memory



(c) Execution time and speedup

Figure 5.7: Results of LSB problem. In (a) the boundary condition is shown with the optimized topology using hybrid BESO. Plot (b) shows the variation of GPU global memory used using soft-kill and hybrid approaches. In (c) the individual execution times are shown along with the speedup for hybrid BESO.

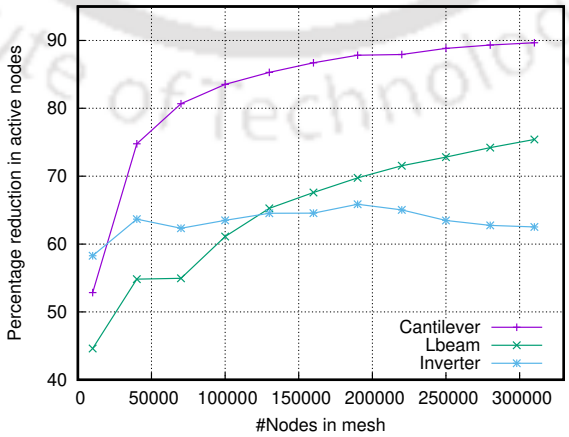


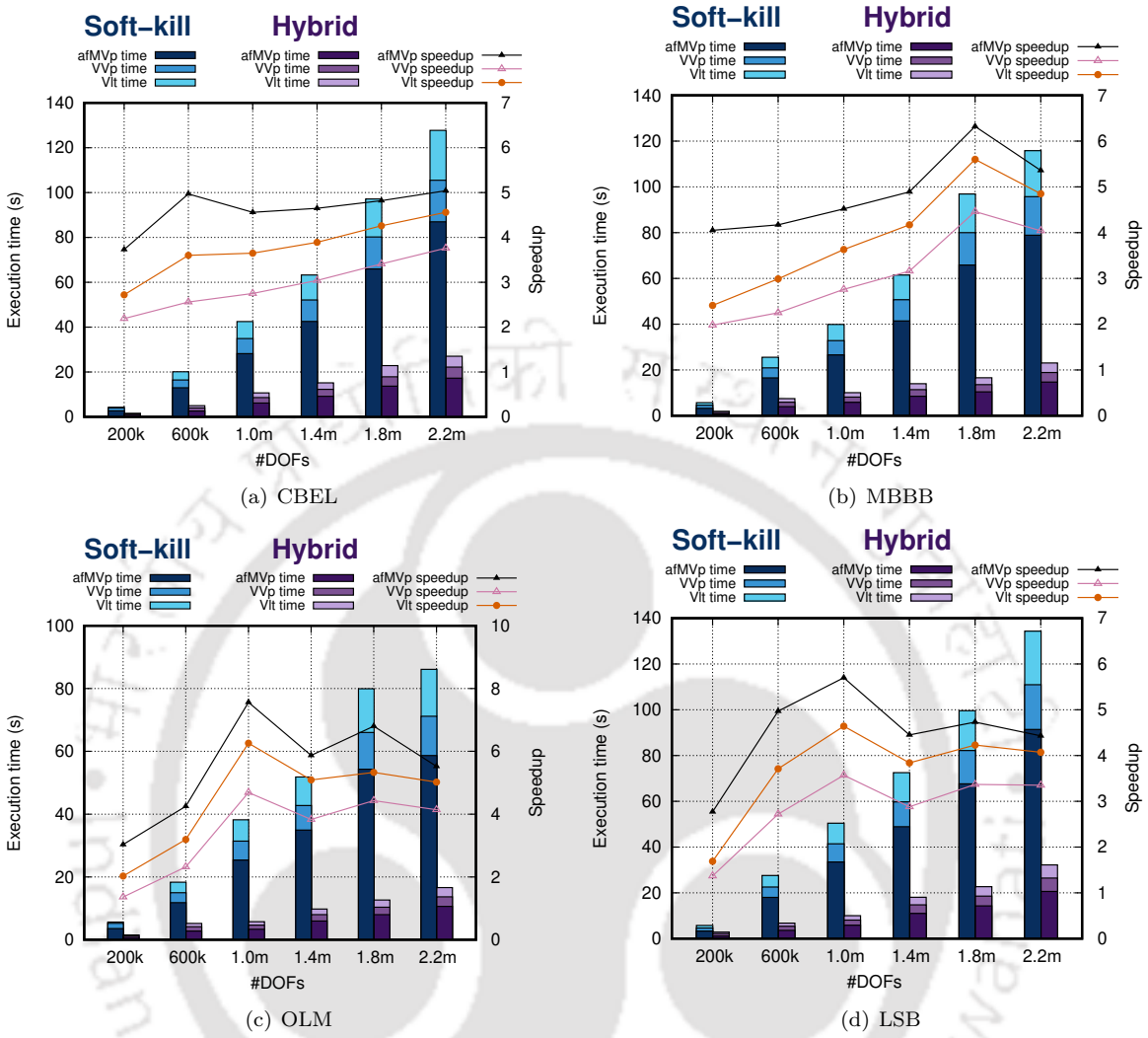
Figure 5.8: Percentage reduction in functional DOFs with increasing mesh size.

Table 5.2: Execution times (second) of FEA, sensitivity filter, and structure update for four benchmark problems using hybrid BESO and soft-kill BESO are listed. For all four problems, mesh sizes are taken that contain approximately 2.2 million DOFs.

	Hybrid BESO				Soft-kill BESO			
	CLEB	MBBB	OLM	LSB	CLEB	MBBB	OLM	LSB
FEA	29.92	27.05	21.54	35.46	107.18	97.69	79.31	112.63
Filter	0.17	0.17	0.17	0.22	0.17	0.17	0.17	0.22
Update	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

vector-vector product (VVp) and vector linear transformations (Vlt). Figure 5.9 shows the execution times of these individual stages using soft-kill and hybrid BESO for all four benchmark problems. In all plots of Figure 5.9, different shades of blue and purple stacked bars are used to represent execution times for different stages. Along the secondary Y-axis, the corresponding speedup values for individual stages of FEA in hybrid BESO, are plotted. It can be seen that for all problems and using both soft-kill and hybrid BESO, afMVp stage is the most computationally expensive, followed by Vlt and VVp. It can also be seen that for all three stages of FEA, hybrid BESO outperforms soft-kill in terms of execution time. Among the three stages, the highest amount of speedup is achieved for the afMVp stage that ranges from $5\times - 8\times$ for different mesh sizes in different benchmark problem. The speedup value for the Vlt stage varies between $4\times - 6\times$ for different problem and different mesh sizes. Among the three stages, the least improvements are achieved for the VVp stage with approximately $3\times - 4\times$ speedup for different problems. It can be observed that PCG for LSB problem and CBEL problem consumes the most amount of time, whereas OLM problem consumes the least. Overall, the highest speedup values for afMVp, Vlt, and VVp stages are observed in the case of OLM problem.

Apart from the FEA stage, the other two key stages in BESO are sensitivity filter and structure update. These stages consume significantly less amount of time than the FEA stage. Due to acceleration of the entire BESO pipeline, the proportion of execution times consumed by these stages is further reduced when compared to implementations from the literature where only the FEA is accelerated on the GPU. Figure 5.10 shows the variation of execution time of the sensitivity filter and structure update stages for all four benchmark problems using different mesh sizes. Again, meshes with approximately the same number of DOFs are taken for different problems to ensure uniform comparison. Due to the disparity in the magnitudes, the sensitivity filter time and the structure update time are plotted separately along the primary and secondary Y-axes using green and blue colors, respectively. It can be seen that the sensitivity filter time increases exponentially with increasing mesh sizes. This does not create a problem in the present implementation, since even for the largest mesh sizes, the filter only consumes a tiny fraction of total execution time. The execution times for structure update are seen to increase linearly with increasing mesh sizes following a sharp rise at the beginning. These seem to be little variation in the execution times of these stages with respect to different problems and times for same mesh sizes are very close for all four problems.



(a) CBEL (b) MBBB

(c) OLM (d) LSB

Figure 5.9: Execution time and speedup comparison for PCG steps.

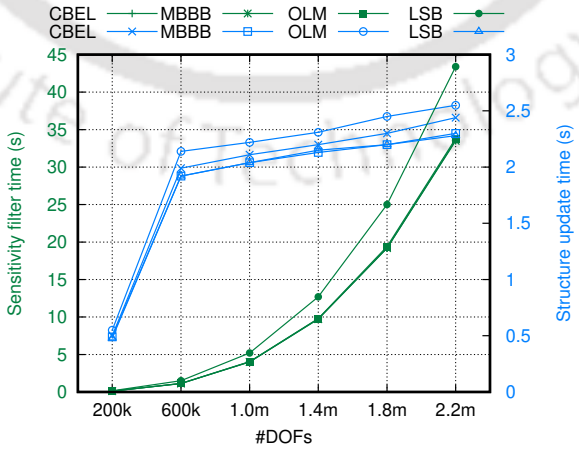


Figure 5.10: Sensitivity filter and structure update time for CBEL, MBBB, OLM, and LSB problems using hybrid BESO.

In order to properly understand the behaviour of hybrid BESO in the context of FEA, a series of plots is presented in Figure 5.11. Figure 13a shows the iteration-wise number of functional DOFs (n_{fDOF}) along the primary Y-axis in green. Along the secondary Y-axis, the corresponding iteration-wise execution times of the FEA stage in soft-kill and hybrid BESO are shown in blue. Further, a dotted line is used for soft-kill BESO and a solid line is used for hybrid BESO. From the figure, it can be observed that the number of functional DOFs keeps reducing uniformly till a pivot point at approximately iteration number 80, after which, it becomes almost constant. The corresponding values of the execution times for soft-kill BESO are seen to increase steadily till approximately the same pivot point, after which, a flat trend is observed. It is counter-intuitive that with the decrease in functional DOFs, the FEA time is seen to increase. The reason behind this increase is that at the beginning, the structure is completely solid and hence the resulting global stiffness matrix is well-conditioned, resulting in fast convergence of PCG. Afterwards, as void regions are introduced and links start to emerge, the PCG stage takes more and more iterations to converge. This is corroborated in Figure 13i for CBEL problem, that shows the iteration wise number of functional DOFs along with PCG iterations taken to converge in that particular optimization iteration. The PCG iterations are seen to increase for both soft-kill and hard-kill approaches steadily till the pivot point of functional DOFs is reached and, afterwards, flat trends are observed for both lines. Although, the PCG iterations do increase with increasing BESO iteration, the effective size of the domain keeps reducing due to the reduction in functional DOFs. This has a compensating effect on the FEA time of hybrid BESO. The increase in FEA time due to emerging void regions is compensated by reduction in FEA time caused by reduction in functional DOFs and therefore, reduction in the effective order of the global stiffness matrix. Due to this reason, in Figure 13a, the FEA time for hybrid BESO exhibits an overall flat trend throughout the BESO iterations and no steady increase is observed unlike soft-kill BESO. The same results for MBBB problem are shown in Figures 13b and 13ii. The results for the OLM problem are shown in Figures 13c and 13iii. The results for the LSB problem are shown in Figures 13d and 13iv. From these plots the same observation can be made that the hybrid BESO consumes less time and takes less number of iterations to converge for the FEA stage.

5.3 Closure

A GPU-adapted hybrid BESO method was presented that combined both the soft-kill and the hard-kill approaches. For the entire optimization algorithm, soft-kill approach was followed, whereas for the FEA stage, a hard-kill approach was adopted. The entire BESO method pipeline was accelerated on GPU using different granularity of threads and a combination of thrust library functions and custom kernels. FEA was identified to be the key stage in the entire parallelized BESO pipeline consuming majority of the total execution time. Steps in the JPCG-based FEA were further broken down into

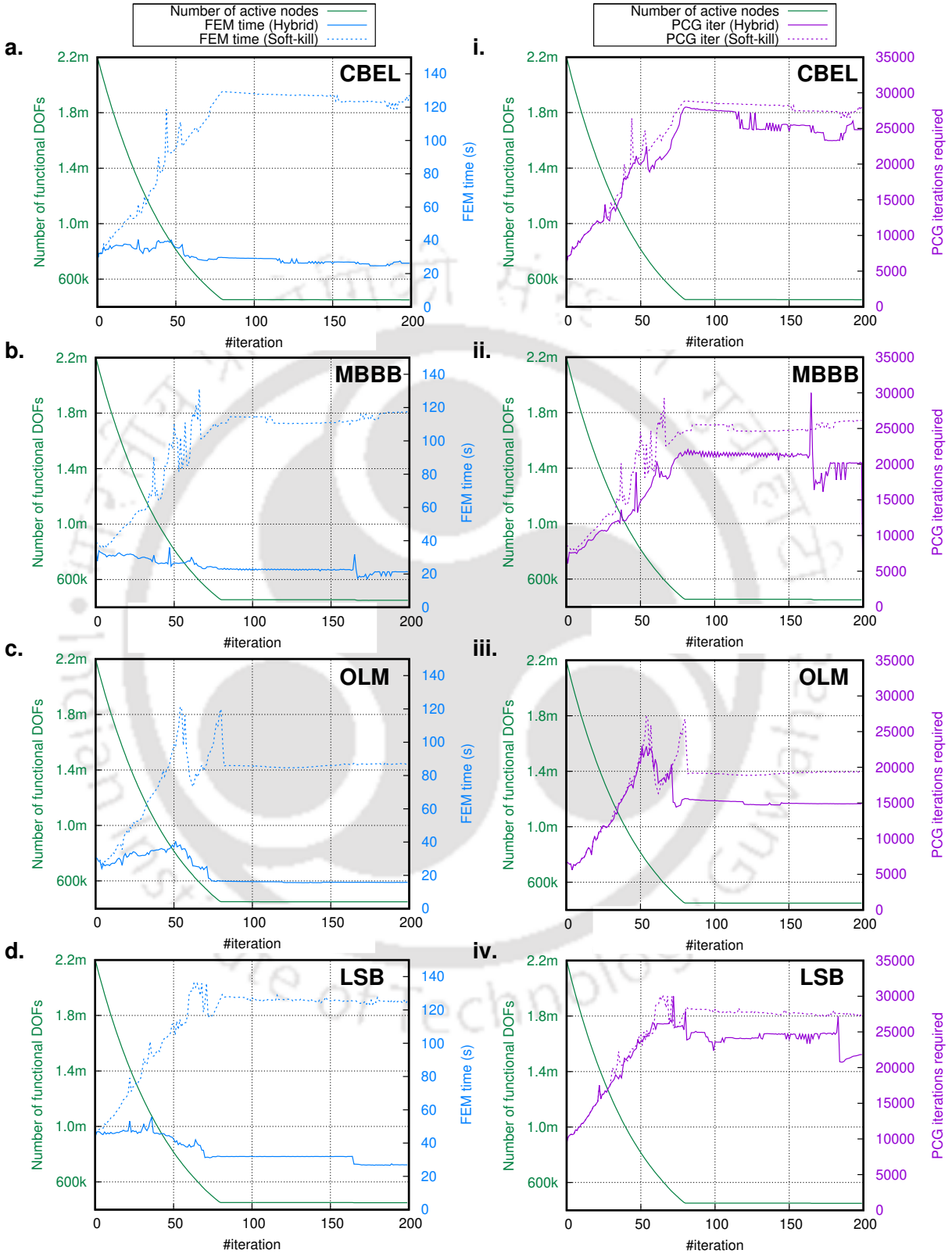


Figure 5.11: Iteration-wise execution times and PCG iterations for soft-kill and hybrid BESO.

simple linear algebra operations, afMVp, VVp, and Vlt, to parallelize them using a combination of thrust calls and custom kernels on GPU. A new numbering scheme was introduced with a parallel algorithm for computing the list of functional DOFs, required by the hard-kill FEA at every iteration of the optimization. The GPU-based hybrid BESO and GPU-based soft-kill BESO implementations were tested using four high-resolution benchmark problems having more than two million DOFs. The proposed hybrid BESO is found to produce optimal topologies for all four benchmark problems.

In the absence of a completely parallelized BESO implementation in the literature, the proposed hybrid BESO was compared against an entirely parallel soft-kill implementation developed by the authors. The performance analysis revealed significant benefits for the GPU-based hybrid BESO over the GPU-based soft-kill BESO in terms of execution times, global memory consumption, and the convergence of FEA. Comparison of the execution times showed speedups in the range of $2\times - 4\times$ for the hybrid method over soft-kill using all mesh sizes and all problems. Higher speedups were observed in general for higher mesh sizes having more number of DOFs. By comparing the timings for the different steps of JPCG in FEA, it was observed that for all problems, the most amount of speedup was achieved in the afMVp stage followed by the Vlt stage and VVp stage. In all problems, the memory requirements were seen to increase linearly with the increasing number of DOFs. Comparison of global memory usage for the soft-kill and hybrid approaches revealed up to 20% difference in memory consumed for higher mesh sizes of all four problems. The number of functional DOFs was seen to decrease steadily till a pivot point, after which, a flat trend was observed. In soft-kill, with the decrease in functional DOFs, the FEA time was seen to increase till the pivot point following which, a flat trend is observed. For hybrid BESO, on the other hand, the effect of increase in FEA time was compensated by decrease in effective mesh size, resulting in a flat trend throughout for all four problems. Furthermore, due to the removal of non-functional DOFs, FEA was seen to converge in less number of iterations in case of hybrid BESO compared to the soft-kill BESO for the same JPCG parameters.

Chapter 6

Conclusions and Future Work

In this thesis, different steps of FEA have been accelerated on GPU by incorporating novel algorithm-level and HPC-based modifications to the standard algorithms from the literature. As an important application of FEA, structural topology optimization was implemented on GPU with a focus on two density-based methods: SIMP and BESO. By using careful profiling of the application pipeline, the most time-consuming steps of both FEA and topology optimization were identified. Subsequently, by focusing on the building blocks of these important steps, the entire applications were accelerated with two important aims: reducing the computation time and reducing the memory footprint of the application.

6.1 Conclusions

Following are the important conclusions of this thesis.

- Proposed workload division strategy of FEA assembly outperforms the standard GPU-based assembly in terms of execution time and performance.
- The neighbor matrix introduced for the proposed matrix generation strategy is dependant on the mesh information alone, and can be easily adopted for different element types and order.
- It can be concluded that performing assembly and numerical integration in the same kernel perform worse than having separate kernels for the two processes. This difference is more pronounced for lower order elements.
- Proposed mesh reduction strategy dynamically reduces the effective number of design variables in density-based structural topology optimization resulting in reduced execution time and reduced memory footprint.

- Complete acceleration of the topology optimization pipeline minimizes CPU-GPU data transfer, resulting in faster execution.
- Proposed hybrid BESO outperforms the standard BESO on GPU by reducing execution time, reducing memory requirements, and improving FEA convergence.
- In all implementations of FEA and topology optimization in this thesis, the speedup and performance values are found to be more pronounced for large-scale problems with higher mesh sizes.
- Although significant speedups are observed for all GPU implementations of FEA and topology optimization, the values of achieved speedup vary with different class of problems even with the same mesh sizes.

Based on the observations from this thesis, an outline of the possible scopes of future studies is presented in the next section.

6.2 Future Work

- In this thesis, the focus is kept on GPU implementation of matrix generation and matrix-free solution of linear system of equations in FEA. For future work, the other stages, such as the mesh generation, model simplification, direct solvers, can be accelerated on GPU.
- In this thesis, the GPU-based mesh reduction strategy is successfully applied to SIMP and BESO method. In future, the same strategy can be adopted for other topology optimization methods such as level-set and GA to accelerate them on GPU.
- The methodologies proposed in this thesis can be extended to solve topology optimization problems involving multiple conflicting objectives. Since in this case, the solver is expected to still consume most of the total execution time, similar computational benefits can be expected. However, their actual efficacy needs to be examined for these classes of problems.
- For hybrid BESO, a penalty parameter value of 10 is suggested in this thesis, based on the experiments carried out with different benchmark examples, different problem sizes, and different BESO parameters. In order to obtain a comprehensive guideline for choosing the value of penalty parameter for an unknown problem, further studies need to be conducted.
- One shortcoming of this study is the use of relatively simple benchmark examples. While they are sufficient for establishing the efficacy of the proposed methods, more complex problems can be solved to bring out the advantages in a better manner.

- All the studies in this work are implemented on single GPU. This makes the scale of the implementations limited based on the GPU resources available. An MPI-based implementation which can distribute the workload on multiple nodes with multiple GPUs can massively broaden the scope of the implementations presented in this thesis.

References

Aage, N., E. Andreassen, and B. S. Lazarov (2015). Topology optimization using petsc: An easy-to-use, fully parallel, open source topology optimization framework. Structural and Multidisciplinary Optimization 51(3), 565–572.

Aage, N., E. Andreassen, B. S. Lazarov, and O. Sigmund (2017). Giga-voxel computational morphogenesis for structural design. Nature 550(7674), 84–86.

Abdelfattah, A., H. Ltaief, D. Keyes, and J. Dongarra (2016). Performance optimization of sparse matrix-vector multiplication for multi-component pde-based applications using gpus. Concurrency and Computation: Practice and Experience 28(12), 3447–3465.

Adams, L. and J. Ortega (1982). A multi-color SOR method for parallel computation. In ICPP, pp. 53–56. Citeseer.

Allaire, G., F. Jouve, and A.-M. Toader (2004). Structural optimization using sensitivity analysis and a level-set method. Journal of Computational Physics 194(1), 363–393.

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

Amir, O., N. Aage, and B. S. Lazarov (2014). On multigrid-cg for efficient topology optimization. Structural and Multidisciplinary Optimization 49(5), 815–829.

Banaś, K., F. Kruzel, and J. Bielański (2016). Finite element numerical integration for first order approximations on multi- and many-core architectures. Computer Methods in Applied Mechanics and Engineering 305, 827 – 848.

Banaś, K., P. Plaszewski, and P. Maciol (2014). Numerical integration on GPUs for higher order finite elements. Computers & Mathematics with Applications 67(6), 1319 – 1344.

Bell, N. and M. Garland (2008, December). Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation.

Bendsøe, M. P. (1989). Optimal shape design as a material distribution problem. Structural optimization 1(4), 193–202.

Bendsøe, M. P. and O. Sigmund (2003). Topology optimization: Theory, methods and applications (2 ed.). Springer-Verlag Berlin Heidelberg.

Bendsøe, M. P. and N. Kikuchi (1988). Generating optimal topologies in structural design using a homogenization method. Computer Methods in Applied Mechanics and Engineering 71(2), 197 – 224.

Bolz, J., I. Farmer, E. Grinspun, and P. Schröoder (2003, July). Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans. Graph. 22(3), 917–924.

Borrvall, T. and J. Petersson (2001). Large-scale topology optimization in 3d using parallel computing. Computer Methods in Applied Mechanics and Engineering 190(46), 6201 – 6229.

Botsch, M., D. Bommes, C. Vogel, and L. Kobbelt (2004, Oct). GPU-based tolerance volumes for mesh processing. In Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on, pp. 237–243.

Boubekeur, T. and M. Alexa (2009). Mesh simplification by stochastic sampling and topological clustering. Computer and Graphics – Special Issue on IEEE Shape Modeling International 2009 33, 241–249.

Bruns, T. E. (2007). Topology optimization of convection-dominated, steady-state heat transfer problems. International Journal of Heat and Mass Transfer 50(15-16), 2859–2873.

Buatois, L., G. Caumon, and B. Levy (2009, June). Concurrent number cruncher: A GPU implementation of a general sparse linear solver. International Journal of Parallel, Emergent and Distributed Systems 24(3), 205–223.

Cai, Y., G. Li, and H. Wang (2013). A parallel node-based solution scheme for implicit finite element method using GPU. Procedia Engineering 61, 318 – 324.

Cantwell, C., S. Sherwin, R. Kirby, and P. Kelly (2011). From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. Computers & Fluids 43(1), 23 – 28.

Cao, T.-T., A. Nanjappa, M. Gao, and T.-S. Tan (2014). A GPU accelerated algorithm for 3D Delaunay Triangulation. In Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '14, New York, NY, USA, pp. 47–54. ACM.

Cecka, C., A. J. Lew, and E. Darve (2011). Assembly of finite element methods on graphics processors. International Journal for Numerical Methods in Engineering 85(5), 640–669.

Cevahir, A., A. Nukada, and S. Matsuoka (2009). Fast conjugate gradients with multiple gpus. In G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. A. Sloot (Eds.), Computational Science – ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I, pp. 893–903. Berlin, Heidelberg: Springer Berlin Heidelberg.

Cevahir, A., A. Nukada, and S. Matsuoka (2010). High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. Computer Science-Research and Development 25(1-2), 83–91.

Challis, V. J., A. P. Roberts, and J. F. Grotowski (2014). High resolution topology optimization using graphics processing units (GPUs). Structural and Multidisciplinary Optimization 49(2), 315–325.

Christiansen, A. N., M. Nobel-Jørgensen, N. Aage, O. Sigmund, and J. A. Bærentzen (2014). Topology optimization using an explicit interface representation. Structural and Multidisciplinary Optimization 49(3), 387–399.

Cucinotta, F., E. Guglielmino, G. Longo, G. Risitano, D. Santonocito, and F. Sfravara (2019). Topology optimization additive manufacturing-oriented for a biomedical application. In Advances on Mechanics, Design Engineering and Manufacturing II, pp. 184–193. Springer.

De Troya, M. A. S. and D. A. Tortorelli (2018). Adaptive mesh refinement in stress-constrained topology optimization. Structural and Multidisciplinary Optimization 58(6), 2369–2386.

Deaton, J. D. and R. V. Grandhi (2014). A survey of structural and multidisciplinary continuum topology optimization: post 2000. Structural and Multidisciplinary Optimization 49(1), 1–38.

DeCoro, C. and N. Tatarchuk (2007). Real-time mesh simplification using the gpu. In Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games, I3D '07, New York, NY, USA, pp. 161–166. ACM.

Dinh, Q. and Y. Marechal (2016, March). Toward real-time finite-element simulation on gpu. IEEE Transactions on Magnetics 52(3), 1–4.

Duan, X.-B., F.-F. Li, and X.-Q. Qin (2015). Adaptive mesh method for topology optimization of fluid flow. Applied Mathematics Letters 44, 40–44.

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

Dühring, M. B., J. S. Jensen, and O. Sigmund (2008). Acoustic design by topology optimization. Journal of sound and vibration 317(3-5), 557–575.

Dziekonski, A., P. Sypek, A. Lamecki, and M. Mrozowski (2012a). Accuracy, memory, and speed strategies in GPU-based finite-element matrix-generation. IEEE Antennas and Wireless Propagation Letters 11, 1346–1349.

Dziekonski, A., P. Sypek, A. Lamecki, and M. Mrozowski (2012b). Finite element matrix generation on a GPU. Progress In Electromagnetics Research 128, 249–265.

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

Farhat, C. and L. Crivelli (1989). A general approach to nonlinear FE computations on shared-memory multiprocessors. Computer Methods in Applied Mechanics and Engineering 72(2), 153–171.

Filipovic, J., I. Peterlik, and J. Fousek (2009). Gpu acceleration of equations assembly in finite elements method-preliminary results. In SAAHPC: Symposium on Application Accelerators in HPC.

Fu, Z., T. J. Lewis, R. M. Kirby, and R. T. Whitaker (2014). Architecting the finite element method pipeline for the gpu. Journal of Computational and Applied Mathematics 257, 195 – 211.

George, T., V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury (2011). Multifrontal factorization of sparse SPD matrices on GPUs. In Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, pp. 372–383. IEEE.

Georgescu, S., P. Chow, and H. Okuda (2013). GPU acceleration for FEM-based structural analysis. Archives of Computational Methods in Engineering 20(2), 111–121.

Georgescu, S. and H. Okuda (2010). Conjugate gradients on multiple GPUs. International Journal for Numerical Methods in Fluids 64(10-12), 1254–1273.

Geveler, M., D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek (2011). Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. In Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering, pp. 22.

Göddeke, D., R. Strzodka, and S. Turek (2005). Accelerating double precision FEM simulations with GPUs. Univ. Dortmund, Fachbereich Mathematik.

Göddeke, D., R. Strzodka, and S. Turek (2007). Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in fem simulations. International Journal of Parallel, Emergent and Distributed Systems 22(4), 221–256.

Gogu, C. (2015). Improving the efficiency of large scale topology optimization through on-the-fly reduced order model construction. International Journal for Numerical Methods in Engineering 101(4), 281–304.

Gribanov, I., R. Taylor, and R. Sarracino (2018). Parallel implementation of implicit finite element model with cohesive zones and collision response using cuda. International Journal for Numerical Methods in Engineering 115(7), 771–790.

Helfenstein, R. and J. Koko (2012). Parallel preconditioned conjugate gradient algorithm on GPU. Journal of Computational and Applied Mathematics 236(15), 3584 – 3590.

Herrero, D., J. Martínez, and P. Martí (2013). An implementation of level set based topology optimization using gpu. In 10th World Congress on Structural and Multidisciplinary Optimization, Orlando, Florida, USA, pp. 1–10.

Hjelmervik, J. and J. C. Léon (2007, June). GPU-accelerated shape simplification for mechanical-based applications. In Shape Modeling and Applications, 2007. SMI '07. IEEE International Conference on, pp. 91–102.

Hjelmervik, J. M. and J.-C. Léon (2010). Simplification of fem-models on cell be. In M. Dæhlen, M. Floater, T. Lyche, J.-L. Merrien, K. Mørken, and L. L. Schumaker (Eds.), Mathematical Methods for Curves and Surfaces: 7th International Conference, MMCS 2008, Tønsberg, Norway, June 26-July 1, 2008, Revised Selected Papers, pp. 261–273. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ho-Le, K. (1988). Finite element mesh generation methods: a review and classification. Computer-Aided Design 20(1), 27 – 38.

Huang, X. and M. Xie (2010a). Evolutionary topology optimization of continuum structures: methods and applications. John Wiley & Sons.

Huang, X. and Y. Xie (2008). A new look at eso and beso optimization methods. Structural and Multidisciplinary Optimization 35(1), 89–92.

Huang, X. and Y. M. Xie (2009). Bi-directional evolutionary topology optimization of continuum structures with one or multiple materials. Computational Mechanics 43(3), 393–401.

Huang, X. and Y.-M. Xie (2010b). A further review of eso type methods for topology optimization. Structural and Multidisciplinary Optimization 41(5), 671–683.

Hughes, T. J., R. M. Ferencz, and J. O. Hallquist (1987). Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients. Computer Methods in Applied Mechanics and Engineering 61(2), 215 – 248.

Ito, Y., A. M. Shih, A. K. Erukala, B. K. Soni, A. Chernikov, N. P. Chrisochoides, and K. Nakahashi (2007). Parallel unstructured mesh generation by an advancing front method. Mathematics and Computers in Simulation 75(5–6), 200 – 209.

Jansen, M., G. Lombaert, M. Schevenels, and O. Sigmund (2014). Topology optimization of fail-safe structures using a simplified local damage model. Structural and Multidisciplinary Optimization 49(4), 657–666.

Kim, S. Y., I. Y. Kim, and C. K. Mechefske (2012). A new efficient convergence criterion for reducing computational expense in topology optimization: reducible design variable method. International Journal for Numerical Methods in Engineering 90(6), 752–783.

Kiran, U., D. Sharma, and S. S. Gautam (2018). Gpu-warp based finite element matrices generation and assembly using coloring method. Journal of Computational Design and Engineering 6(4), 705–718.

Kirk, D. B. and W.-m. W. Hwu (2010). Programming Massively Parallel Processors: A Hands-on Approach (1st ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Kirk, D. B. and W. H. Wen-mei (2012). Programming massively parallel processors: a hands-on approach. Newnes.

Kiss, I., S. Gyimothy, Z. Badics, and J. Pavo (2012, Feb). Parallel realization of the element-by-element FEM technique by CUDA. IEEE Transactions on Magnetics 48(2), 507–510.

Knepley, M. G. and A. R. Terrel (2013, feb). Finite element integration on gpus. ACM Trans. Math. Softw. 39(2).

Komatitsch, D., D. Michéa, and G. Erlebacher (2009). Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. Journal of Parallel and Distributed Computing 69(5), 451 – 460.

Krawezik, G. P. and G. Poole (2009). Accelerating the ANSYS direct sparse solver with GPUs. Proc. Symposium on Application Accelerators in High Performance Computing (SAAHPC). NCSA, Urbana-Champaign, IL.

Krishnan, S. (2013). Generating 3d topologies with multiple constraints on the gpu. In 10th World congress on structural and multidisciplinary optimization, pp. 1–9.

Krüger, J. and R. Westermann (2003). Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics (TOG) 22(3), 908–916.

Li, R. and Y. Saad (2013). GPU-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing 63(2), 443–466.

Li, Y., B. Zhou, and X. Hu (2021). A two-grid method for level-set based topology optimization with gpu-acceleration. Journal of Computational and Applied Mathematics 389, 113336.

Liao, Z., Y. Zhang, Y. Wang, and W. Li (2019). A triple acceleration method for topology optimization. Structural and Multidisciplinary Optimization 60(2), 727–744.

Liu, H., Y. Tian, H. Zong, Q. Ma, M. Y. Wang, and L. Zhang (2019). Fully parallel level set method for large-scale structural topology optimization. Computers & Structures 221, 13–27.

Lucas, R. F., G. Wagenbreth, D. M. Davis, and R. Grimes (2010). Multifrontal computations on GPUs and their multi-core hosts. In International Conference on High Performance Computing for Computational Science, pp. 71–82. Springer.

Maciol, P., P. Plaszewski, and K. Banaś (2010). 3D finite element numerical integration on GPUs. Procedia Computer Science 1(1), 1093 – 1100. ICCS 2010.

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

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

Martínez-Frutos, J. and D. Herrero-Pérez (2015). Efficient matrix-free GPU implementation of fixed grid finite element analysis. Finite Elements in Analysis and Design 104, 61 – 71.

Martínez-Frutos, J. and D. Herrero-Pérez (2017). Gpu acceleration for evolutionary topology optimization of continuum structures using isosurfaces. Computers & Structures 182, 119–136.

Martínez-Frutos, J., P. J. Martínez-Castejón, and D. Herrero-Pérez (2015). Fine-grained GPU implementation of assembly-free iterative solver for finite element problems. Computers & Structures 157, 9 – 18.

Martínez-Frutos, J., P. J. Martínez-Castejón, and D. Herrero-Pérez (2017). Efficient topology optimization using gpu computing with multilevel granularity. Advances in Engineering Software 106, 47–62.

Martínez-Frutos, J. and D. Herrero-Pérez (2016). Large-scale robust topology optimization using multi-gpu systems. Computer Methods in Applied Mechanics and Engineering 311, 393 – 414.

Maute, K. and E. Ramm (1995). Adaptive topology optimization. Structural optimization 10(2), 100–112.

Misztal, M. K. and J. A. Bærentzen (2012). Topology-adaptive interface tracking using the deformable simplicial complex. ACM Transactions on Graphics (TOG) 31(3), 1–12.

Mlejnek, H. (1992). Some aspects of the genesis of structures. Structural optimization 5(1-2), 64–69.

Mukherjee, S., D. Lu, B. Raghavan, P. Breitkopf, S. Dutta, M. Xiao, and W. Zhang (2021). Accelerating large-scale topology optimization: State-of-the-art and challenges. Archives of Computational Methods in Engineering 28(7), 4549–4571.

Munk, D. J., T. Kipouros, and G. A. Vio (2019). Multi-physics bi-directional evolutionary topology optimization on gpu-architecture. Engineering with Computers 35(3), 1059–1079.

Munk, D. J., T. Kipouros, G. A. Vio, G. T. Parks, and G. P. Steven (2018). Multiobjective and multiphysics topology optimization using an updated smart normal constraint bi-directional evolutionary structural optimization method. Structural and Multidisciplinary Optimization 57(2), 665–688.

Munk, D. J., T. Kipouros, G. A. Vio, G. P. Steven, and G. T. Parks (2017). Topology optimisation of micro fluidic mixers considering fluid-structure interactions with a coupled lattice boltzmann algorithm. Journal of Computational Physics 349, 11–32.

Nair, A. U., D. G. Taggart, and F. J. Vetter (2007). Optimizing cardiac material parameters with a genetic algorithm. Journal of Biomechanics 40(7), 1646–1650.

Naumov, M. (2011). Incomplete-LU and cholesky preconditioned iterative methods using CUSPARSE and CUBLAS.

Nguyen, T. H., G. H. Paulino, J. Song, and C. H. Le (2010). A computational paradigm for multiresolution topology optimization (mtop). Structural and Multidisciplinary Optimization 41(4), 525–539.

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

Qi, M., T. T. Cao, and T. S. Tan (2013, May). Computing 2D constrained Delaunay Triangulation using the GPU. IEEE Transactions on Visualization and Computer Graphics 19(5), 736–748.

Ram, L. and D. Sharma (2017). Evolutionary and gpu computing for topology optimization of structures. Swarm and Evolutionary Computation 35, 1–13.

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

Ratnakar, S. K., S. Sanfui, and D. Sharma (2020). Simp-based structural topology optimization using unstructured mesh on gpu. In N. Kumar, S. Tibor, R. Sindhwani, and P. Srivastava (Eds.), Advances in Interdisciplinary Engineering: Select Proceedings of FLAME 2020, pp. 1–10. Springer.

Ratnakar, S. K., S. Sanfui, and D. Sharma (2021a, 12). Graphics Processing Unit-Based Element-by-Element Strategies for Accelerating Topology Optimization of Three-Dimensional Continuum Structures Using Unstructured All-Hexahedral Mesh. Journal of Computing and Information Science in Engineering 22(2). 021013.

Ratnakar, S. K., S. Sanfui, and D. Sharma (2021b, 12). Graphics Processing Unit-Based Element-by-Element Strategies for Accelerating Topology Optimization of Three-Dimensional Continuum Structures Using Unstructured All-Hexahedral Mesh. Journal of Computing and Information Science in Engineering 22(2). 021013.

Reguly, I. Z. and M. B. Giles (2015). Finite element algorithms and data structures on graphical processing units. International Journal of Parallel Programming 43(2), 203–239.

Remacle, J.-F., R. Gandham, and T. Warburton (2016). GPU accelerated spectral finite elements on all-hex meshes. Journal of Computational Physics 324, 246 – 257.

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

Rong, G., T.-S. Tan, T.-T. Cao, and Stephanus (2008). Computing two-dimensional Delaunay Triangulation using graphics hardware. In Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, I3D '08, New York, NY, USA, pp. 89–97. ACM.

Sanfui, S. and D. Sharma (2017a, Feb). A two-kernel based strategy for performing assembly in fea on the graphics processing unit. In AMIAMS (Ed.), 2017 International Conference on Advances in Mechanical, Industrial, Automation and Management Systems (AMIAMS), pp. 1–9.

Sanfui, S. and D. Sharma (2017b). A two-kernel based strategy for performing assembly in FEA on the graphics processing unit. In IEEE International Conference on Advances in Mechanical, Industrial, Automation and Management Systems(AMIAMS 2017), pp. 1–7.

Sanfui, S. and D. Sharma (2019, July). Exploiting symmetry in elemental computation and assembly stage of gpu-accelerated fea. In G. X. G. R. Liu, Fangsen Cui (Ed.), Proceedings at the 10th International Conference on Computational Methods (ICCM2019), Volume 6, pp. 641–651. ScienTech Publisher.

Sanfui, S. and D. Sharma (2021, 07). Symbolic and Numeric Kernel Division for Graphics Processing Unit-Based Finite Element Analysis Assembly of Regular Meshes With Modified Sparse Storage Formats. Journal of Computing and Information Science in Engineering 22(1).

Sao, P., R. Vuduc, and X. S. Li (2014). A distributed CPU-GPU sparse direct solver. In Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings, pp. 487–498. Cham: Springer International Publishing.

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

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

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

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

Shewchuk, J. R. (1994, 8). An introduction to the conjugate gradient method without the agonizing pain. Technical Report 1, Carnegie Mellon University, School of Computer Science, CMU. https://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf.

Sigmund, O. (2001). A 99 line topology optimization code written in matlab. Structural and Multidisciplinary Optimization 21(2), 120–127.

Tang, Y., A. Kurtz, and Y. F. Zhao (2015). Bidirectional evolutionary structural optimization (beso) based design method for lattice structure to be fabricated by additive manufacturing. Computer-Aided Design 69, 91–101.

Tao, Y., Y. Deng, S. Mu, Z. Zhang, M. Zhu, L. Xiao, and L. Ruan (2015). Gpu accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication. Concurrency and Computation: Practice and Experience 27(14), 3771–3789.

Thakur, A., A. G. Banerjee, and S. K. Gupta (2009). A survey of CAD model simplification techniques for physics-based simulation applications. Computer-Aided Design 41(2), 65 – 80.

Tian, J., G. Li, W. Fei, and G. Zeng (2013). An efficient GPU acceleration format for finite element analysis. Journal of Electronics (China) 30(6), 599–608.

Valdez, S. I., S. Botello, M. A. Ochoa, J. L. Marroquín, and V. Cardoso (2017). Topology optimization benchmarks in 2d: Results for minimum compliance and minimum volume in planar stress problems. Archives of Computational Methods in Engineering 24(4), 803–839.

Venkataraman, S., M. Sohoni, and G. Elber (2001). Blend recognition algorithm and applications. In Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications, SMA '01, New York, NY, USA, pp. 99–108. ACM.

Wadbro, E. and M. Berggren (2009, November). Megapixel topology optimization on a graphics processing unit. SIAM Review 51(4), 707–721.

Wang, M., H. Klie, M. Parashar, and H. Sudan (2009a). Solving sparse linear systems on NVIDIA Tesla GPUs. In Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09, Berlin, Heidelberg, pp. 864–873. Springer-Verlag.

Wang, M., H. Klie, M. Parashar, and H. Sudan (2009b). Solving sparse linear systems on NVIDIA Tesla GPUs. In G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. A. Sloot (Eds.), Computational Science – ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I, pp. 864–873. Berlin, Heidelberg: Springer Berlin Heidelberg.

Wang, M. Y., X. Wang, and D. Guo (2003). A level set method for structural topology optimization. Computer methods in applied mechanics and engineering 192(1-2), 227–246.

Wang, Y., Z. Liao, M. Ye, Y. Zhang, W. Li, and Z. Xia (2020). An efficient isogeometric topology optimization using multilevel mesh, mgcg and local-update strategy. Advances in Engineering Software 139, 102733.

Woźniak, M. (2015). Fast gpu integration algorithm for isogeometric finite element method solvers using task dependency graphs. Journal of Computational Science 11, 145–152.

Woźniak, M., K. Kuźnik, M. Paszyński, V. Calo, and D. Pardo (2014). Computational cost estimates for parallel shared memory isogeometric multi-frontal solvers. Computers & Mathematics with Applications 67(10), 1864 – 1883.

Wu, W. and P. A. Heng (2004). A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting. Computer Animation and Virtual Worlds 15(3-4), 219–227.

Xiao, M., D. Lu, P. Breitkopf, B. Raghavan, S. Dutta, and W. Zhang (2020). On-the-fly model reduction for large-scale structural topology optimization using principal components analysis. Structural and Multidisciplinary Optimization 62, 209–230.

Xie, Y. and G. Steven (1993). A simple evolutionary procedure for structural optimization. Computers & Structures 49(5), 885–896.

Yoo, J. and I. Lee (2017). Efficient density based topology optimization using dual-layer element and variable grouping method for large 3d applications. In World Congress of Structural and Multidisciplinary Optimisation, pp. 967–978. Springer.

Zayer, R., M. Steinberger, and H. Seidel (2017, Sep.). Sparse matrix assembly on the gpu through multiplication patterns. In HPEC (Ed.), 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–8.

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

Zhou, M. and G. Rozvany (1991). The coc algorithm, part ii: Topological, geometrical and generalized shape optimization. Computer Methods in Applied Mechanics and Engineering 89(1), 309–336. Second World Congress on Computational Mechanics.

Zienkiewicz, O. C., R. L. Taylor, and R. Lee (1977). The finite element method, Volume 3. McGraw hill London.